UADY

FACULTAD DE MATEMÁTICAS

Sistemas Distribuidos

Unidad 2

Comunicación entre procesos

Sistemas Distribuidos

- **2. COMUNICACIÓN ENTRE PROCESOS**

  El alumno aplicará los modelos de comunicación entre procesos distribuidos que le permitan desarrollar aplicaciones acordes a las arquitecturas de sistemas distribuidos.

  - Introducción.
  - Protocolos de Internet para la comunicación de procesos.
    - Sockets UDP.
    - ***Sockets TCP.***
    - Comunicación en grupo: multidifusión IP.
  - Representación externa de datos.
  - Comunicación entre procesos distribuidos.
    - RPC.
    - RMI.

Comunicación por *streams* TCP

- Un *stream* es un flujo de bytes en el que puede escribirse y desde el que pueden leerse datos.
- Oculta:
  - Tamaño de los mensajes. No importa el tamaño.
  - Mensajes perdidos. Acuse de recibo y retransmisión de perdidos.
  - Control de flujo de transmisión.
  - Duplicación y ordenación de mensajes.
  - Destinos de los mensajes. *Connect* y *Accept*. Los procesos no requieren conocer la dirección y puerto del destino para intercambiar mensajes.

Comunicación por *streams* TCP

- Al momento de establecer una conexión, un extremo juega el papel de cliente y el otro, el papel de servidor, aunque después se comuniquen de igual a igual.
- El rol de cliente implica la creación de un conector de tipo stream sobre cualquier puerto y la posterior petición de conexión con el servidor en su puerto de servicio.
- El papel de servidor involucra la creación de un conector de escucha ligado al puerto de servicio y la espera de clientes por conexiones.

Comunicación por *streams* TCP

- Se crean un par de conectores, el del cliente y el del servidor por donde se tiene un par de streams, uno en cada dirección.
- Concordancia de ítems de datos. Los dos procesos que se comunican necesitan estar de acuerdo en el tipo de datos transmitidos por el stream.
- Bloqueo. Los datos escritos en un stream se almacenan en el búfer del conector destino. Bloqueo de espera de datos. Bloqueo de espera de transmisión.
- Hilos. Cuando un servidor acepta una conexión, genera un nuevo hilo para comunicarse con el nuevo cliente.

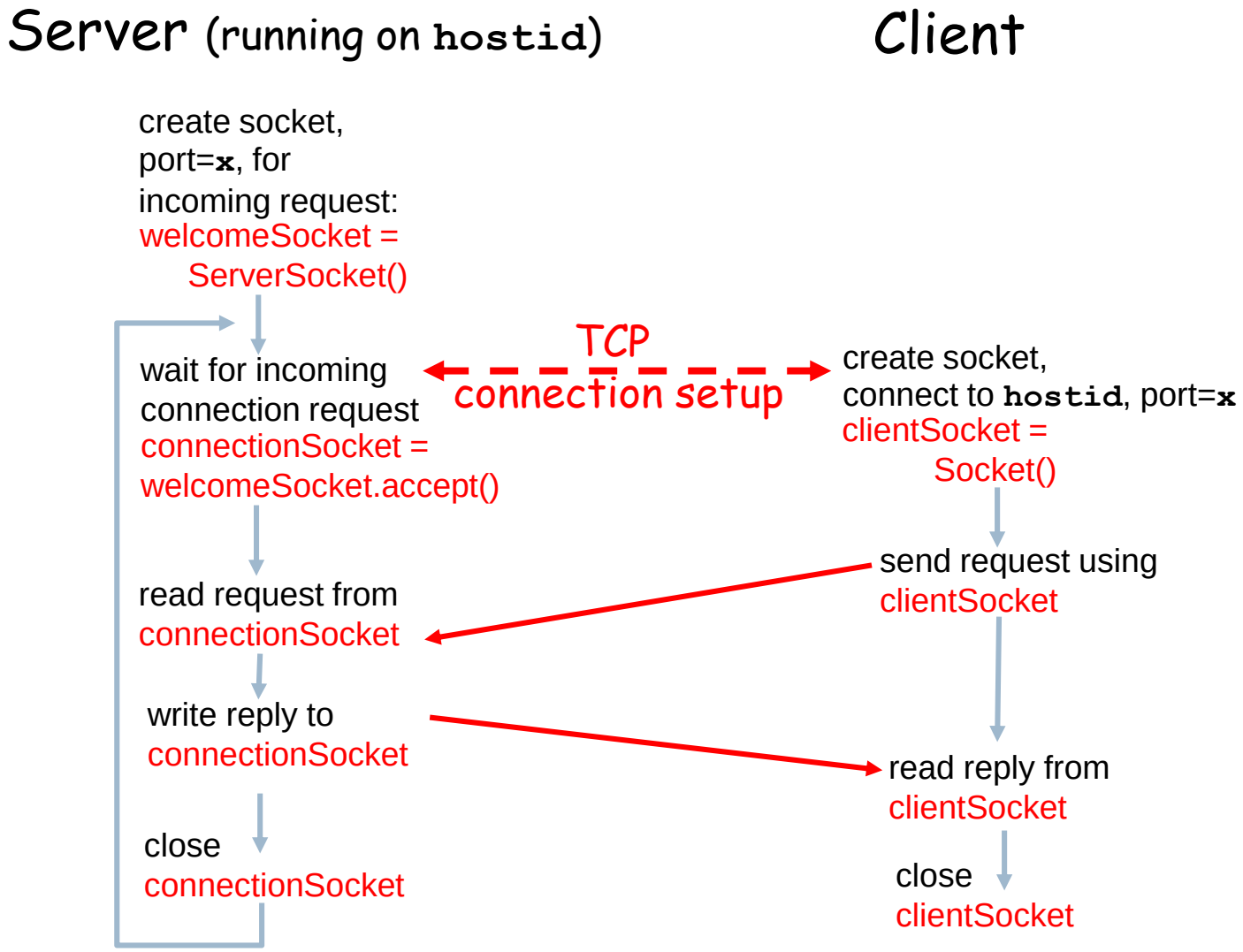
Modelo de fallo

- Los streams TCP utilizan una suma de comprobación para detectar y rechazar los paquetes corruptos.
- Utilizan un número de secuencia para detectar y eliminar paquetes duplicados.
- Utilizan timeouts y retransmisión de los paquetes perdidos.
- Si la pérdida de paquetes sobrepasa cierto límite, o hay congestión en la red, que no permita entregar los acuses de recibo, TCP puede declarar rota la conexión.
- TCP no es fiable pues no garantiza la entrega de mensajes en presencia de algún tipo de problema.

API Java para *streams* TCP

- Clase ServerSocket.
  - Para uso de un servidor. Crea un socket en el puerto de servidor que escucha las peticiones de conexión de los clientes.
- Clase Socket.
  - Para uso del cliente. Se utiliza un constructor para crear un socket, especificando el nombre DNS y el puerto del host. Además, realiza la conexión con el servidor especificado.

Interacción con socket Cliente/Servidor: TCP



Clase ServerSocket

- Ciclo de vida básico de un servidor:
  1. Crear un nuevo *socket* servidor en un puerto específico usando un constructor.
  2. El *socket* servidor espera conexiones de los clientes por medio de un método de aceptación. Al establecer una conexión, el método de aceptación regresa como valor un *socket* normal por el cual se transmitirá la información.
  3. Dependiendo de las necesidades, se obtienen flujos de entrada y/o salida.
  4. El servidor y el cliente interactúan de acuerdo a un protocolo acordado.
  5. El servidor, el cliente o ambos cierran la conexión.
  6. El servidor regresa al paso 2 a esperar la próxima conexión.

- En caso de que el paso 4 tome un tiempo muy largo o indefinido, se puede crear un hilo para que interactúe con el cliente y de esta manera el servidor estará listo para atender la siguiente conexión.

Constructores ServerSocket

- **ServerSocket(int port) throws IOException,BindException**
- **ServerSocket(int port, int queuelength) throws IOException,BindException**
- **ServerSocket(int port, int queuelength, InetAddress bindAddress) throws IOException**

- Un valor de cero para el puerto hace que el sistema seleccione uno al azar. Esto no es recomendable ya que los clientes requieren conocer el número de puerto.
- El sistema almacena las peticiones de conexión de acuerdo a una fila (FIFO) hasta un límite máximo después del cual son rechazadas. La clase ServerSocket permite variar la longitud de la fila, pero ésta nunca podrá ser mayor que la capacidad del sistema operativo.
- El último constructor permite seleccionar una dirección para sólo atender las peticiones que lleguen a ella. Esto es útil en sistemas con múltiples interfaces.
- Las excepciones suelen deberse principalmente a dos causas: el puerto especificado ya está en uso o no se tienen privilegios para seleccionar ese puerto.

Métodos ServerSocket

- **public InetAddress getInetAddress()**
- **public int getLocalPort()**
- **public Socket accept() throws IOException**
  - Este método bloquea la ejecución del servidor hasta recibir una conexión. El valor de retorno es un objeto de la clase Socket que tiene disponibles los métodos correspondientes (getInputStream(), getOutputStream(), etc.).
- **public void close() throws IOException**
  - Cerrar un socket servidor permite a otros servidores asignarse a ese puerto, pero cerrar el socket creado por el servidor sólo permite a otros clientes conectarse.
- **public String toString()**
  - Regresa el valor del puerto al que está conectado el servidor entre otros datos.
- **public synchronized void setSoTimeout(int timeout) throws SocketException**
  - Establece los milisegundos que debe esperar el método accept() por una conexión antes de lanzar una excepción. Un valor de cero indica una espera indefinida.

Clase Socket

- Estilo común de uso de los sockets:
  1. Crear un nuevo socket usando un constructor de la clase.
  2. El socket trata de conectarse al host remoto.
  3. Una vez establecida la conexión, tanto el host local como el remoto obtienen flujos de entrada y salida del socket para transmitir datos.
  4. Cuando la transmisión se complete, uno o ambos lados cierran la conexión.

Constructores Socket

- **Socket (String host, int port) throws UnknownHostException, IOException**
- **Socket (InetAddress host, int port) throws IOException**
- **Socket (String host, int port, InetAddress interface, int localPort) throws IOException**
- **Socket (InetAddress host, int port, InetAddress interface, int localPort) throws IOException**

- Estos constructores no sólo crean un objeto socket, sino que también tratan de conectarse al host especificado.
- Las excepciones pueden lanzarse debido a que el host es desconocido, está caído, no está aceptando conexiones, etc.
- Los dos últimos constructores además permiten especificar la dirección y el puerto por donde se van a transmitir los datos en caso de estar en un ambiente con múltiples hosts. Si se pasa un cero como puerto local, entonces se escoge al azar uno disponible a partir del 1024.

Métodos Socket

- **public InetAddress getInetAddress()**
- **public int getPort()**
- **public InetAddress getLocalAddress()**
- **public int getLocalPort()**

- **public InputStream getInputStream() throws IOException**
  - Este método regresa un canal a través del cual se puede recibir información del socket. El valor de retorno suele convertirse en otro tipo de flujo de entrada con mayor funcionalidad.
- **public OutputStream getOutputStream() throws IOException**
  - Este método regresa un canal a través del cual se puede mandar información al socket. El valor de retorno suele convertirse en otro tipo de flujo de salida con mayor funcionalidad.

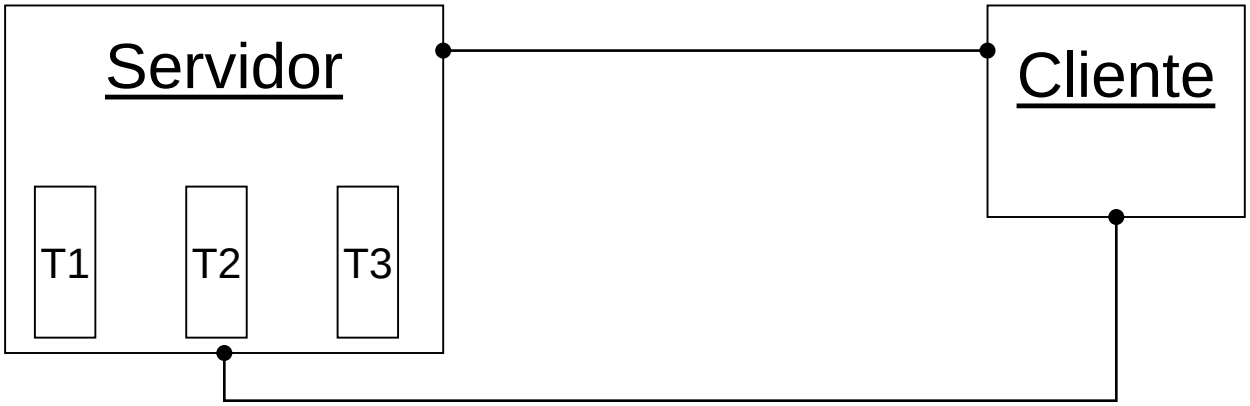
Métodos Socket

- **public synchronized void close() throws IOException**
  - Siempre es conveniente cerrar una conexión que no está en uso, ya que de lo contrario no se podrán conectar otras aplicaciones a ese puerto.
- **public String toString()**
  - Regresa en un formato no modificable todos los valores de los métodos get para conocer la dirección y el puerto de los sockets.
- **public void setTcpNoDelay(boolean on) throws SocketException**
  - Un valor verdadero asegura que los paquetes son enviados tan pronto como estén listos, mientras que uno falso obliga a seguir un esquema donde el host local debe esperar a recibir un paquete del host remoto antes de mandar otro paquete.
- **public void setSoLinger(boolean on, int seconds) throws SocketException**
  - Esta opción indica qué hacer con los paquetes no enviados después de cerrada la conexión. Un valor igual o mayor que cero indica que debe esperarse esa cantidad de tiempo antes de desechar los paquetes. Y un -1, que debe esperarse indefinidamente hasta mandar todo.
- **public synchronized void setSoTimeout(int ms) throws SocketException**
  - Al leer datos del socket, la función se bloquea hasta recibir suficientes bytes. Pero al especificar un tiempo mayor a cero milisegundos, se lanza una excepción al expirar el plazo sin verse afectada la conexión.

Servidor Concurrente (con múltiples hilos)

- La ejecución concurrente es fundamental para el servidor porque la concurrencia permite que múltiples clientes obtengan un servicio sin tener que esperar a que el servidor termine con peticiones anteriores.
- En este diseño, el servidor crea un nuevo *thread o proceso* para manejar a cada cliente.
- ¿Qué pasa con el identificador del cliente y el servidor?
- El protocolo de transporte asigna un identificador a cada cliente así como a cada servicio.
- La máquina en el servidor usa la combinación única del cliente y el servidor para identificar la copia correcta del servidor concurrente.

Servidor Concurrente (con múltiples hilos)



Comunicación bidireccional

Estrategia de un servidor con hilos

Ejemplo. TCPClient

```
import java.net.*;
import java.io.*;
public class TCPClient {
        public static void main (String args[]) {
        // argumentos: Mensaje y nombre del servidor
        Socket s = null;
           try{
                   int serverPort = 7896;
                   s = new Socket(args[1], serverPort);
                   DataInputStream in = new DataInputStream( s.getInputStream());
                   DataOutputStream out =
                           new DataOutputStream( s.getOutputStream());
                   out.writeUTF(args[0]);        // UTF is a string encoding
                   String data = in.readUTF();
                   System.out.println("Received: "+ data) ;
           }catch (UnknownHostException e){
                           System.out.println("Sock:"+e.getMessage());
           }catch (EOFException e){System.out.println("EOF:"+e.getMessage());
           }catch (IOException e){System.out.println("IO:"+e.getMessage());}
        }
        }
```

Ejemplo. TCPServer

```
import java.net.*;
import java.io.*;
public class TCPServer {
   public static void main (String args[]) {
                try{
                                int serverPort = 7896;
                                ServerSocket listenSocket = new ServerSocket(serverPort);
                                while(true) {                   Socket clientSocket = listenSocket.accept();
                                                Connection c = new Connection(clientSocket);
                                } } catch(IOException e) {System.out.println("Listen :"+e.getMessage());}
   } }
class Connection extends Thread {
                DataInputStream in;
                DataOutputStream out;
                Socket clientSocket;
                public Connection (Socket aClientSocket) {
                   try {          clientSocket = aClientSocket;
                                in = new DataInputStream( clientSocket.getInputStream());
                                out =new DataOutputStream( clientSocket.getOutputStream());
                                this.start();
                    } catch(IOException e)  {System.out.println("Connection:"+e.getMessage());}  }
                public void run(){
                   try {        String data = in.readUTF();
                                out.writeUTF(data);
                   } catch(EOFException e) {System.out.println("EOF:"+e.getMessage());
                   } catch(IOException e) {System.out.println("IO:"+e.getMessage());}
   } }
```