# Unit 6: Socket For Clients

A socket is an endpoint for communication between two machines. It is a connection between two hosts.

It can perform seven basic operations:

- Connect to a remote machine
- Send data
- Receive data
- Close a connection
- Bind to a port
- Listen for incoming data
- Accept connections from remote machines on the bound port

## Using Sockets

### Investigating Protocols with Telnet:

Telnet is a network protocol that provides a command-line interface for communication with remote devices or servers. It can be used to manually interact with servers, test network services, and debug network protocols. Although Telnet is largely replaced by more secure alternatives like SSH, it remains a useful tool for network diagnostics and testing.

1. **Basic Telnet Usage:** To start a Telnet session, you can use the Telnet client command followed by the server address and the port number. For example, to connect to a web server on port 80 (HTTP), you would use:

   telnet example.com 80

2. **Manually Sending Requests:** Once connected, you can manually type commands or requests according to the protocol. For example, for HTTP, you might type:

   GET / HTTP/1.1
   Host: example.com

## Reading from Servers with Sockets:

Reading from servers using sockets involves establishing a connection, sending a request, and then reading the response. This process is fundamental in network programming and is commonly used in client-server architectures, such as web browsers communicating with web servers.

### Steps for Reading from Servers with Sockets

1. **Create a Socket:** First, create a socket using the appropriate API in your programming language. For example, in Java, you would use the Socket class.
2. **Connect to the Server:** Use the socket to connect to the server's IP address and port number. For instance, to connect to a web server, you typically use port 80 (HTTP) or port 443 (HTTPS).
3. **Send a Request:**Once connected, you send a request to the server. The format of this request depends on the protocol being used. For example, an HTTP request might look like:

   ```
   GET / HTTP/1.1
   Host: example.com
   ```

4. **Read the Response:** After sending the request, read the server's response from the socket's input stream. The response typically includes headers and content, which you can parse according to the protocol's specifications.
5. **Close the Socket:** Once communication is complete, close the socket to free up system resources.

Basic example in Java that demonstrates an HTTP request(Client side socket):

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

public class SocketExample {
    public static void main(String[] args) {
        String host = "example.com";
        int port = 80;

        try (Socket socket = new Socket(host, port)) {
            // Send request
            OutputStream output = socket.getOutputStream();
            String request = "GET / HTTP/1.1\r\n" +
                             "Host: " + host + "\r\n" +
                             "Connection: close\r\n" +
                             "\r\n";
            output.write(request.getBytes());
            output.flush();

            // Read response
            BufferedReader reader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

### Writing to the Servers with Sockets:

Writing to servers using sockets involves sending data from a client to a server over a network. This process typically includes establishing a socket connection, sending a request or data to the server, and optionally receiving a response. In many cases, writing to a server could involve sending HTTP requests, uploading files, or sending data for processing.

Example:

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

public class SocketExample {
    public static void main(String[] args) {
        String host = "example.com";
        int port = 80;

        try (Socket socket = new Socket(host, port)) {
            // Sending data to the server
            OutputStream output = socket.getOutputStream();

            // Prepare the data you want to send
            String postData = "key1=value1&key2=value2";
            String request = "POST /submit HTTP/1.1\r\n" +
                    "Host: " + host + "\r\n" +
                    "Content-Type: application/x-www-form-urlencoded\r\n" +
                    "Content-Length: " + postData.length() + "\r\n" +
                    "Connection: close\r\n" +
                    "\r\n" +
                    postData;

            output.write(request.getBytes());
```

```
            output.flush();

            // Reading response from the server
            BufferedReader reader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

## Constructing and Connecting Sockets:

The java.net.Socket class is Java's fundamental class for performing client-side TCP operations. Other client-oriented classes that make TCP network connections such as URL, URLConnection, Applet, and JEditorPane all ultimately end up invoking the methods of this class. This class itself uses native code to communicate with the local TCP stack of the host operating system.

### Basic Constructors:

Each Socket constructor specifies the host and the port to connect to. Hosts may be specified as an InetAddress or a String. Remote ports are specified as int values from 1 to 65535:

```
public Socket(String host, int port) throws UnknownHostException, IOException
public Socket(InetAddress host, int port) throws IOException
```

For example:

```
try {
  Socket toOReilly = new Socket("www.oreilly.com", 80);
  // send and receive data...
} catch (UnknownHostException ex) {
  System.err.println(ex);
} catch (IOException ex) {
  System.err.println(ex);
}
```

**Question:** write a program to perform a basic two way communication between client and server. (Solution is below these client and server program)

**Create the Server Program**

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

public class SimpleServer {
    public static void main(String[] args) {
        int port = 5000;  // Choose an available port number

        try (ServerSocket serverSocket = new ServerSocket(port)) {
            System.out.println("Server is listening on port " + port);

            while (true) {
                Socket socket = serverSocket.accept();
                System.out.println("New client connected");

                // Read data sent by the client
                BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
                String message = in.readLine();
                System.out.println("Received: " + message);

                // Send response to the client
                PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
                out.println("Hello, Client!");
```

```
            // Close the connection
            in.close();
            out.close();
            socket.close();
        }
    } catch (IOException e) {
        System.err.println("IOException: " + e.getMessage());
    }
  }
}
```

**Create the Client Program**

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

public class SimpleClient {
   public static void main(String[] args) {
      String hostname = "localhost"; // or the IP address of the server
      int port = 5000;              // Same port as the server

      try (Socket socket = new Socket(hostname, port)) {
         System.out.println("Connected to the server");

         // Send data to the server
         PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
         out.println("Hello, Server!");

         // Receive response from the server
         BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
         String response = in.readLine();
         System.out.println("Server response: " + response);

      } catch (UnknownHostException e) {
         System.err.println("Unknown host: " + hostname);
      } catch (IOException e) {
         System.err.println("IOException: " + e.getMessage());
      }
```

```
  }
}
```

**Picking a Local Interface to Connect From:**

In Java, two constructors for the Socket class allow specifying both the remote host and port, as well as the local network interface and port from which to connect:

```
public Socket(String host, int port, InetAddress interface, int localPort)
      throws IOException, UnknownHostException
public Socket(InetAddress host, int port, InetAddress interface, int localPort)
      throws IOException
```

These constructors are used to establish a connection from a specific network interface and port. The local interface can be a physical device (like an Ethernet card) or a virtual one (on a multi-homed host). If the localPort is set to 0, Java automatically selects a random available port in the range 1024-65535.

Choosing a specific local network interface can be useful, especially in systems like routers or firewalls with multiple interfaces, where you may want to control which network path outgoing data uses. For example, a program might need to send error logs from a specific internal network interface to an internal mail server.

These constructors can throw IOException or UnknownHostException if there are issues connecting to the specified host or binding to the local interface. A common IOException subtype that might be encountered is BindException, which occurs if the program cannot bind to the requested local network interface. This behavior could be leveraged to enforce software licensing by restricting a program to run on specific hosts, though this method is not foolproof due to the ease of reverse-engineering Java programs.

**Constructing Without Connecting:**

In Java, creating a Socket object without immediately establishing a network connection. This is achieved using the no-argument constructor Socket(), which initializes the socket but does not connect it to any remote host.

```
public Socket()
```

You can connect later by passing a SocketAddress to one of the connect() methods. For example:

```
Socket socket = new Socket();
SocketAddress address = new InetSocketAddress("time.nist.gov", 13);
socket.connect(address);
```

This approach offers more control over socket initialization and connection, making it useful for specific networking needs and cleaner code management.

**Socket Addresses:**

The SocketAddress class in Java represents a connection endpoint but is abstract and does not include methods beyond a default constructor. It is primarily used to store transient connection details like IP address and port, and it is typically associated with TCP/IP sockets. The InetSocketAddress class, a concrete subclass of SocketAddress, is commonly used in practice.

Methods for SocketAddress:

- getRemoteSocketAddress(): Returns the address of the remote system connected to the socket.
- getLocalSocketAddress(): Returns the local address from which the socket is connected.

Both methods return null if the socket is not connected.

**Example:** This program will create a socket, connect to a remote server, retrieve and print the local and remote socket addresses, and then reconnect using the stored remote address.

```
import java.io.IOException;
import java.net.InetAddress;
import java.net.InetSocketAddress;
import java.net.Socket;
import java.net.SocketAddress;
```

```
public class SocketAddressExample {

    public static void main(String[] args) {
        // Define remote server details
        String host = "www.example.com";
        int port = 80;

        try {
            // Create a new socket
            Socket socket = new Socket();

            // Connect to the remote server
            InetSocketAddress remoteAddress = new InetSocketAddress(host, port);
            socket.connect(remoteAddress);

            // Get and print the local and remote addresses
            SocketAddress localSocketAddress = socket.getLocalSocketAddress();
            SocketAddress remoteSocketAddress = socket.getRemoteSocketAddress();

            System.out.println("Local Socket Address: " + localSocketAddress);
            System.out.println("Remote Socket Address: " + remoteSocketAddress);

            // Close the first socket
            socket.close();

            // Reconnect using the stored remote address
            Socket socket2 = new Socket();
            socket2.connect(remoteSocketAddress);

            // Print the new connection's remote address
            System.out.println("Reconnected to Remote Socket Address: " +
socket2.getRemoteSocketAddress());

            // Close the second socket
            socket2.close();

        } catch (IOException ex) {
            ex.printStackTrace();
        }
```

```
    }
}
```

**Proxy Servers:**

The constructor public Socket(Proxy proxy) creates an unconnected socket that can use a specified proxy server for connections. By default, sockets use system-wide proxy settings defined by socksProxyHost and socksProxyPort properties. However, this constructor allows you to specify a proxy server, such as a SOCKS proxy, directly. You can also use Proxy.NO_PROXY to bypass all proxies and connect directly, though this will fail if a firewall blocks direct connections.

For example, to connect to a remote host via a SOCKS proxy at "myproxy.example.com" on port 1080, you can use:

```
SocketAddress proxyAddress = new InetSocketAddress("myproxy.example.com", 1080);
Proxy proxy = new Proxy(Proxy.Type.SOCKS, proxyAddress);
Socket s = new Socket(proxy);
SocketAddress remote = new InetSocketAddress("login.ibiblio.org", 25);
s.connect(remote);
```

## Getting Information About a Socket

Socket objects have several properties that are accessible through getter methods:

- Remote address
- Remote port
- Local address
- Local port

Here are the getter methods for accessing these properties:

```
public InetAddress getInetAddress()
public int getPort()
public InetAddress getLocalAddress()
public int getLocalPort()
```

There are no setter methods. These properties are set as soon as the socket connects, and are fixed from there on.
The getInetAddress() and getPort() methods tell you the remote host and port the Socket is connected to; or, if the connection is now closed, which host and port the Socket was connected to when it was connected. The getLocalAddress() and getLocalPort() methods tell you the network interface and port the Socket is connected from.

Example: Get a socket's information

```
import java.net.*;
import java.io.*;
public class SocketInfo {
 public static void main(String[] args) {
 for (String host : args) {
 try {
 Socket theSocket = new Socket(host, 80);
 System.out.println("Connected to " + theSocket.getInetAddress()
 + " on port " + theSocket.getPort() + " from port "
 + theSocket.getLocalPort() + " of "
 + theSocket.getLocalAddress());
 } catch (UnknownHostException ex) {
 System.err.println("I can't find " + host);
 } catch (SocketException ex) {
 System.err.println("Could not connect to " + host);
 } catch (IOException ex) {
 System.err.println(ex);
 }
 }
 }
}
```

**Closed or Connected?**

The isClosed() method returns true if the socket is closed, false if it isn't. If you're uncertain about a socket's state, you can check it with this method rather than risking an IOException. For example:

```
if (socket.isClosed()) {
…….
…….
} else {
…….
……
}
```

However, this is not a perfect test. If the socket has never been connected in the first place, isClosed() returns false, even though the socket isn't exactly open.

The Socket class also has an isConnected() method. The name is a little misleading. It does not tell you if the socket is currently connected to a remote host (like if it is unclosed). Instead, it tells you whether the socket has ever been connected to a remote host.

**toString():**

The Socket class overrides only one of the standard methods from java.lang.Ob ject: toString(). The toString() method produces a string that looks like this:

Socket[addr=www.oreilly.com/198.112.208.11,port=80,localport=50055]

This is useful primarily for debugging. Don't rely on this format; it may change in the future. All parts of this string are accessible directly through other methods (specifically getInetAddress(), getPort(), and getLocalPort()).

## Setting Socket Options

Socket options specify how the native sockets on which the Java Socket class relies send and receive data. Configuring socket options can greatly influence the behavior and performance of network connections.

Java supports nine options for client-side sockets:

1. TCP_NODELAY
2. SO_BINDADDR
3. SO_TIMEOUT
4. SO_LINGER
5. SO_SNDBUF
6. SO_RCVBUF
7. SO_KEEPALIVE
8. OOBINLINE
9. IP_TOS

**1. TCP_NODELAY:** By setting this option on a Socket, you can control the use of Nagle's algorithm. This is useful when you need to ensure that small, interactive messages are sent immediately without waiting for more data to accumulate.

Example:

```
Socket socket = new Socket();
socket.setTcpNoDelay(true);
```

**2. SO_BINDADDR:** Although not currently supported in Java, this option allows you to specify which local IP address to bind the socket to. This is useful when you have multiple network interfaces and want to bind to a specific one.

**3. SO_LINGER:** Controls whether the socket should wait for unsent data to be sent before closing (linger) or close immediately. This can be configured using setSoLinger:

Example:

```
Socket socket = new Socket();
socket.setSoLinger(true, 10); // Wait up to 10 seconds for data to be sent
```

**4. SO_TIMEOUT:** This option sets a timeout for blocking read operations on a Socket. If no data is received within the specified timeout period, a SocketTimeoutException is thrown. This helps in preventing the application from hanging indefinitely during read operations:

Example:

```
Socket socket = new Socket();
socket.setSoTimeout(5000); // 5 seconds timeout
```

**5. SO_SNDBUF:** This option sets the size of the send buffer, which can impact how much data the socket can handle before blocking. Configuring a larger buffer can improve performance when sending large amounts of data.

Example:

```
Socket socket = new Socket();
socket.setSendBufferSize(8192); // 8 KB buffer size
```

**6. SO_RCVBUF:** Similarly, this option sets the size of the receive buffer. Increasing the buffer size can improve performance for receiving large amounts of data.

Example:

```
Socket socket = new Socket();
socket.setReceiveBufferSize(8192); // 8 KB buffer size
```

**7. SO_KEEPALIVE:** Enabling this option allows the socket to periodically send keep-alive packets to ensure that the connection is still active. This helps detect dropped connections.

Example:

```
Socket socket = new Socket();
socket.setKeepAlive(true);
```

**8. OOBINLINE:** This option determines if out-of-band (urgent) data should be read inline with regular data or as a separate stream. This can be useful for handling urgent data differently from normal data:

Example:

```
Socket socket = new Socket();
socket.setOOBInline(true);
```

**9. IP_TOS:** Sets the Type of Service (ToS) field in the IP header to specify desired quality of service for the socket connection, such as prioritizing traffic:

Example:

```
Socket socket = new Socket();
socket.setOption(StandardSocketOptions.IP_TOS, 0x10); // Set ToS value
```

## Sockets in GUI Applications:

Java can handle network communication within GUI applications, though integrating networking with user interfaces (like Swing) presents challenges due to asynchronous operations and user interactions.

### Whois:

Whois is a directory service protocol used for retrieving information about Internet hosts and domains, such as contact details. The Whois protocol is designed to query directory information about Internet hosts and domains. It provides details such as contact information for domain administrators.

**Process:**

- **Connection:** The client establishes a TCP connection to port 43 on the Whois server.
- **Query:** The client sends a search string, which could be a domain name or personal name, terminated by a carriage return/linefeed sequence (\r\n).
- **Response:** The server responds with human-readable information about the queried domain or host and then closes the connection.
- **Display:** The client receives and displays this information to the user.

**Network Client Library:**

Network protocols should be encapsulated in separate libraries from GUI code. Networking code should be separated into distinct libraries from GUI code. This separation ensures clarity, modularity, and maintainability of the application.

**Whois Class Example:**

- **Fields:** `host` (an `InetAddress` object) and `port` (an `int`) define the server connection details.
- **Constructors:** Multiple constructors initialize these fields with different combinations of arguments.

- **Method:** `lookUpNames()` method performs the Whois query and processes the server's response. It uses enums for specifying record types and databases to ensure type safety and prevent errors.

# Unit 7: Socket for Servers

For servers that accept connections, Java provides a ServerSocket class that represents server sockets. In essence, a server socket's job is to sit by the phone and wait for in- coming calls. More technically, a server socket runs on the server and listens for in- coming TCP connections.

## #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 this:

1. **Create ServerSocket:** Instantiate a ServerSocket on a specified port.
2. **Listen for Connections:** Use the accept() method to wait for client connections. This method returns a Socket object when a connection is made.
3. **Communicate:** Obtain input and output streams from the Socket object to exchange data with the client.
4. **Interaction:** Follow a protocol for data exchange until the connection needs to be closed.
5. **Close Connection:** Either the server or the client closes the connection.
6. **Repeat:** The server goes back to listening for new connections.

Example: For a daytime server on port 13, the server sends the current time to the client in a human-readable format and then closes the connection.

**Serving Binary Data:**

To send binary data in Java, you use an OutputStream to write a byte array, as opposed to using a Writer for text data. For example, a time server following RFC 868 sends a 4-byte, big-endian, unsigned integer representing the number of seconds since January 1, 1900, GMT. Since Java's Date class works with milliseconds since January 1, 1970, GMT, you need to convert the current time accordingly.

Example: A time server

```
import java.io.*;
import java.net.*;
import java.util.Date;
public class TimeServer {
 public final static int PORT = 37;
 public static void main(String[] args) {

long differenceBetweenEpochs = 2208988800 L;
 try (ServerSocket server = new ServerSocket(PORT)) {
 while (true) {
 try (Socket connection = server.accept()) {
 OutputStream out = connection.getOutputStream();
 Date now = new Date();
 long msSince1970 = now.getTime();
 long secondsSince1970 = msSince1970/1000;
 long secondsSince1900 = secondsSince1970
 + differenceBetweenEpochs;
 byte[] time = new byte[4];
 time[0]
 = (byte) ((secondsSince1900 & 0x00000000FF000000L) >> 24);
 time[1]
 = (byte) ((secondsSince1900 & 0x0000000000FF0000L) >> 16);
 time[2]
 = (byte) ((secondsSince1900 & 0x000000000000FF00L) >> 8);
 time[3] = (byte) (secondsSince1900 & 0x00000000000000FFL);
 out.write(time);
 out.flush();
 } catch (IOException ex) {
 System.err.println(ex.getMessage());
```

```
 }

 }
 } catch (IOException ex) {
 System.err.println(ex);
 }
 }
}
```

### Multithreaded Servers:

Uses threads to handle each client connection simultaneously. This allows the server to continue accepting and processing new connections while existing ones are handled in separate threads.

Benefits of Multithreading:

- **Concurrency:** Threads enable the server to manage multiple connections at the same time without waiting for one to complete before starting another.
- **Efficiency:** Threads are more lightweight compared to creating separate processes for each connection. This reduces the overhead and resource usage associated with managing multiple connections.
- **Scalability:** Allows the server to handle a larger number of simultaneous connections. Unlike older Unix servers, which create a new process for each connection (leading to significant resource consumption and limited scalability), a multithreaded approach can handle more connections efficiently.

### Writing to and Reading from Servers with Sockets:

When building servers that interact with clients, you often need to both send and receive data. Here's a breakdown of the essentials for implementing this functionality:

**1. Socket Basics**

- **InputStream and OutputStream**: For a socket connection, you use InputStream to read data from the client and OutputStream to send data to the client.

**2. Handling Connections**

- **Accepting Connections**: You accept a connection using ServerSocket.accept().
- **Getting Streams**: Once a connection is established, you obtain both InputStream and OutputStream from the Socket object:

```
InputStream in = connection.getInputStream();
OutputStream out = connection.getOutputStream();
```

**3. Reading and Writing Data**

- **Reading Data**: Use the InputStream to read data sent by the client. You typically read data in chunks (e.g., using a buffer) until the end of the stream is reached or the client closes the connection.
- **Writing Data**: Use the OutputStream to send data back to the client. This can be done in response to what was read, based on the protocol in use.

**Closing Server Sockets:** Properly closing server sockets is an important aspect of network programming to ensure resources are managed correctly and ports are freed up for other applications.

## #Logging in Servers

Logging is crucial for diagnosing issues and understanding the behavior of server applications over time. Here's a guide on what to log, how to handle logs, and best practices to avoid common pitfalls:

## What to Log

1. **Requests**:
    - **Audit Log**: This log should record details about each request received by the server. This might include:
        - **Timestamp**: When the request was received.
        - **Client Information**: IP address, and sometimes user identifiers.
        - **Request Details**: Type of request, data sent, and other relevant details.
        - **Response**: What was sent back to the client.
    - **Operation-Level Logging**: If your server performs multiple operations per request, you may log each operation separately. For example, a dictionary server might log each word lookup as a separate entry.
2. **Server Errors**:
    - **Error Log**: This log should capture unexpected exceptions and errors that occur while the server is running. This includes:
        - **Exception Details**: Stack traces, error messages, and any contextual information.
        - **Timestamp**: When the error occurred.
        - **Code Location**: File name, method name, or line number where the error occurred.

## How to Log

Logging is essential for monitoring and debugging applications. Java's built-in logging framework, java.util.logging, provides a simple yet powerful way to handle logging. Here's how to use it effectively:

### Creating and Configuring a Logger

1. **Initialize a Logger**:
    - **Single Logger per Class**: It's common practice to create one logger per class. This makes it easy to identify where log messages are coming from.

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

2. **Logger Configuration**: Loggers can be configured to output to various destinations (files, databases, etc.) via external configuration files. By default, logs are often written to a file or console.
3. **Log Levels**: Java defines seven logging levels. Use these to indicate the severity of log messages.
    - Level.SEVERE (highest value)
    - Level.WARNING
    - Level.INFO
    - Level.CONFIG
    - Level.FINE
    - Level.FINER
    - Level.FINEST (lowest value)

- **Usage**: Use INFO for general logs, WARNING or SEVERE for errors. Lower levels (FINE, FINER, FINEST) are usually used for debugging and should be disabled in production.

## Constructing Server Sockets

In Java, ServerSocket is used to create server-side sockets that listen for incoming client connections. There are four primary constructors for ServerSocket, allowing you to configure various aspects of the server socket:

1. **Default Constructor:**
   public ServerSocket() throws IOException

   Creates an unbound ServerSocket. Typically, you would use this constructor when you plan to bind the socket later using the bind method.

2. **Port Only:** Creates a ServerSocket bound to the specified port number. This will use default values for the queue length and bind to all available network interfaces.

   Example: ServerSocket httpd = new ServerSocket(80);

3. **Port and Queue Length:** Creates a ServerSocket bound to the specified port number with a specified maximum length for the connection request queue.

   Example: ServerSocket httpd = new ServerSocket(80, 50);

4. **Port, Queue Length, and Bind Address:** Creates a ServerSocket bound to the specified port number and network interface (or IP address) with a specified maximum length for the connection request queue.

   Example:
   ```
   InetAddress localAddress = InetAddress.getByName("192.168.1.100");
   ServerSocket httpd = new ServerSocket(80, 50, localAddress);
   ```

**Example:** 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 {
 ServerSocket server = new ServerSocket(port);
 } catch (IOException ex) {
 System.out.println("There is a server on port " + port + ".");
 }
 }
 }
}
```

## Constructing Without Binding

The no-arguments constructor for ServerSocket creates an instance without binding it to a specific port. This can be useful if you want to configure socket options before binding or if you need to select a port dynamically at runtime.

Example:

```
ServerSocket serverSocket = new ServerSocket();
serverSocket.bind(null);
int port = serverSocket.getLocalPort();
```

```
System.out.println("Bound to port: " + port);
```

## Getting Information About a Server Socket

Once a ServerSocket has been bound to a port and network interface, you might need to retrieve details about its configuration. This can be useful for scenarios where the port was chosen dynamically or when working with multiple interfaces.

1. **getInetAddress()**
    - **Purpose**: Retrieves the IP address of the network interface the ServerSocket is bound to.
    - **Returns**: An InetAddress object representing the local IP address, or null if the socket has not been bound.

      Example:

      ```
      ServerSocket serverSocket = new ServerSocket(0);
      InetAddress inetAddress = serverSocket.getInetAddress();
      System.out.println("Server is bound to IP address: " + inetAddress);
      ```

2. **getLocalPort()**
    - **Purpose:** Retrieves the port number on which the ServerSocket is listening.
    - **Returns:** An integer representing the port number, or -1 if the socket is not bound.

      Example:

      ```
      ServerSocket serverSocket = new ServerSocket(0);
      int port = serverSocket.getLocalPort();
      System.out.println("Server is listening on port: " + port);
      ```

Example: . 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);
 }
 }
}
```

## Socket Options

Socket options specify how the native sockets on which the ServerSocket class relies send and receive data. 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 before throwing a java.io.InterruptedIOException. If SO_TIMEOUT is 0, accept() will never time out. The default is to never timeout.

```
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. As you probably expect, there are two methods to get and set this option:

```
public boolean getReuseAddress() throws SocketException
public void setReuseAddress(boolean on) throws SocketException
```

**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
```

### Class of Service

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 connections and even held up for several hours without major harm.

Four general traffic classes are defined for TCP:

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

The setPerformancePreferences() method expresses the relative preferences given to connection time, latency, and bandwidth for sockets accepted on this server:

```
public void setPerformancePreferences(int connectionTime, int latency,
    int bandwidth)
```

## HTTP Servers

Various types of HTTP servers that can be built with server sockets in Java, each serving different purposes and complexity levels:

1. **Single-File HTTP Server:** This server responds to all requests with the same file. It is designed to be simple, with configurable options for the filename, port, and content encoding. If not specified, it defaults to port 80 and ASCII encoding.
2. **Redirector**: This server redirects incoming requests to a specified URL. It uses a 302 FOUND response code to perform the redirection. Each request is handled in a new thread, which is straightforward but less efficient than using a thread pool.
3. **Full-Fledged HTTP Server (JHTTP)**: A more advanced server that serves an entire document tree, including various file types like images and HTML. It handles GET requests and uses a pool of connections for processing, allowing it to manage larger and more complex requests efficiently.

# Unit 8: Secure sockets

Secure sockets provide a layer of security for data transmitted over a network by encrypting the data and ensuring its integrity. Secure sockets in Java are implemented using the Java Secure Socket Extension (JSSE), which provides support for SSL and TLS protocols. The key components involved are SSLSocket, SSLSocketFactory, SSLServerSocket, and SSLServerSocketFactory.

## Secure Communications

Secure communications in Java involve using SSL/TLS protocols to encrypt data transmitted between clients and servers. This ensures that data remains confidential and is protected from unauthorized access or tampering.

## Creating Secure Client Sockets

1. **Create SSLSocketFactory:** Use SSLSocketFactory.getDefault() or create a custom factory using SSLContext to configure the socket.

   **Example:**

   ```
   SSLContext sslContext = SSLContext.getInstance("TLS");
   sslContext.init(null, null, new SecureRandom());
   SSLSocketFactory socketFactory = sslContext.getSocketFactory();
   ```

2. **Create SSLSocket:** Obtain an SSLSocket from the factory and configure it as needed.

   Example:

   ```
   SSLSocket socket = (SSLSocket) socketFactory.createSocket("server.com", 443);
   ```

3. **Configure SSL Parameters:** Set SSL parameters like enabled protocols and cipher suites.

   Example:

   ```
   socket.setEnabledProtocols(new String[] { "TLSv1.3" });
   ```

```
socket.setEnabledCipherSuites(new String[] { "TLS_AES_128_GCM_SHA256" });
```

4. **Establish Connection:** Connect to the server and perform secure data exchange.

## Event Handlers

Event handlers manage the lifecycle of secure connections. In Java, you handle these events through methods and callbacks:

- **Connection Established:** Use SSLSocket's methods to perform actions when a connection is established.
- **Data Received:** Read encrypted data using input streams.
- **Connection Closed:** Clean up resources and close the socket when the connection ends.

## Session Management

- **Session Creation:**Secure sessions are created automatically when a connection is established.
- **Session Resumption:**Implement session resumption to avoid re-establishing sessions from scratch. This involves using session tickets or session identifiers.
- **Session Termination:**Properly terminate sessions to ensure that no sensitive information remains.

## Client Mode

In client mode, the client is responsible for initiating the secure connection to the server. Key tasks include:

- **Client Authentication:** Verifying the server's identity using certificates.

- **Session Negotiation:** Negotiating the parameters of the secure session with the server.

## Creating Secure Server Sockets

When creating a secure server socket:

1. **Configure SSL Parameters:** Set up SSLServerSocketFactory and configure SSL parameters like key and trust stores.
2. **Start Listening for Connections:** Open a secure server socket and listen for incoming client connections.
3. **Accept Connections:** Accept incoming secure connections and handle them appropriately.

## Configuring SSLServerSockets

1. **Choosing Cipher Suites:** Select appropriate cipher suites to ensure strong encryption. Cipher suites define the encryption algorithms used in secure communication. The SSLServerSocket class has the same three methods for determining which cipher suites are supported and enabled as SSLSocket does:

```
public abstract String[] getSupportedCipherSuites()
public abstract String[] getEnabledCipherSuites()
public abstract void     setEnabledCipherSuites(String[] suites)
```

2. **Session Management:** Manage SSL sessions, including session creation, resumption, and termination to optimize performance and security. Both client and server must agree to establish a session. The server side uses the setE nableSessionCreation() method to specify whether this will be allowed and the ge tEnableSessionCreation() method to determine whether this is currently allowed:

```
public abstract void setEnableSessionCreation(boolean allowSessions)
public abstract boolean getEnableSessionCreation()
```

3. **Client Mode:** The SSLServerSocket class has two methods for determining and specifying whether client sockets are required to authenticate themselves to the server. By passing true to the setNeedClientAuth() method, you specify that only connections in which the client is able to authenticate itself will be accepted.

```
public abstract void setNeedClientAuth(boolean flag)
public abstract boolean getNeedClientAuth()
```

# Unit 9: Nonblocking I/O

Nonblocking I/O allows programs to manage multiple I/O operations efficiently by avoiding blocking operations. It is essential for high-performance applications and systems requiring scalable and responsive behavior. In Java, nonblocking I/O is implemented using the java.nio package, including Selector, SocketChannel, and AsynchronousSocketChannel.

| Blocking I/O | Non-Blocking I/O |
|---|---|
| The thread is blocked until the I/O operation completes. | The thread can continue executing other tasks; I/O operations return immediately if they cannot be completed. |
| Blocking: The thread waits for the operation to complete before continuing. | Non-Blocking: The thread checks the status of the operation and proceeds without waiting for completion. |
| Less efficient; threads can be idle while waiting for I/O operations to complete. | More efficient; avoids blocking threads, allowing better resource utilization. |
| Requires multiple threads or processes to handle multiple I/O operations simultaneously. | Can handle many I/O operations with fewer threads, often using event-driven or asynchronous approaches. |
| Simpler to implement; straightforward programming model. | More complex; requires careful handling of events or asynchronous operations. |

| | |
|---|---|
| Suitable for simple applications or scenarios with low concurrency. | Ideal for high-performance, scalable applications with many concurrent I/O operations. |

## An Example of client and Server

```
import java.io.IOException;
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
public class ChargenClient {
 public static int DEFAULT_PORT = 19;
 public static void main(String[] args) {
 if (args.length == 0) {
 System.out.println("Usage: java ChargenClient host [port]");
 return;
 }
 int port;
 try {
 port = Integer.parseInt(args[1]);
 } catch (RuntimeException ex) {
 port = DEFAULT_PORT;
 }
 try {
 SocketAddress address = new InetSocketAddress(args[0], port);
 SocketChannel client = SocketChannel.open(address);
 ByteBuffer buffer = ByteBuffer.allocate(74);

WritableByteChannel out = Channels.newChannel(System.out);
 while (client.read(buffer) != -1) {
 buffer.flip();
 out.write(buffer);
 buffer.clear();
 }
```

```
 } catch (IOException ex) {
 ex.printStackTrace();
 }
 }
}
```

## Buffers

Buffers are crucial in I/O operations as they temporarily hold data while it's being transferred between different components or systems. Buffers help manage data flow, improve performance, and handle discrepancies in the speed of data production and consumption.

Buffer tracks four key pieces of information. All buffers have the same methods to set and get these values,

1. **position:** The next location in the buffer that will be read from or written to. This starts counting at 0 and has a maximum value equal to the size of the buffer. It can be set or gotten with these two methods:

```
public final int    position()
public final Buffer position(int newPosition)
```

2. **capacity:** The maximum number of elements the buffer can hold. This is set when the buffer is created and cannot be changed thereafter. It can be read with this method:

```
public final int capacity()
```

3. **limit:** The end of accessible data in the buffer. You cannot write or read at or past this point without changing the limit, even if the buffer has more capacity. It is set and gotten with these two methods:

```
public final int    limit()
public final Buffer limit(int newLimit)
```

4. **mark:** A client-specified index in the buffer. It is set at the current position by invoking the mark() method. The current position is set to the marked position by invoking reset():

```
public final Buffer mark()
public final Buffer reset()
```

### Creating Buffers:

Ways to create buffers,

1. **Allocation:** The basic allocate() method simply returns a new, empty buffer with a specified fixed capacity. For example, these lines create byte and int buffers, each with a size of 100:

```
ByteBuffer buffer1 = ByteBuffer.allocate(100);
IntBuffer  buffer2 = IntBuffer.allocate(100);
```

2. **Direct allocation:** The allocateDirect() method is specific to ByteBuffer and creates a direct buffer. This buffer may not have a backing array and might use native memory for storage. Direct buffers can improve performance for I/O operations since they can be accessed directly by native I/O operations.

```
ByteBuffer buffer = ByteBuffer.allocateDirect(100);
```

3. **Wrapping:** If you already have an array of data that you want to output, you'll normally wrap a buffer around it, rather than allocating a new buffer and copying its components into the buffer one at a time. For example:

```
byte[] data = "Some data".getBytes("UTF-8");
ByteBuffer buffer1 = ByteBuffer.wrap(data);
char[] text = "Some text".toCharArray();
CharBuffer buffer2 = CharBuffer.wrap(text);
```

### Filling and Draining:

Buffers are designed for sequential access. Recall that each buffer has a current position identified by the position() method that is somewhere between zero and the number of elements in the buffer, inclusive. The buffer's position is incremented by one when an element is read from or written to the buffer. For example, suppose you allocate a CharBuffer with capacity 12, and fill it by putting five characters into it:

```
CharBuffer buffer = CharBuffer.allocate(12);
buffer.put('H');
buffer.put('e');
buffer.put('l');
buffer.put('l');
buffer.put('o');
```

The position of the buffer is now 5. This is called filling the buffer.

Draining a buffer means reading data from the buffer. Once data has been read, the buffer can be used for other operations, such as writing new data.

### Bulk Methods:

Even with buffers, it's often faster to work with blocks of data rather than filling and draining one element at a time. The different buffer classes have bulk methods that fill and drain an array of their element type.

For example, ByteBuffer has put() and get() methods that fill and drain a ByteBuffer from a preexisting byte array or subarray:

```
public ByteBuffer get(byte[] dst, int offset, int length)
public ByteBuffer get(byte[] dst)
public ByteBuffer put(byte[] array, int offset, int length)
public ByteBuffer put(byte[] array)
```

These put methods insert the data from the specified array or subarray, beginning at the current position. The get methods read the data into the argument array or subarray beginning at the current position. Both put and get increment the position by the length

of the array or subarray. The put methods throw a BufferOverflowException if the buffer does not have sufficient space for the array or subarray.

### Data Conversion:

All data in Java ultimately resolves to bytes. Any primitive data type—int, double, float, etc. can be written as bytes. data conversion between primitive types and bytes is facilitated through the ByteBuffer class. This class provides methods to write primitive data types as bytes into a buffer and read bytes from a buffer to form primitive data types.

```
public abstract char       getChar()
public abstract ByteBuffer putChar(char value)
public abstract char       getChar(int index)
public abstract ByteBuffer putChar(int index, char value)
public abstract short      getShort()
public abstract ByteBuffer putShort(short value)
public abstract short      getShort(int index)
public abstract ByteBuffer putShort(int index, short value)
public abstract int        getInt()
public abstract ByteBuffer putInt(int value)
public abstract int        getInt(int index)
public abstract ByteBuffer putInt(int index, int value)
public abstract long       getLong()
public abstract ByteBuffer putLong(long value)
public abstract long       getLong(int index)
public abstract ByteBuffer putLong(int index, long value)
public abstract float      getFloat()
public abstract ByteBuffer putFloat(float value)
public abstract float      getFloat(int index)
public abstract ByteBuffer putFloat(int index, float value)
public abstract double     getDouble()
public abstract ByteBuffer putDouble(double value)
public abstract double     getDouble(int index)
public abstract ByteBuffer putDouble(int index, double value)
```

### View Buffers:

view buffers allow you to interpret the data in a ByteBuffer as a different primitive data type. This is useful when you know that the data in a ByteBuffer is structured in a specific way and you want to access it using a more specialized buffer type (such as IntBuffer or DoubleBuffer). View buffers provide a way to access the underlying byte data in different formats while maintaining the same underlying memory.

View buffers are created with one of these six methods in ByteBuffer:

```
public abstract ShortBuffer  asShortBuffer()
public abstract CharBuffer   asCharBuffer()
public abstract IntBuffer    asIntBuffer()
public abstract LongBuffer   asLongBuffer()
public abstract FloatBuffer  asFloatBuffer()
public abstract DoubleBuffer asDoubleBuffer()
```

### Compacting a buffers:

Compacting a buffer is a process that helps manage and optimize the buffer's usage when some data has been read or written, but the buffer is not yet completely exhausted. This operation is particularly useful when you want to reuse the buffer for further I/O operations without reallocating a new buffer.

Most writable buffers support a compact() method:

```
public abstract ByteBuffer   compact()
public abstract IntBuffer    compact()
public abstract ShortBuffer  compact()
public abstract FloatBuffer  compact()
public abstract CharBuffer   compact()
public abstract DoubleBuffer compact()
```

### Duplicating Buffers:

Duplicating buffers involves creating a new buffer that shares the same underlying data as the original buffer but maintains its own position, limit, and mark. This can be useful when you need to perform multiple operations on the same data without modifying the original buffer's state.

It’s often desirable to make a copy of a buffer to deliver the same information to two or more channels. The duplicate() methods:

```
public abstract ByteBuffer   duplicate()
public abstract IntBuffer    duplicate()
public abstract ShortBuffer  duplicate()
public abstract FloatBuffer  duplicate()
public abstract CharBuffer   duplicate()
public abstract DoubleBuffer duplicate()
```

Duplication is useful when you want to transmit the same data over multiple channels, roughly in parallel.

### Slicing Buffers:

Slicing buffers is a technique that allows you to create a new buffer that represents a subsequence (or "slice") of the original buffer. This is useful when you need to operate on a specific part of the buffer without affecting the rest of it. Like duplicating buffers, the sliced buffer shares the same underlying data as the original buffer, but it has its own position, limit, and mark.

```
public abstract ByteBuffer   slice()
public abstract IntBuffer    slice()
public abstract ShortBuffer  slice()
public abstract FloatBuffer  slice()
public abstract CharBuffer   slice()
public abstract DoubleBuffer slice()
```

This is useful when you have a long buffer of data that is easily divided into multiple parts such as a protocol header followed by the data. You can read out the header, then

slice the buffer and pass the new buffer containing only the data to a separate method or class.

### Marking and Resetting:

Like input streams, buffers can be marked and reset if you want to reread some data. Unlike input streams, this can be done to all buffers, not just some of them. For a change, the relevant methods are declared once in the Buffer superclass and inherited by all the various subclasses:

```
public final Buffer mark()
public final Buffer reset()
```

The reset() method throws an InvalidMarkException, a runtime exception, if the mark is not set. The mark is also unset when the position is set to a point before the mark.

### Object Methods:

The buffer classes all provide the usual equals(), hashCode(), and toString() meth- ods. They also implement Comparable, and therefore provide compareTo() methods. However, buffers are not Serializable or Cloneable.

Two buffers are considered to be equal if:

- They have the same type (e.g., a ByteBuffer is never equal to an IntBuffer but may be equal to another ByteBuffer).
- They have the same number of elements remaining in the buffer.
- The remaining elements at the same relative positions are equal to each other.

Note that equality does not consider the buffers' elements that precede the position, the buffers' capacity, limits, or marks. For example, this code fragment prints true even though the first buffer is twice the size of the second:

```
CharBuffer buffer1 = CharBuffer.wrap("12345678");
CharBuffer buffer2 = CharBuffer.wrap("5678");
buffer1.get();
buffer1.get();
buffer1.get();
buffer1.get();
System.out.println(buffer1.equals(buffer2));
```

## Channels

Channels move blocks of data into and out of buffers to and from various I/O sources such as files, sockets, datagrams, and so forth. The channel class hierarchy is rather convoluted, with multiple interfaces and many optional operations. However, for purposes of network programming there are only three really important channel classes, SocketChannel, ServerSocketChannel, and DatagramChannel; and for the TCP connections we've talked about so far you only need the first two.

### SocketChannel:

The SocketChannel class reads from and writes to TCP sockets. The data must be encoded in ByteBuffer objects for reading and writing. Each SocketChannel is associated with a peer Socket object that can be used for advanced configuration, but this requirement can be ignored for applications where the default options are fine.

- **Connecting:** The SocketChannel class does not have any public constructors. Instead, you create a new SocketChannel object using one of the two static open() methods:

  ```
  public static SocketChannel open(SocketAddress remote) throws IOException
  public static SocketChannel open() throws IOException
  ```

- Reading: To read from a SocketChannel, first create a ByteBuffer the channel can store data in. Then pass it to the read() method:

  ```
  public abstract int read(ByteBuffer dst) throws IOException
  ```

- **Writing:** Socket channels have both read and write methods. In general, they are full duplex. In order to write, simply fill a ByteBuffer, flip it, and pass it to one of the write methods, which drains it while copying the data onto the output—pretty

much the reverse of the reading process.

```
public abstract int write(ByteBuffer src) throws IOException
```

- **Closing:** Just as with regular sockets, you should close a channel when you're done with it to free up the port and any other resources it may be using:

```
public void close() throws IOException
```

### ServerSocketChannel:

The ServerSocketChannel class has one purpose: to accept incoming connections. You cannot read from, write to, or connect a ServerSocketChannel. The only operation it supports is accepting a new incoming connection. The class itself only declares four methods, of which accept() is the most important. ServerSocketChannel also inherits several methods from its superclasses, mostly related to registering with a Selector for notification of incoming connections.

Q. Creating server socket channels.

```
try {
  ServerSocketChannel server = ServerSocketChannel.open();
  ServerSocket socket = serverChannel.socket();
  SocketAddress address = new InetSocketAddress(80);
  socket.bind(address);
} catch (IOException ex) {
  System.err.println("Could not bind to port 80 because " + ex.getMessage());
}
```

### The Channels Class:

Channels is a simple utility class for wrapping channels around traditional I/O-based streams, readers, and writers, and vice versa. It's useful when you want to use the new I/O model in one part of a program for performance, but still interoperate with legacy APIs that expect streams. It has methods that convert from streams to channels and methods that convert from channels to streams, readers, and writers:

```
public static InputStream newInputStream(ReadableByteChannel ch)
public static OutputStream newOutputStream(WritableByteChannel ch)
public static ReadableByteChannel newChannel(InputStream in)
public static WritableByteChannel newChannel(OutputStream out)
public static Reader newReader (ReadableByteChannel channel,
    CharsetDecoder decoder, int minimumBufferCapacity)
public static Reader newReader (ReadableByteChannel ch, String encoding)
public static Writer newWriter (WritableByteChannel ch, String encoding)
```

**Asynchronous Channels:**

Asynchronous channels in Java are part of the java.nio.channels package and are used to perform non-blocking I/O operations. Unlike traditional blocking I/O channels, where operations such as reading or writing block the current thread until they complete, asynchronous channels allow operations to proceed in the background, enabling the program to continue executing other tasks. This is particularly useful in high-performance and scalable applications, such as web servers, where multiple I/O operations might need to be handled simultaneously without stalling the application's flow.

- Asynchronous channels allow you to initiate I/O operations that will complete at some point in the future.
- Operations like reading from or writing to a channel return immediately, and the completion of these operations is handled by a callback or by polling.

## Readiness Selection

Readiness Selection in the context of Java NIO (Non-blocking I/O) refers to the mechanism used to monitor multiple channels (like network sockets or files) to determine which ones are ready for operations such as reading, writing, or connecting without blocking the thread. This concept is implemented through the Selector class in the Java NIO package.

### The Selector class:

The Selector class allows a single thread to efficiently manage multiple channels by determining which channels are ready for a particular I/O operation (e.g., reading, writing, connecting). It works with channels that are in non-blocking mode, which means the I/O operations on these channels do not block the execution of the thread.

When a channel is registered with a selector, a SelectionKey object is created, representing the registration of the channel with the selector. The SelectionKey keeps track of the channel, the selector, and the I/O operations the channel is interested in (e.g., read, write, connect).

The SelectionKey class defines four named bit constants used to select the type of the operation:

- SelectionKey.OP_ACCEPT
- SelectionKey.OP_CONNECT
- SelectionKey.OP_READ
- SelectionKey.OP_WRITE

### The SelectionKey Class:

When a channel is registered with a selector, a SelectionKey object is created to represent this registration. The SelectionKey contains information about the channel, the selector, the interest set (operations the channel is interested in), and the ready set (operations the channel is ready for).

When retrieving a SelectionKey from the set of selected keys, you often first test what that key is ready to do. There are four possibilities:

```
public final boolean isAcceptable()
public final boolean isConnectable()
public final boolean isReadable()
public final boolean isWritable()
```

Once you know what the channel associated with the key is ready to do, retrieve the channel with the channel() method:

```
public abstract SelectableChannel channel()
```

# Unit 3: UDP

UDP (User Datagram Protocol) is one of the core protocols of the Internet Protocol suite, used for transmitting data over a network. Unlike TCP (Transmission Control Protocol), UDP is a connectionless protocol, which means it sends data without establishing a connection between the sender and receiver.

## The UDP Protocol:

The User Datagram Protocol (UDP) is a core communication protocol used in networking for transmitting data between devices. Unlike the Transmission Control Protocol (TCP), UDP is connectionless and does not guarantee the delivery of packets, making it faster but less reliable.

Despite its limitations, it is used in many applications because of its unique advantages. While it lacks the reliability and features of TCP, these same characteristics make it well-suited for specific scenarios where speed and efficiency are more critical than data integrity.

## UDP clients:

A UDP client is an application that sends data to a UDP server using the User Datagram Protocol (UDP). Because UDP is connectionless and does not guarantee delivery, the client simply sends datagrams (packets) to a server without establishing a connection or ensuring that the server has received the data.

## UDP servers:

A UDP server is a network service that listens for incoming datagrams (packets) from clients and processes them using the User Datagram Protocol (UDP). Because UDP does not maintain a connection, UDP servers are generally stateless. They do not need to keep track of client state, making them simpler and more scalable.

## The DatagramPacket Class

The DatagramPacket class is for handling UDP (User Datagram Protocol) communications. It is used to represent data that is either sent or received over a network via UDP. Since UDP is connectionless and sends data in discrete packets (datagrams), the DatagramPacket class encapsulates this data along with information about the destination or source.

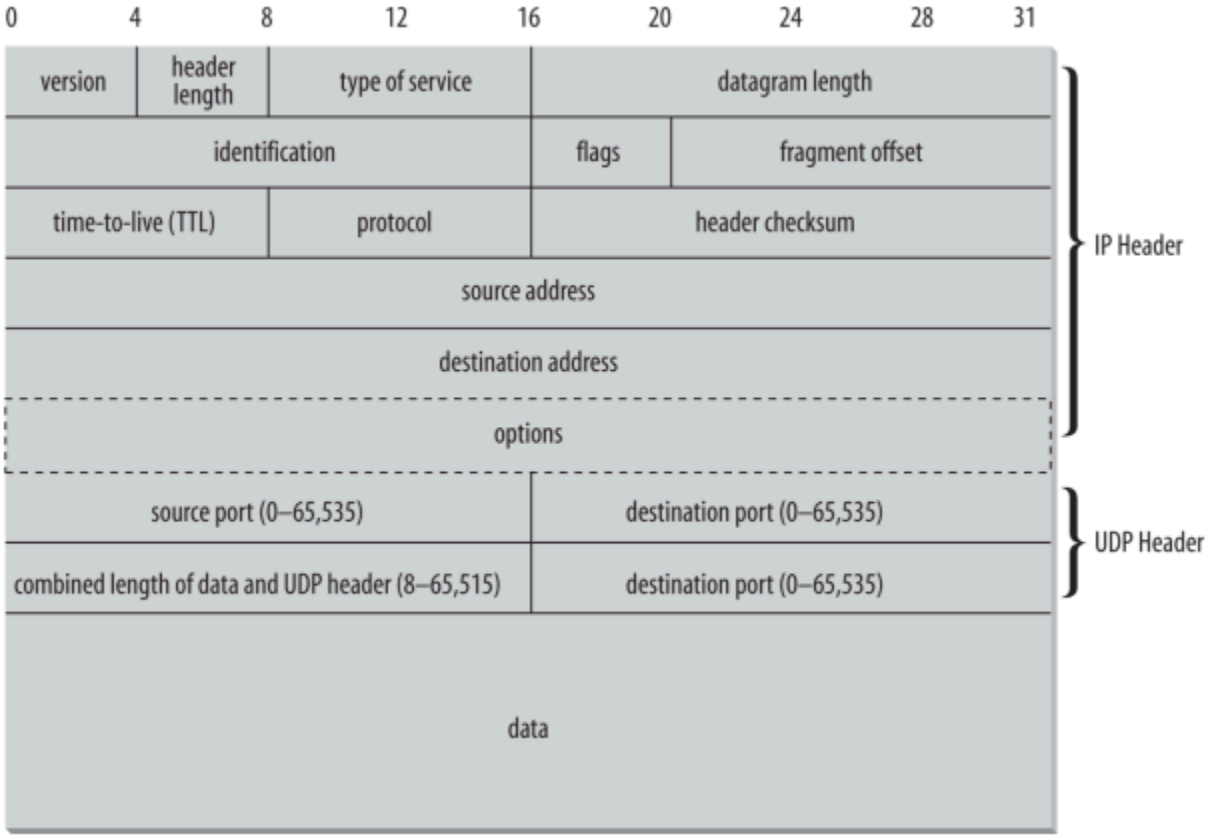



Figure: The structure of a UDP datagram

In Java, a UDP datagram is represented by an instance of the DatagramPacket class:

```
public final class DatagramPacket extends Object
```

This class provides methods to get and set the source or destination address from the IP header, to get and set the source or destination port, to get and set the data, and to get and set the length of the data.

### The Constructors:

The DatagramPacket class provides several constructors to create datagram packets for sending or receiving data. Each constructor is designed to accommodate different use cases for UDP communication.

- **Constructors for receiving datagrams:** These two constructors create new DatagramPacket objects for receiving data from the network:

  ```
  public DatagramPacket(byte[] buffer, int length)
  public DatagramPacket(byte[] buffer, int offset, int length)
  ```

  If the first constructor is used, when a socket receives a datagram, it stores the datagram's data part in buffer beginning at buffer[0] and continuing until the packet is completely stored or until length bytes have been written into the buffer.

- **Constructors for sending datagrams:** These four constructors create new DatagramPacket objects used to send data across the network:

  ```
  public DatagramPacket(byte[] data, int length,
      InetAddress destination, int port)
  public DatagramPacket(byte[] data, int offset, int length,
      InetAddress destination, int port)
  public DatagramPacket(byte[] data, int length,
      SocketAddress destination)
  public DatagramPacket(byte[] data, int offset, int length,
      SocketAddress destination)
  ```

  Each constructor creates a new DatagramPacket to be sent to another host. The packet is filled with length bytes of the data array starting at offset or 0 if offset is not used.

### The get Methods:

DatagramPacket has six methods that retrieve different parts of a datagram: the actual data plus several fields from its header. These methods are mostly used for datagrams received from the network.

1. **getAddress():** Returns the IP address of the sender or receiver.
   Example: `InetAddress address = receivePacket.getAddress();`

2. **getPort():** Returns the port number of the sender or receiver.
   Example: `int port = receivePacket.getPort();`

3. **getOffset():** Returns the offset within the buffer where the data starts. This method is less commonly used but can be helpful if you are working with packets that involve specific portions of a buffer.

4. **getLength():** Returns the length of the data contained in the packet. This is useful for determining how much data was actually received or should be sent.
   Example: `int length = receivePacket.getLength();`

5. **getData():** Returns the byte array that contains the data of the packet.
   Example: `byte[] buffer = receivePacket.getData();`

6. **getSocketAddress():** The getSocketAddress() method returns a SocketAddress object containing the IP address and port of the remote host. As is the case for getInetAddress(), if the datagram was received from the Internet, the address returned is the address of the machine that sent it (the source address).

Example: Construct a DatagramPacket to receive data.

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

public class DatagramExample {

  public static void main(String[] args) {

    String s = "This is a test.";

    try {
      byte[] data = s.getBytes("UTF-8");
      InetAddress ia = InetAddress.getByName("www.ibiblio.org");
      int port = 7;
      DatagramPacket dp
          = new DatagramPacket(data, data.length, ia, port);
      System.out.println("This packet is addressed to "
          + dp.getAddress() + " on port " + dp.getPort());


     System.out.println("There are " + dp.getLength()
         + " bytes of data in the packet");
     System.out.println(
         new String(dp.getData(), dp.getOffset(), dp.getLength(), "UTF-8"));
    } catch (UnknownHostException | UnsupportedEncodingException ex) {
      System.err.println(ex);
    }
  }
}
```

**The setter Methods:**

Most of the time, the six constructors are sufficient for creating datagrams. However, Java also provides several methods for changing the data, remote address, and remote port after the datagram has been created. These methods might be important in a situation where the time to create and garbage collect new DatagramPacket objects is a significant performance hit.

- **public void setData(byte[] data):** The setData() method changes the payload of the UDP datagram. You might use this method if you are sending a large file (where large is defined as “bigger than can comfortably fit in one datagram”) to a remote host.

- **public void setData(byte[] data, int offset, int length):** This overloaded variant of the setData() method provides an alternative approach to sending a large quantity of data. Instead of sending lots of new arrays, you can put all the data in one array and send it a piece at a time.

- **public void setAddress(InetAddress remote):** The setAddress() method changes the address a datagram packet is sent to. This might allow you to send the same datagram to many different recipients.

- **public void setPort(int port):** The setPort() method changes the port a datagram is addressed to. I honestly can’t think of many uses for this method. It could be used in a port scanner application that tried to find open ports running particular UDP-based services such as FSP.

- **public void setAddress(SocketAddress remote):** The setSocketAddress() method changes the address and port a datagram packet is sent to. You can use this when replying.

- **public void setLength(int length):** The setLength() method changes the number of bytes of data in the internal buffer that are considered to be part of the datagram’s data as opposed to merely unfilled space. This method is useful when receiving datagrams, as we’ll explore later in this chapter. When a datagram is received, its length is set to the length of the incoming data.

## The DatagramSocket Class:

To send or receive a DatagramPacket, you must open a datagram socket. In Java, a datagram socket is created and accessed through the DatagramSocket class:

```
public class DatagramSocket extends Object
```

All datagram sockets bind to a local port, on which they listen for incoming data and which they place in the header of outgoing datagrams. If you're writing a client, you don't care what the local port is, so you call a constructor that lets the system assign an unused port (an anonymous port).

### The Constructors:

The DatagramSocket class is used for creating and managing UDP (User Datagram Protocol) sockets. It provides constructors that allow you to open a datagram socket for communication. Each constructor has a specific purpose, depending on whether you need to bind the socket to a specific port or network interface or use an anonymous port.

Here's a summary of the constructors:

1. **DatagramSocket():** This constructor creates a DatagramSocket that binds to any available port on the local machine. The system automatically assigns an available port number. It is used when you don't need a specific port and want the system to choose one for you.
   Example: `DatagramSocket socket = new DatagramSocket();`

2. **DatagramSocket(int port):** This constructor creates a DatagramSocket that binds to a specified port on the local machine. The socket listens for incoming datagrams on the given port across all available network interfaces. It is used when you need to use a specific port, often for server applications.
   Example: `DatagramSocket socket = new DatagramSocket(9876);`

3. DatagramSocket(int port, InetAddress laddr): This constructor creates a DatagramSocket that binds to a specified port on a specified local network interface. It allows you to specify both the port and the network interface (e.g., a specific IP address) to which the socket should be bound. When you need to bind the socket to a specific IP address on a multi-homed machine (a machine with multiple network interfaces).
   Example:

```
InetAddress localAddress = InetAddress.getByName("192.168.1.100");
DatagramSocket socket = new DatagramSocket(9876, localAddress);
```

4. **DatagramSocket(SocketAddress bindaddr):** This constructor allows you to bind the socket to a specific SocketAddress, which can include both an IP address and a port. It provides more flexibility in specifying the binding address. When you need precise control over both the IP address and the port to which the socket is bound.

   Example:
   ```
   SocketAddress address = new InetSocketAddress("192.168.1.100", 9876);
   DatagramSocket socket = new DatagramSocket(address);
   ```

## Sending Datagrams:

To send data over a network using UDP (User Datagram Protocol), you create a DatagramPacket containing the data and use a DatagramSocket to send it.

**1) Create a DatagramSocket:**

- This socket is the endpoint for sending data.
- You can create it using an anonymous port or bind it to a specific port.

```
DatagramSocket socket = new DatagramSocket();
```

**2) Prepare the Data to Send:**

- The data you want to send is typically stored in a byte[] array.
  ```
  byte[] data = "Hello, World!".getBytes();
  ```

**3) Create a DatagramPacket:**

- The DatagramPacket contains the data to be sent, the length of the data, the destination IP address, and the destination port number.
  ```
  InetAddress address = InetAddress.getByName("192.168.1.100");
  ```

**4) Send the Datagram:**

- Use the send() method of the DatagramSocket to send the packet.
  socket.send(packet);

**5) Close the Socket:**

- After sending the data, you should close the socket.
  socket.close();

## Receiving Datagrams:

To receive data over a network using UDP, you create a DatagramPacket to hold the incoming data and use a DatagramSocket to receive it.

**1) Create a DatagramSocket:**

- The socket is created and bound to a specific port where it listens for incoming datagrams.
  DatagramSocket socket = new DatagramSocket(9876);

**2) Create a DatagramPacket:**

- The DatagramPacket should be large enough to hold the incoming data.
- The constructor typically only specifies the buffer and its size since the sender's address and port are unknown initially.
  byte[] buffer = new byte[1024];
  DatagramPacket packet = new DatagramPacket(buffer, buffer.length);

**3) Receive the Datagram:**

- The receive() method of the DatagramSocket blocks until a datagram is received, at which point it fills the DatagramPacket.socket.receive(packet);

**4) Process the Data:**

- You can access the data from the DatagramPacket and process it as needed.
  String receivedData = new String(packet.getData(), 0, packet.getLength());
  System.out.println("Received: " + receivedData);

**5) Close the Socket:**

- After receiving the data, you should close the socket.
  socket.close();

## Managing Connections

DatagramSocket can be "connected" in a loose sense to a specific address and port, but this does not create a reliable connection like TCP. Instead, it restricts the socket to send and receive datagrams only to/from the specified address and port. Here's how the relevant methods work:

1. **connect(InetAddress host, int port) Method:** Restricts the DatagramSocket so it only communicates with a specified remote host and port. After calling connect(), the socket will only send data to the specified host and port. The socket will also only receive data from the specified host and port. Datagrams from other sources will be silently discarded.

   DatagramSocket socket = new DatagramSocket();
   InetAddress address = InetAddress.getByName("example.com");
   socket.connect(address, 1234);

2. **disconnect() Method:** Removes any previous restrictions set by connect(), allowing the DatagramSocket to send and receive data from any host and port again. Useful when the application logic requires switching communication to different hosts or when the socket should be able to receive datagrams from any host.
   socket.disconnect();

3. **getPort() Method:** Retrieves the port number to which the socket is “connected.” Returns the port number if the socket is connected. Otherwise, returns -1.

int port = socket.getPort();

4. **getInetAddress() Method:** Returns the InetAddress of the host to which the socket is “connected.” Return Value: Returns the InetAddress if the socket is connected. Otherwise, returns null.
   InetAddress address = socket.getInetAddress();

5. **getRemoteSocketAddress() Method:** similar to getInetAddress() but returns an InetSocketAddress that includes both the IP address and port of the remote endpoint. Returns the InetSocketAddress if the socket is connected. Otherwise, it returns null.

## Socket Options

Java supports six socket options for UDP:

- SO_TIMEOUT
- SO_RCVBUF
- SO_SNDBUF
- SO_REUSEADDR
- SO_BROADCAST
- IP_TOS

### SO_REUSEADDR

This option allows a socket to bind to an address that is already in use. This is useful for applications that need to restart and bind to the same port (like a server application) without waiting for the operating system to clear the previous socket. It also allows multiple sockets to bind to the same address and port for multicast or broadcast operations.

Example: serverSocket.setReuseAddress(true);

**SO_BROADCAST**

This option allows a datagram socket to send and receive broadcast packets. It is used when the application needs to send data to all devices in a local network segment. This is generally used in UDP communication, such as broadcasting discovery packets to find devices or services on a network.

Example: datagramSocket.setBroadcast(true);

Note:SO_TIMEOUT, SO_RCVBUF, SO_SNDBUF, IP_TOS these options are already explained in previous units.

# Unit 11: IP Multicast

IP multicast is a method used in networking to efficiently send data to multiple receivers at the same time using a single transmission. Instead of sending separate copies of the data to each recipient (as with unicast), multicast allows the sender to transmit a single packet that is then distributed to multiple receivers who have expressed interest in receiving that data.

## IP Multicast Features:

**1) Multicast Address**:

- IP multicast uses a special range of IP addresses for multicast groups. These addresses are in the range of `224.0.0.0` to `239.255.255.255` for IPv4. Each multicast group is identified by a unique multicast IP address.
- For IPv6, multicast addresses are in the `ff00::/8` range.

**2) Multicast Group**:

- A multicast group is a set of hosts that want to receive the same data stream. Hosts can join or leave a multicast group dynamically.
- The sender does not need to know who the receivers are; it simply sends data to a multicast address, and all members of that group receive the data.

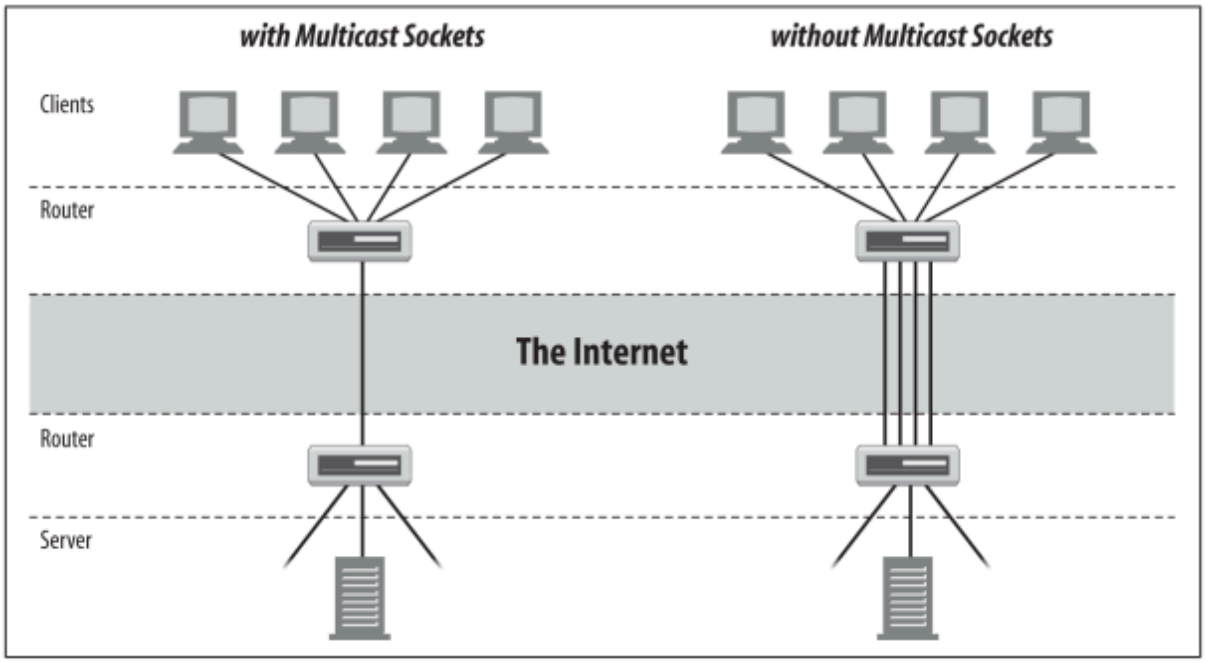
**3) Multicast Client:** A multicast client is a device or application that wants to receive data sent to a multicast group. To receive the multicast data, a client must join a specific multicast group.

**4) Multicast Server:** A multicast server is a device or application that sends data to a multicast group. The server does not send data to each client individually. Instead, it sends a single stream of data to a multicast group address, and network infrastructure ensures the data is delivered to all clients in that group.

**5) Multicast Router:** Multicast routers are responsible for managing multicast group membership and distributing multicast data packets across the network. Routers ensure

that multicast data is only forwarded to the network segments where there are interested clients.

**6) Multicast Routing**: Routers use multicast routing protocols (like PIM) to manage the distribution of multicast data efficiently. The routers maintain a multicast routing table that helps them forward multicast packets only to networks where there are interested receivers.




## Working with multicast sockets:

### 1) The MulticastSocket Constructor

The MulticastSocket class in Java is a subclass of DatagramSocket and provides the ability to join multicast groups and receive multicast data. It has several constructors, but the most commonly used ones are:

a) **Default Constructor**:
MulticastSocket multicastSocket = new MulticastSocket();
This constructor creates a multicast socket and binds it to an automatically

assigned port.

b) **Constructor with Port**:
MulticastSocket multicastSocket = new MulticastSocket(int port);
This constructor creates a multicast socket and binds it to a specific port number. This is useful when you want to listen on a specific port for multicast traffic.

Example:
MulticastSocket multicastSocket = new MulticastSocket(4446);

This creates a multicast socket bound to port 4446.

## 2. Communicating with a Multicast Group

Communicating with a multicast group involves both sending data to and receiving data from the group.

**a) Joining a Multicast Group:** To receive data from a multicast group, a MulticastSocket must first join the group.

**b) Joining a Group**:
InetAddress group = InetAddress.getByName("230.0.0.1");
multicastSocket.joinGroup(group);

Here, the multicast socket joins the group identified by the IP address 230.0.0.1. This IP is within the range reserved for multicast traffic (224.0.0.0 to 239.255.255.255 for IPv4).

**c) Leaving a Group**:
multicastSocket.leaveGroup(group);

This method is used to leave a multicast group when you no longer want to receive data from it.

**d) Sending Data to a Multicast Group:** To send data to a multicast group, you create a DatagramPacket with the data and the multicast group address and then use the send() method of the MulticastSocket.

**e) Receiving Data from a Multicast Group:** To receive data, you must use the receive() method of the MulticastSocket. This method blocks until a packet is received.

# Unit 12: RMI (Remote Method Invocation)

The RMI (Remote Method Invocation) is an API that provides a mechanism to create distributed applications in java. The RMI allows an object to invoke methods on an object running in another JVM.

**Implement RMI service Interface:**

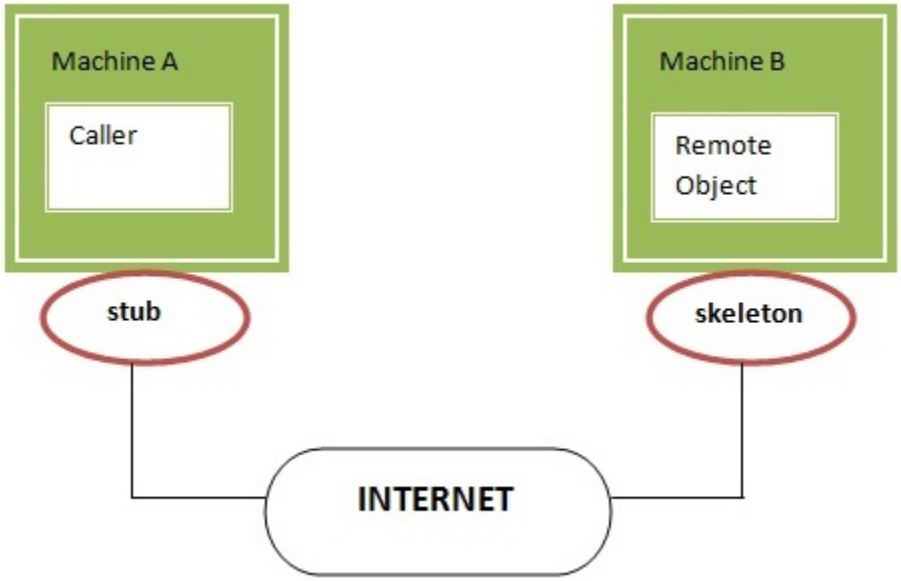



Figure: RMI service Interface

1. Defining a remote interface
2. Implementing the remote interface
3. Creating Stub and Skeleton objects from the implementation class using rmic (RMI compiler)
4. Start the rmi registry.
5. Create and execute the server application program
6. Create and execute the client application program.

Step 1: Defining the remote interface The first thing to do is to create an interface that will provide the description of the methods that can be invoked by remote clients. This interface should extend the Remote interface and the method prototype within the interface should throw the RemoteException.

Step 2: Implementing the remote interface The next step is to implement the remote interface. To implement the remote interface, the class should extend to the UnicastRemoteObject class of java.rmi package. Also, a default constructor needs to be created to throw the java.rmi.RemoteException from its parent constructor in class.

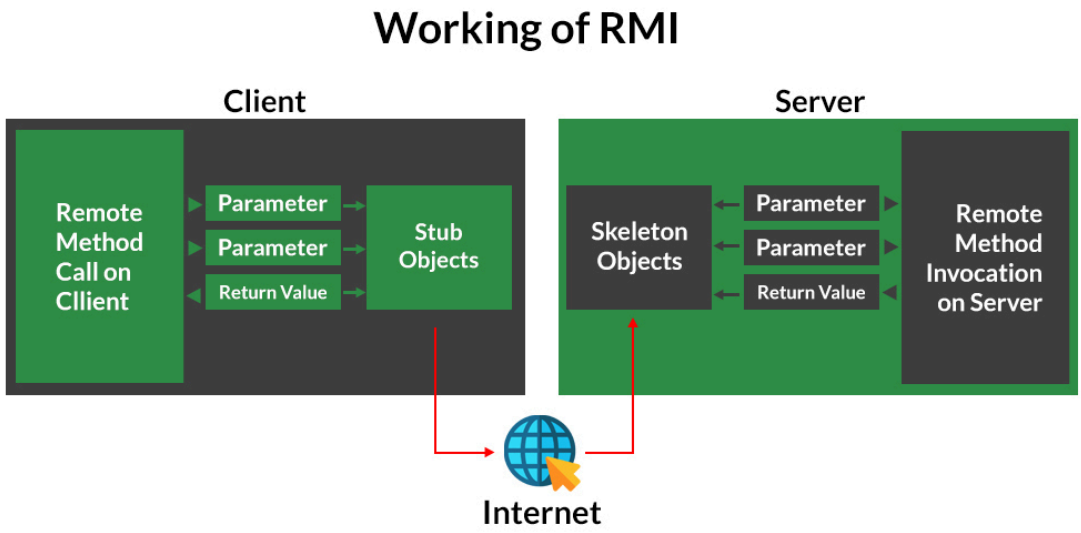
Step 3: Creating Stub and Skeleton objects from the implementation class using rmic The rmic tool is used to invoke the rmi compiler that creates the Stub and Skeleton objects. Its prototype is rmic classname. For the above program the following command need to be executed at the command prompt rmic SearchQuery.

Step 4: Start the rmi registry Start the registry service by issuing the following command at the command prompt start rmi registry

Step 5: Create and execute the server application program The next step is to create the server application program and execute it on a separate command prompt.

**creating and RMI client and server.**

Creating an RMI (Remote Method Invocation) client and server in Java involves several steps, including defining a remote interface, implementing the server, creating a client, and managing the RMI registry. RMI allows Java objects to invoke methods on an object running in another JVM, potentially on a different machine.



7. Defining a remote interface
8. Implementing the remote interface
9. Creating Stub and Skeleton objects from the implementation class using rmic (RMI compiler)
10. Start the rmi registry.
11. Create and execute the server application program
12. Create and execute the client application program.

Step 1: Defining the remote interface The first thing to do is to create an interface that will provide the description of the methods that can be invoked by remote clients. This interface should extend the Remote interface and the method prototype within the interface should throw the RemoteException.

Step 2: Implementing the remote interface The next step is to implement the remote interface. To implement the remote interface, the class should extend to the UnicastRemoteObject class of java.rmi package. Also, a default constructor needs to be created to throw the java.rmi.RemoteException from its parent constructor in class.

Step 3: Creating Stub and Skeleton objects from the implementation class using rmic The rmic tool is used to invoke the rmi compiler that creates the Stub and Skeleton objects. Its prototype is rmic classname. For the above program the following command need to be executed at the command prompt rmic SearchQuery.

Step 4: Start the rmi registry Start the registry service by issuing the following command at the command prompt start rmi registry

Step 5: Create and execute the server application program The next step is to create the server application program and execute it on a separate command prompt.