**Unidad 2. Comunicación en red con sockets.**

1. **Introducción:**

* **Red:** Conjunto de dispositivos interconectados de forma alámbrica o inalámbrica.
* **Internet:** Conjunto de redes interconectadas a nivel mundial.
* **Comunicación:** Proceso de traslado de información entre dispositivos utilizando una red o Internet. Elementos de la comunicación:
* **Emisor:** Genera la información y transfiere al medio.
* **Receptor:** Recibe la información a través del medio.
* **Mensaje:** Información que se transmite.
* **Medio (o canal):** Elemento físico por el que viaja el mensaje.
* **Código:** Sistema de señales utilizado para transmitir el mensaje.
* **Arquitectura cliente/servidor:** Modelo más utilizado para las comunicaciones entre dispositivos. Elementos:
* **Servidor:** Dispositivo que ofrece el servicio, que está a la espera de que se conecte un cliente y le haga una petición.
* **Cliente:** Dispositivo que **s**olicita los servicios de un servidor mediante una petición.
* **Arquitectura de redes:** Originalmente (años 70) las redes solamente funcionaban entre dispositivos del mismo fabricante y no existía compatibilidad hardware y software. Por ello se crearon otras soluciones:
* **Modelo de referencia OSI:** La organización ISO creó en 1983 el modelo de interconexión OSI (Open System Interconnection) para estandarizar la comunicación entre dispositivos. Sin embargo, es un modelo conceptual sobre el diseño y estructura, pero no define los **protocolos** (reglas que permiten la comunicación entre dispositivos). OSI estructura la comunicación en 7 niveles o capas. Acabó perdiendo importancia ante TCP/IP debido a la alta burocracia europea y la cantidad de recursos que requería su desarrollo.
* **Arquitectura TCP/IP:** Desarrollado por ARPA (Defensa de EEUU), se creó la red ARPANET para conectar universidades utilizando los protocolos TCP e IP.

**→ Arquitectura de capas:** Ambos modelos se basan en capas porque permiten dividir el problema de la comunicación en subproblemas más sencillos. Así, cada capa soluciona un problema y proporciona servicios a la siguiente.

* **TCP/IP:** Se compone de 4 niveles o capas. Cada nivel añade una cabecera a los datos que genera que es útil para los protocolos de cada capa del receptor, que los va eliminando a medida que avanza cada capa:

**A diagram of a diagram

Description automatically generated with medium confidence**

* **Capa de acceso a red:** Transforma la información recibida de la capa de Internet en señales transmisibles a través del medio físico, y viceversa, transforma las señales físicas en información transmisible a la capa de Internet. Este nivel implemente las comunicaciones entre máquinas conectadas directamente (físicamente, a través de un switch) mediante el uso de direcciones MAC de origen y destino.
* **Capa de Internet:** Permite enviar datos a través de redes conectadas mediante routers, utilizando el protocolo IP. Características:
* **Protocolo orientado a conexión:** No se anuncia su envío ni se requiere aceptación.
* **Falta de confirmación:** No hay certeza de entrega de datos.
* **No detecta errores:** Solamente de datos en la cabecera, pero no de la información.
* **Utiliza las direcciones IP:** Localiza dispositivos por sus interfaces de red.
* **Utiliza tablas de enrutamiento:** Sigue los caminos para llegar al destino.

Las cabeceras de los datagramas formados en este nivel incluyen la IP de origen y la IP de destino.

* **Capa de transporte:** Entrega la información a la aplicación correspondientes. Si hay varias aplicaciones conectadas, esta capa es la que se encarga de derivar los paquetes a cada aplicación del dispositivo, utilizando los protocolos TCP o UDP. Para identificar las aplicaciones a las que se entregan los paquetes (demultiplexación), se utilizan los puertos, que son números que identifican aplicaciones. Como se usan 16 bits para identificarlos, puede haber 65536 puertos, en tres clases (conocidos, registrados y dinámicos). Diferencias entre los protocolos:
* **TCP:** Permite que los paquetes lleguen en secuencia y sin errores, garantizando la recepción.
* **UDP:** No asegura que los datos lleguen al destino ni sin errores, pero agilizan la conexión (por ejemplo, para streaming).

Las cabeceras de los datagramas formados en este nivel incluyen el puerto de origen y el puerto de destino.

* **Capa de aplicación:** Permite la comunicación entre aplicaciones, mediante protocolos HTTP, FTP, IMAP, DNS, etc. Cada uno de estos protocolos tiene asignado el puerto en la capa de transporte, pero es la capa de aplicación la que se comunica con ellos. Los datos de esta capa también incorporan una cabecera que depende del tipo de protocolo utilizado.

1. **Comunicaciones en Java:**

* **Sockets:** Conectores o interfaces (de software) de entrada/salida de datos, que permiten la comunicación entre procesos. Se identifican mediante el número de puerto, por lo que la comunicación con el servidor debe incluir **tanto la IP como el puerto que permanece a la escucha** (LISTEN status) de dicha de conexión. Una vez se establece la conexión (y es aceptada por el servidor), este **genera un nuevo socket con otro número de puerto** para atender esa conexión, dejando libre el puerto original para nuevas peticiones de conexión. Se encuentran en la capa de transporte, por lo que utilizan los protocolos **TCP** (para sockets orientados a conexión) o **UDP** (para sockets no orientados a conexión).
* **Sockets orientados a conexión:** Para manipular estos sockets con Java se utilizan las clases:
* **ServerSocket:** Permite crear el socket del servidor, y se pueden emplear los métodos siguientes (pueden lanzar IOException):
* **ServerSocket(int port):** Constructor, recibe el puerto que estará a la escucha:

ServerSocket server = new ServerSocket(1234);

* **Socket accept():** Crea un hilo queda a la espera de una conexión de cliente (llamada bloqueante). Una vez se produzca la conexión, el método devolverá un objeto de tipo Socket que permite la comunicación con el cliente.

Socket cliente = server.accept();

* **close():** Cierra el objeto ServerSocket.

server.close();

* **Socket:** Representa un extremo de la comunicación: si se crea con el método accept() de la clase ServerSocket será el extremo servidor, pero si se crea con el constructor de Socket será el cliente. Métodos de la clase Socket:
* **Socket(String host, int port):** Crea un socket y lo conecta al host y puerto indicados.

Socket cliente = new Socket(“127.0.0.1”, 1234);

* **close():** Cierra el objeto Socket.
* **Flujo de datos:** Los datos que se transmiten y reciben a través de un socket se pueden manipular como streams de información. Los métodos de Socket getInputStream() y getOutputStream() generan objetos de tipo InputStream y OutputStream respectivamente, que permiten leer o escribir **bytes** en el stream. Por su parte, las clases DataInputStream y DataOutputStream permiten escribir y leer otros tipos de datos en el stream del socket, con métodos mucho más útiles (readInt(), writeInt(), readUTF(), writeUTF(), etc.). Ejemplo:

public class Server {  
 public static void main(String[] args) {  
 try {  
 System.*out*.println("SERVIDOR: Creado...");  
 ServerSocket server = new ServerSocket(1234);  
 System.*out*.println("SERVIDOR: Esperando conexión del

cliente...");  
  
 Socket conexionCliente = server.accept();  
 System.*out*.println("SERVIDOR: Cliente conectado.");  
  
 InputStream entrada = conexionCliente.getInputStream();  
 OutputStream salida = conexionCliente.getOutputStream();  
  
 DataInputStream flujoEntrada = new DataInputStream(entrada);  
 DataOutputStream flujoSalida = new DataOutputStream(salida);  
  
 System.*out*.println("SERVIDOR: Envío información al

cliente...");  
 flujoSalida.writeUTF("¡Hola, soy el servidor!");  
 System.*out*.println("SERVIDOR: Y espero una respuesta....");  
 String lineaRecibida = flujoEntrada.readUTF();  
 System.*out*.println("SERVIDOR: El mensaje recibido es: " +

lineaRecibida);  
 conexionCliente.close();  
 server.close();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
}

public class Client {  
 public static void main(String[] args) {  
 try {  
 System.*out*.println("CLIENTE: Creado...");  
 Socket client = new Socket("localhost", 1234);  
 System.*out*.println("CLIENTE: Conectando al servidor...");  
  
 InputStream entrada = client.getInputStream();  
 OutputStream salida = client.getOutputStream();  
  
 DataInputStream flujoEntrada = new DataInputStream(entrada);  
 DataOutputStream flujoSalida = new DataOutputStream(salida);  
  
 System.*out*.println("CLIENTE: Espero el saludo del

servidor...");  
 String lineaRecibida = flujoEntrada.readUTF();  
 System.*out*.println("CLIENTE: El mensaje recibido es: " +

lineaRecibida);  
 flujoSalida.writeUTF("¡Hola, soy el cliente!");  
  
 client.close();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
}

Cada programa se ejecuta de forma independiente y generará una consola propia. El servidor enviará los datos y gestionará la conexión, quedando a la espera de una conexión, que se produce cuando se lanza el cliente.

* **Conexión con múltiples clientes:** Es necesario crear hilos para que un servidor pueda atender de forma concurrente varias peticiones de clientes, de lo contrario, tendrá que atenderlos uno a uno. De este modo, cada vez que reciba una solicitud de conexión, el servidor debe lanzar un hilo que gestione la conexión con el cliente. De esta forma, aunque un hilo se quede bloqueado esperando datos de un cliente, el resto pueden funcionar.
* **Lanzar varios hilos para diferentes clientes:** Se debe crear una clase de gestión de conexiones que implemente la interfaz Runnable y que pueda lanzarse en un hilo independiente por cada conexión. Ejemplo:

class GestorConexion implements Runnable {  
 Socket socket;  
  
 public GestorConexion(Socket socket) {  
 this.socket = socket;  
 }  
  
 @Override  
 public void run() {  
 try {  
 InputStream entrada = socket.getInputStream();  
 OutputStream salida = socket.getOutputStream();  
 DataInputStream flujoEntrada = new DataInputStream(entrada);  
 DataOutputStream flujoSalida = new DataOutputStream(salida);  
  
 Scanner scanner = new Scanner(System.*in*);  
 String mensaje;  
 do {  
 mensaje = flujoEntrada.readUTF();  
 System.*out*.println("El mensaje recibido es " + mensaje);  
 } while (!mensaje.equalsIgnoreCase("fin"));  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
}

Luego, esta clase se llama desde el servidor para crear tantas conexiones e hilos como clientes haya. El servidor estará en bucle mientras esté activo y aceptará nuevas conexiones:

public class Server {  
 public static void main(String[] args) {  
 try (ServerSocket server = new ServerSocket(1234)) {  
 System.*out*.println("-----SERVIDOR-----");  
 System.*out*.println("Esperando la conexión del cliente...");  
  
 while (true) {  
 Socket conexionCliente = server.accept();  
 System.*out*.println("Cliente conectado: " +

conexionCliente.getInetAddress().getHostAddress());  
 GestorConexion gestor = new

GestorConexion(conexionCliente);  
 Thread thread = new Thread(gestor);  
 thread.start();  
 }  
 }catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
}

Por su parte, el cliente gestionará lo que hará, sin afectar en nada ni al hilo ni al servidor:

public class Client {  
 public static void main(String[] args) {  
 try (Socket client = new Socket("localhost",1234)) {  
 System.*out*.println("-----CLIENTE-----");  
 OutputStream salida = client.getOutputStream();  
 DataOutputStream flujoSalida = new DataOutputStream(salida);  
  
 Scanner scanner = new Scanner(System.*in*);  
 String mensaje;  
 do {  
 System.*out*.print("Escribe la línea a enviar: ");  
 mensaje = scanner.nextLine();  
 flujoSalida.writeUTF(mensaje);  
 } while (!mensaje.equalsIgnoreCase("fin"));  
 System.*out*.println("Finalizando cliente.");  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
}

**→ Lanzar varias instancias de un programa:** Hay que configurar IntelliJ para poder hacerlo. Seleccionar el programa, click derecho > More Run/Debug > Modify Run Configuration > Modify Options > Allow Multiple Instances.

* **Escritura y lectura simultánea:** La comunicación entre cliente y servidor es bilateral, de modo que es útil tener activos dos hilos en cada extremo: uno para enviar mensajes y otro para recibir mensajes. De este modo, los mensajes son concurrentes, y no hay que esperar respuesta para enviar otro mensaje.
* **Clase Server:** Gestiona la conexión y lanza dos hilos (uno de entrada y otro de salida):

public class Server {  
 public static void main(String[] args) {  
 try(ServerSocket server = new ServerSocket(1234)) {  
 System.*out*.println("-----SERVIDOR-----");  
 System.*out*.println("Esperando conexión con cliente...");  
 Socket client = server.accept();  
 System.*out*.println("¡Cliente conectado!");  
  
 InputManager input = new InputManager(client);  
 Thread inputThread = new Thread(input);  
 inputThread.start();  
 OutputManager output = new OutputManager(client);  
 Thread outputThread = new Thread(output);  
 outputThread.start();  
  
 try {  
 inputThread.join();  
 outputThread.join();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 System.*out*.println("Finalizado servidor.");  
 } catch(IOException e) {  
 e.printStackTrace();  
 }  
 }  
}

* **Clase Client:** También lanza dos hilos independientes:

public class Client {  
 public static void main(String[] args) {  
 System.*out*.println("-----CLIENTE-----");  
 try (Socket socket = new Socket("localhost", 1234)) {  
 InputManager input = new InputManager(socket);  
 Thread inputThread = new Thread(input);  
 inputThread.start();  
 OutputManager output = new OutputManager(socket);  
 Thread outputThread = new Thread(output);  
 outputThread.start();  
  
 try {  
 inputThread.join();  
 outputThread.join();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 System.*out*.println("Finalizando cliente.");  
 } catch(IOException e) {  
 e.printStackTrace();  
 }  
 }  
}

* **Clase para la gestión del hilo de salida:**

public class InputManager implements Runnable{  
 Socket socket;  
  
 public InputManager(Socket socket) {  
 this.socket = socket;  
 }  
  
 @Override  
 public void run() {  
 try {  
 InputStream entrada = socket.getInputStream();  
 DataInputStream flujoEntrada = new DataInputStream(entrada);  
  
 String mensaje;  
 do {  
 mensaje = flujoEntrada.readUTF();  
 System.*out*.println("Mensaje recibido: " + mensaje);  
 } while (!mensaje.equalsIgnoreCase("fin"));  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
}

* **Clase para la gestión del hilo de entrada:**

public class OutputManager implements Runnable{  
 Socket socket;  
  
 public OutputManager(Socket socket) {  
 this.socket = socket;  
 }  
  
 @Override  
 public void run() {  
 Scanner scanner = new Scanner(System.*in*);  
 try {  
 OutputStream salida = socket.getOutputStream();  
 DataOutputStream flujoSalida = new DataOutputStream(salida);  
  
 String mensaje;  
 do {  
 System.*out*.print("Escribe un mensaje: ");  
 mensaje = scanner.nextLine();  
 flujoSalida.writeUTF(mensaje);  
 } while (!mensaje.equalsIgnoreCase("fin"));  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
}