Networking in Java

Networking in Java primarily deals with creating distributed applications that communicate over a network. Applications are broadly categorized into:

1. **Standalone Applications:** Run on a single machine without requiring network communication or client-server architecture.
2. **Distributed Applications:** Logic is spread across multiple machines, typically using a client-server model, to perform tasks collaboratively.

Distributed Applications can be further classified:

Type	**Description**	**Client**	**Server**	**Technologies Used**
Web-Based	Application logic resides mainly on the server.	Browser	Single Server hosting application logic	Servlets, JSPs, Frameworks (Spring, etc.)
Remote-Based	Application logic distributed between client and server.	Java Program	Server hosting part of application logic	Socket Programming, RMI, CORBA, EJBs, Web Services

Export to Sheets

Socket Programming

Socket programming is a mechanism to build remote-based distributed applications by establishing communication channels (**Sockets**) between client and server processes over a network. A **Socket** is an endpoint for communication, allowing data to be sent and received between machines.

Java provides the `java.net` package for socket programming.

Steps to Build a Distributed Application using Socket Programming

At the Client Side:

1. **Create a `Socket`:**
   - `Socket s = new Socket("localhost", 4444);`
   - *Reasoning:* A client `Socket` is created to initiate a connection request to a specific server identified by its IP address (`"localhost"` for the same machine) and port number (`4444`).
2. **Get `OutputStream` from Socket:**
   - `OutputStream os = s.getOutputStream();`
   - *Reasoning:* Obtain the output stream connected to the socket to send data *from* the client *to* the server.
3. **Create `PrintStream` for easy text sending:**
   - `PrintStream ps = new PrintStream(os);`
   - *Reasoning:* Wrap the byte-oriented `OutputStream` in a character-oriented `PrintStream` to easily send lines of text data (`println()`).
4. **Declare and Send Data:**

    - `String data = "Hello"; ps.println(data);`
    - *Reasoning:* Data flows from the `String` to `PrintStream`, then through `OutputStream` to the client `Socket`, and across the network to the server `Socket`.
5. **Get `InputStream` from Socket:**
    - `InputStream is = s.getInputStream();`
    - *Reasoning:* Obtain the input stream connected to the socket to receive data *from* the server *to* the client.
6. **Create `BufferedReader` for easy text receiving:**
    - `BufferedReader br = new BufferedReader(new InputStreamReader(is));`
    - *Reasoning:* Chain streams: `InputStream` (bytes from socket) -> `InputStreamReader` (bytes to characters) -> `BufferedReader` (buffers characters for efficient line reading `readLine()`).
7. **Read Data from `BufferedReader`:**
    - `String data = br.readLine(); System.out.println(data);`
    - *Reasoning:* Reads the line of text sent by the server.

At the Server Side:

1. **Create `ServerSocket`:**
    - `ServerSocket ss = new ServerSocket(4444);`
    - *Reasoning:* A `ServerSocket` is created to listen for incoming connection requests on a specific port (`4444`). This port must be available.
2. **Accept Client Connection:**
    - `Socket s = ss.accept();`
    - *Reasoning:* The `accept()` method blocks until a client connects. Upon connection, it returns a new `Socket` object specific to that client connection, allowing communication with that client. The `ServerSocket` continues listening for other connections.
3. **Create `InputStream` from the client `Socket`:**
    - `InputStream is = s.getInputStream();`
    - *Reasoning:* Get the stream to receive data from *this specific connected client*.
4. **Create `BufferedReader`:**
    - `BufferedReader br = new BufferedReader(new InputStreamReader(is));`
    - *Reasoning:* Chain streams as on the client side to easily read text sent by the client.
5. **Read Data from `BufferedReader`:**
    - `String data = br.readLine(); System.out.println(data);`
    - *Reasoning:* Reads the line of text sent by the client.
6. **Get `OutputStream` from the client `Socket`:**
    - `OutputStream os = s.getOutputStream();`
    - *Reasoning:* Get the stream to send data *to* this specific connected client.
7. **Create `PrintStream`:**
    - `PrintStream ps = new PrintStream(os);`
    - *Reasoning:* Wrap the output stream to easily send text lines back to the client.
8. **Send Data to `PrintStream`:**
    - `String data = "Hi"; ps.println(data);`
    - *Reasoning:* Data flows from the `String` to `PrintStream`, through `OutputStream`, to the server's client `Socket`, and across the network back to the client `Socket`.

Socket Programming Code Examples

Example 1: Simple Send and Receive

- **Client (`ClientApp.java`):** Reads a line from console, sends it to server, reads a line from server, prints it.

  Java

```
import java.net.*;
import java.io.*;
public class ClientApp {
    public static void main(String[] args) throws Exception { //
Basic exception handling
        Socket s = new Socket("localhost", 4444); // Connect to
server

        OutputStream os = s.getOutputStream();
        PrintStream ps = new PrintStream(os);

        BufferedReader br1 = new BufferedReader(new
InputStreamReader(System.in)); // Read from console
        String data1 = br1.readLine();
        ps.println(data1); // Send to server

        InputStream is = s.getInputStream();
        BufferedReader br2 = new BufferedReader(new
InputStreamReader(is)); // Read from server
        String data2 = br2.readLine();
        System.out.println(data2); // Print server response

        s.close(); // Close the socket (important!)
        // br1, ps, br2, os, is are typically closed when the socket
is closed or via try-with-resources
    }
}
```

- **Server (`ServerApp.java`):** Listens on a port, accepts one client, reads a line from client, prints it, reads a line from console, sends it to client.

  Java

```
import java.net.*;
import java.io.*;
public class ServerApp {
    public static void main(String[] args) throws Exception { //
Basic exception handling
        ServerSocket ss = new ServerSocket(4444); // Create server
socket
        System.out.println("Server waiting for client on port
4444..."); // Added for clarity
        Socket s = ss.accept(); // Wait for a client connection
        System.out.println("Client connected!"); // Added for clarity

        InputStream is = s.getInputStream();
        BufferedReader br1 = new BufferedReader(new
InputStreamReader(is)); // Read from client
```

```
        String data1 = br1.readLine();
        System.out.println("Client sent: " + data1); // Print client
data

        OutputStream os = s.getOutputStream();
        PrintStream ps = new PrintStream(os);

        BufferedReader br2 = new BufferedReader(new
InputStreamReader(System.in)); // Read from console
        System.out.print("Enter response to client: "); // Added for
clarity
        String data2 = br2.readLine();
        ps.println(data2); // Send to client

        s.close(); // Close client socket
        ss.close(); // Close server socket (stops listening)
        // br1, ps, br2, os, is streams are typically closed with
their underlying socket/stream
    }
}
```

- **Command Line Execution and Output (Example using inputs "Hello" and "Hi"):**
- `CMD-Server`
- `D:\FullstackJava830\JAVA830\Networking>java ServerApp`
- `Server waiting for client on port 4444...`
- `Client connected!`
- `Client sent: Hello`
- `Enter response to client: Hi`
-
- `CMD-Client`
- `D:\FullstackJava830\JAVA830\Networking>java ClientApp`
- `Hello`
- `Hi`

Example 2: Continuous Communication with Exit Condition ("bye")

- **Client (`ClientApp.java`):** Enters a loop to continuously read from console, send to server, receive from server, and print. Exits if both client sends "bye" and server responds with "bye".

  Java

```
import java.net.*;
import java.io.*;
public class ClientApp {
    public static void main(String[] args) throws Exception {
        Socket s = new Socket("localhost", 4444);

        OutputStream os = s.getOutputStream();
        PrintStream ps = new PrintStream(os);
        BufferedReader consoleReader = new BufferedReader(new
InputStreamReader(System.in)); // Read from console

        InputStream is = s.getInputStream();
        BufferedReader serverReader = new BufferedReader(new
InputStreamReader(is)); // Read from server
```

```
        System.out.println("Client started. Type 'bye' to exit."); //
Added for clarity

        while (true) { // Continuous communication loop
            System.out.print("Client says: "); // Added for clarity
            String data1 = consoleReader.readLine(); // Read from
console
            ps.println(data1); // Send to server

            String data2 = serverReader.readLine(); // Read from
server
            System.out.println("Server says: " + data2); // Print
server response

            // Exit condition
            if (data1.equalsIgnoreCase("bye") && data2 != null &&
data2.equalsIgnoreCase("bye")) {
                System.out.println("Client exiting."); // Added for
clarity
                s.close(); // Close socket
                System.exit(0); // Exit application
            }
             // Optional: Add a break condition if only client sends
"bye" and server doesn't respond "bye"
             if (data1.equalsIgnoreCase("bye") && (data2 == null || !
data2.equalsIgnoreCase("bye"))) {
                 System.out.println("Sent bye, but server didn't
respond bye. Exiting anyway.");
                 s.close();
                 System.exit(0);
             }
        }
    }
}
```

- **Server (`ServerApp.java`):** Listens on a port, accepts one client, enters a loop to continuously read from client, print it, read from console, send to client. Exits if both client sends "bye" and server responds with "bye".

  Java

```
import java.net.*;
import java.io.*;
public class ServerApp {
    public static void main(String[] args) throws Exception {
        ServerSocket ss = new ServerSocket(4444);
        System.out.println("Server waiting for client on port
4444..."); // Added for clarity
        Socket s = ss.accept();
        System.out.println("Client connected!"); // Added for clarity

        InputStream is = s.getInputStream();
        BufferedReader clientReader = new BufferedReader(new
InputStreamReader(is)); // Read from client

        OutputStream os = s.getOutputStream();
        PrintStream ps = new PrintStream(os);
        BufferedReader consoleReader = new BufferedReader(new
InputStreamReader(System.in)); // Read from console
```

```
        System.out.println("Server ready. Type response to client.");
// Added for clarity

        while (true) { // Continuous communication loop
            String data1 = clientReader.readLine(); // Read from
client
            if (data1 == null) { // Handle client disconnection
                 System.out.println("Client disconnected.");
                 break; // Exit loop if client disconnects
            }
            System.out.println("Client says: " + data1); // Print
client data

            System.out.print("Server says: "); // Added for clarity
            String data2 = consoleReader.readLine(); // Read from
console
            ps.println(data2); // Send to client

            // Exit condition
            if (data1.equalsIgnoreCase("bye") &&
data2.equalsIgnoreCase("bye")) {
                System.out.println("Server exiting."); // Added for
clarity
                s.close(); // Close client socket
                ss.close(); // Close server socket
                System.exit(0); // Exit application
            }
        }
        s.close(); // Ensure sockets are closed if loop breaks for
other reasons
        ss.close();
    }
}
```

- **Command Line Execution and Output (Example Conversation):**
- Client-CMD
- D:\FullstackJava830\JAVA830\Networking>java ClientApp
- Client started. Type 'bye' to exit.
- Client says: Hello
- Server says: Hi
- Client says: How are you?
- Server says: Fine
- Client says: What are you doing?
- Server says: Learning Java
- Client says: bye
- Server says: bye
- Client exiting.
-
- Server CMD
- D:\FullstackJava830\JAVA830\Networking>java ServerApp
- Server waiting for client on port 4444...
- Client connected!
- Client says: Hello
- Server says: Hi
- Client says: How are you?
- Server says: Fine
- Client says: What are you doing?
- Server says: Learning Java
- Client says: bye
- Server says: bye

```
Server exiting.
```