Programación de Servicios y Procesos

Tema 3 Programación de comunicaciones en Red

José Luis González Sánchez

Contenidos

1. Fundamentos de comunicaciones TCP/IP
2. Sockets
3. Programación de Sockets
4. Modelo Cliente-Servidor
5. Servicios de Sistema

Fundamento de Comunicaciones TCP/IP

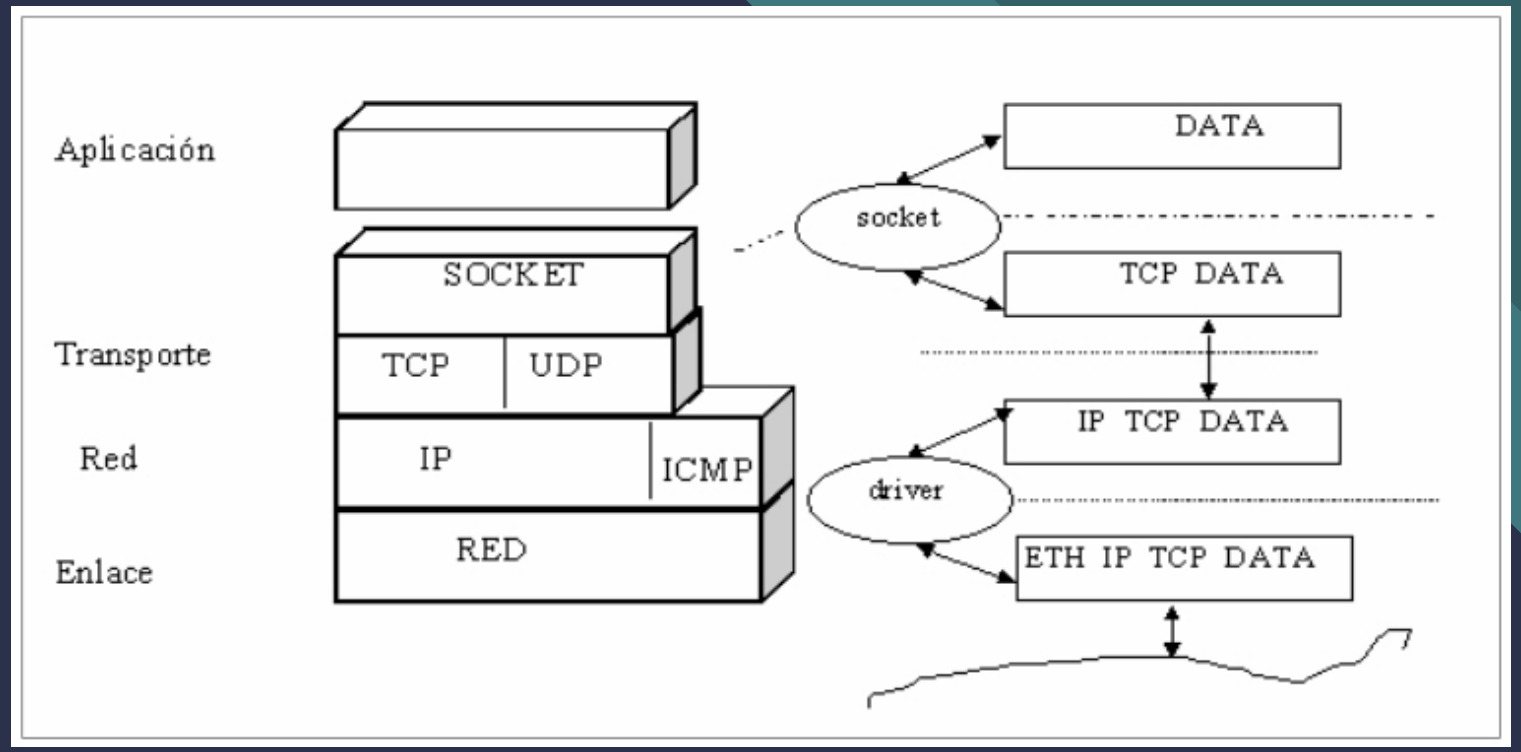
Usando la red para intercambiar datos

Fundamentos TCP/IP

- Las **comunicaciones entre diferentes sistemas se realizan típicamente a través de una red**, que usualmente denominamos local cuando la distancia es pequeña y extensa cuando se trata de equipos muy alejados (se mantiene intencionadamente la ambigüedad en los conceptos de distancia dada la gran variedad de redes existente).
- Tanto si usamos un tipo de red como otro existen multitud de protocolos para el intercambio de información.
- Básicamente se va a disponer de dos tipos de servicio claramente diferenciados:
  - **Un servicio que denominamos sin conexión**, que se parece al funcionamiento de un servicio de correo o de mensajería, en el que origen y destino intercambian uno o más bloques de información sin realizar una conexión previa y donde **no se dispone de un control de secuencia (el orden de entrega puede no corresponder al de envío) ni tampoco control de error (algunos de estos bloques de información pueden perderse sin que se reciba aviso alguno)**.
  - Otro **servicio orientado a la conexión**, en el que **disponemos de un stream que nos asegura la entrega de información ordenada y fiable, sin que se pueda perder información**. En este servicio, antes de enviar información se realiza un proceso de conexión, similar al funcionamiento de un teléfono (donde, antes de hablar, tenemos que completar el proceso de llamada).
- Cada uno de estos tipos de servicio será de aplicación para tareas diferentes y posee un conjunto de características que los hace aptos para diferentes usos.

Fundamentos TCP/IP. Modelo de Capas

- Los modelos de capas de dividen el proceso de comunicación en capas independientes.
- Cada capa proporciona servicios a la capa superior a través de una interfaz y a la vez recibe servicios de la capa inferior a través de la interfaz correspondiente.
- Este tipo de abstracción permite construir sistemas muy flexibles ya que se esconden los detalles de la implementación: la capa N sólo necesita saber que la capa N-1 le proporciona el servicio X, no necesita saber el mecanismo que utiliza dicha capa para lograr su objetivo.



Fundamentos TCP/IP. Modelo de Capas

- La **capa host-red**. Esta capa permite comunicar el ordenador con el medio que conecta el equipo a la red. Para ello primero debe permitir convertir la información en impulsos físicos (p.ej. eléctricos, magnéticos, luminosos) y además, debe permitir las conexiones entre los ordenadores de la red. En esta capa se realiza un direccionamiento físico utilizando las direcciones MAC.
- La **capa de red**. Esta capa es el eje de la arquitectura TCP/IP ya que permite que los equipos envíen paquetes en cualquier red y viajen de forma independiente a su destino (que podría estar en una red diferente). Los paquetes pueden llegar incluso en un orden diferente a aquel en que se enviaron, en cuyo caso corresponde a las capas superiores reordenándolos, si se desea la entrega ordenada. La capa de red define un formato de paquete y protocolo oficial llamado IP (Internet Protocol). El trabajo de la capa de red es entregar paquetes IP a su destino. Aquí la consideración más importante es decidir el camino que tienen que seguir los paquetes (encaminamiento), y también evitar la congestión. En esta capa se realiza el direccionamiento lógico o direccionamiento por IP, ya que ésta es la capa encargada de enviar un determinado mensaje a su dirección IP de destino.
- La **capa de transporte**. La capa de transporte permite que los equipos lleven a cabo una conversación. Aquí se definieron dos protocolos de transporte: TCP (Transmission Control Protocol) y UDP (User Datagram Protocol). El protocolo TCP es un protocolo orientado a conexión y fiable, y el protocolo UDP es un protocolo no orientado a conexión y no fiable En esta capa además se realiza el direccionamiento por puertos. Gracias a la capa anterior, los paquetes viajan de un equipo origen a un equipo destino. La capa de transporte se encarga de que la información se envíe a la aplicación adecuada (mediante un determinado puerto).
- La **capa de aplicación**. Esta capa engloba las funcionalidades de las capas de sesión, presentación y aplicación del modelo OSI. Incluye todos los protocolos de alto nivel relacionados con las aplicaciones que se utilizan en Internet (por ejemplo HTTP, FTP, TELNET).

Fundamentos TCP/IP. Conexiones TCP/UDP

- Existen dos tipos de conexiones:
- **TCP (Transmission Control Protocol)**. Es un protocolo orientado a la conexión que permite que un flujo de bytes originado en una máquina se entregue sin errores en cualquier máquina destino. Este protocolo fragmenta el flujo entrante de bytes en mensajes y pasa cada uno a la capa de red. En el diseño, el proceso TCP receptor reensambla los mensajes recibidos para formar el flujo de salida. TCP también se encarga del control de flujo para asegurar que un emisor rápido no pueda saturar a un receptor lento con más mensajes de los que pueda gestionar.
- **UDP (User Datagram Protocol)**. Es un protocolo sin conexión, para aplicaciones que no necesitan la asignación de secuencia ni el control de flujo TCP y que desean utilizar los suyos propios. Este protocolo también se utilizan para las consultas de petición y respuesta del tipo cliente-servidor, y en aplicaciones en las que la velocidad es más importante que la entrega precisa, como las transmisiones de voz o de vídeo. Uno de sus usos es en la transmisión de audio y vídeo en tiempo real, donde no es posible realizar retransmisiones por los estrictos requisitos de retardo que se tiene en estos casos.
- De esta forma, a la hora de programar nuestra aplicación deberemos elegir el protocolo que queremos utilizar según nuestras necesidades: TCP o UDP.

Fundamentos TCP/IP. Puertos

- Con la capa de red se consigue que la información vaya de un equipo origen a un equipo destino a través de su dirección IP. Pero para que una aplicación pueda comunicarse con otra aplicación es necesario establecer a qué aplicación se conectará. El método que se emplea es el de definir direcciones de transporte en las que los procesos pueden estar a la escucha de solicitudes de conexión. Estos puntos terminales se llaman puertos.
- Aunque muchos de los puertos se asignan de manera arbitraria, ciertos puertos se asignan, por convenio, a ciertas aplicaciones particulares o servicios de carácter universal. De hecho, la IANA (Internet Assigned Numbers Authority) determina, las asignaciones de todos los puertos. Existen tres rangos de puertos establecidos:
- Puertos conocidos [0, 1023]. Son puertos reservados a aplicaciones de uso estándar como: 21 – FTP (File Transfer Protocol), 22 – SSH (Secure SHell), 53 – DNS (Servicio de nombres de dominio), 80 – HTTP (Hypertext Transfer Protocol), etc.
- Puertos registrados [1024, 49151]. Estos puertos son asignados por IANA para un servicio específico o aplicaciones. Estos puertos pueden ser utilizados por los usuarios libremente.
- Puertos dinámicos [49152, 65535]. Este rango de puertos no puede ser registrado y su uso se establece para conexiones temporales entre aplicaciones.
- Cuando se desarrolla una aplicación que utilice un puerto de comunicación, optaremos por utilizar puertos comprendidos entre el rango 1024-49151.

Fundamentos TCP/IP. Nombres

- Los equipos informáticos se comunican entre sí mediante una dirección IP como 193.147.0.29. Sin embargo nosotros preferimos utilizar nombres como www.mec.es porque son más fáciles de recordar y porque ofrecen la flexibilidad de poder cambiar la máquina en la que están alojados (cambiaría entonces la dirección IP) sin necesidad de cambiar las referencias a él.
- El sistema de resolución de nombres (DNS) basado en dominios, en el que se dispone de uno o más servidores encargados de resolver los nombres de los equipos pertenecientes a su ámbito, consiguiendo, por un lado, la centralización necesaria para la correcta sincronización de los equipos, un sistema jerárquico que permite una administración focalizada y, también, descentralizada y un mecanismo de resolución eficiente.
- A la hora de comunicarse con un equipo, puedes hacerlo directamente a través de su dirección IP o puede poner su entrada DNS (p.ej. servidor.miempresa.com). En el caso de utilizar la entrada DNS el equipo resuelve automáticamente su dirección IP a través del servidor de nombres que utilice en su conexión a Internet.

Fundamentos TCP/IP. Clientes-Servidor

- Mediante la arquitectura cliente-servidor se definen servidores pasivos y clientes activos que crean conexiones hacia los servidores. Se realiza un diseño de aplicaciones asimétricas entre cliente y servidor. Es decir, son distintas las aplicaciones que se ejecutan en el servidor a las que se ejecutan en el cliente.
- **La comunicación entre un cliente y un servidor se puede realizar de dos formas distintas mediante conexiones (servicio orientado a conexión) o datagramas (servicio sin conexión).**
- Protocolo orientado a conexión (segmentos): se basa en una conexión establecida en la que queda fijado el destino de la conexión desde el momento de abrirla. Esta conexión permanecerá hasta que se cierre, como ocurre con una llamada telefónica. Este protocolo en TCP/IP se llama TCP (Transmission Control Protocol).
- Protocolo sin conexión (datagramas): por el contrario envía mensajes individuales con el destino grabado en cada uno de ellos, como en una comunicación a través de correo postal, donde no son necesarias las fases de establecimiento y liberación de conexión alguna. Este protocolo en TCP/IP se llama UDP (User Datagram Protocol).
- En este modelo de aplicaciones, el servidor suele tener un papel pasivo, respondiendo únicamente a las peticiones de los clientes, mientras que estos últimos son los que interactúan con los usuarios (y disponen de interface de usuario).
- La comunicación entre cliente y servidor se efectúa través de la red que los une, empleando, además de un protocolo de transporte (TCP o UDP), un protocolo de aplicación que varía de unas aplicaciones a otras y que, como indica su nombre, está codificado en la propia aplicación (mientras que el protocolo de transporte está incorporado en el sistema operativo y es accedido desde los programas Java a través de las clases que se estudiarán en este capítulo).

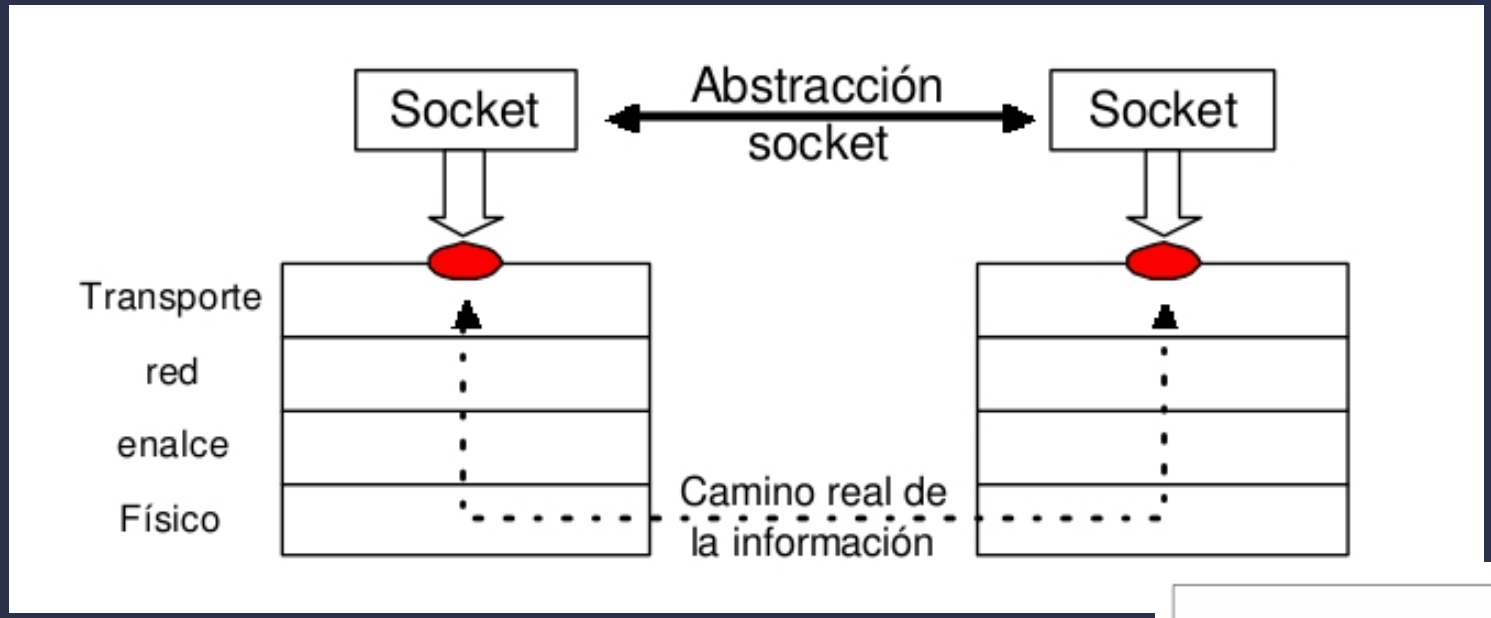
Sockets

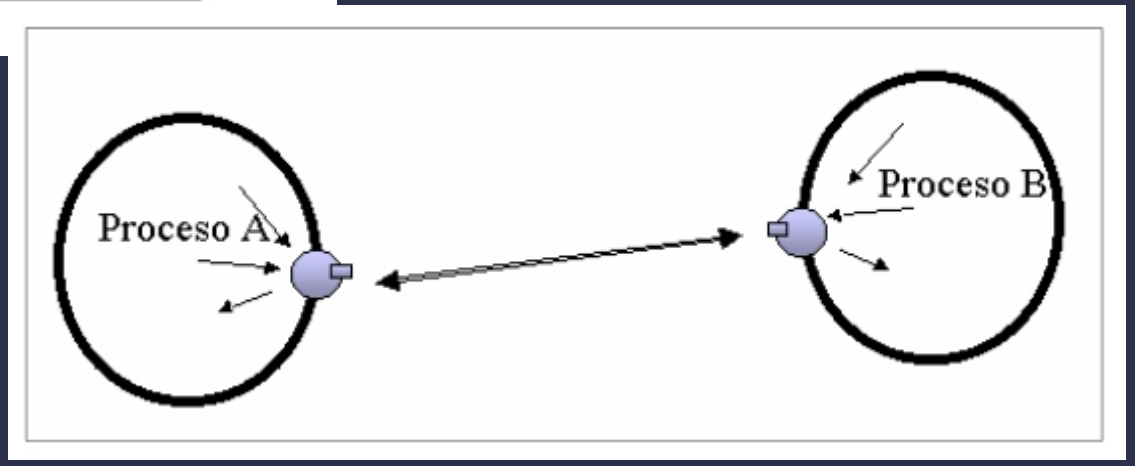
Programando las comunicaciones

Socket

- Los sockets son un sistema de comunicación entre procesos de diferentes máquinas de una red. Más exactamente, un socket es un punto de comunicación por el cual un proceso puede emitir o recibir información.
- Los sockets fueron desarrollados como un intento de generalizar el concepto de pipe (tubería unidireccional para la comunicación entre procesos en el entorno Unix) en 4.2BSD bajo contrato por DARPA (Defense Advanced Research Projects Agency). Sin embargo, fueron popularizados por Berckley Software Distribution, de la Universidad Norteamericana de Berkley.
- Los sockets utilizan una serie de primitivas para establecer el punto de comunicación, para conectarse a una máquina remota en un determinado puerto que esté disponible, para escuchar en él, para leer o escribir y publicar información en él, y finalmente para desconectarse. Con todas las primitivas que ofrecen los sockets, se puede crear un sistema de diálogo muy completo.
- **Un socket es un punto final de un proceso de comunicación. Es una abstracción que permite manejar de una forma sencilla la comunicación entre procesos, aunque estos procesos se encuentren en sistemas distintos, sin necesidad de conocer el funcionamiento de los protocolos de comunicación subyacentes.**
- Es así como estos “puntos finales” sirven de enlaces de comunicaciones entre procesos. Los procesos tratan a los sockets como descriptores de ficheros, de forma que se pueden intercambiar datos con otros procesos transmitiendo y recibiendo a través de sockets.
- Los mecanismos de comunicación entre procesos pueden tener lugar dentro de la misma máquina o a través de una red. Generalmente son usados en forma de cliente-servidor, es decir, cuando un cliente y un servidor establecen una conexión, lo hacen a través de un socket.

Socket


Socket
Abstracción
socket
Socket
Transporte
red
enalce
Físico
Camino real de
la información

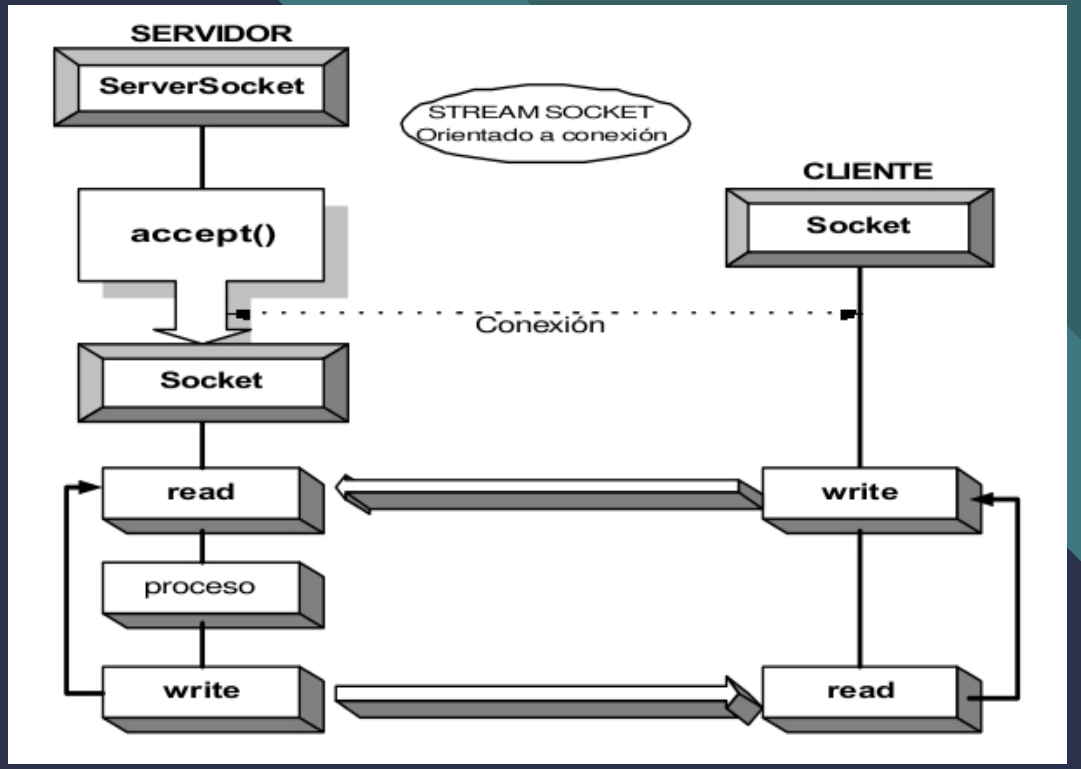

Proceso A
Proceso B

Socket. Tipos

- El tipo de sockets describe la forma en la que se transfiere información a través de ese socket.
- Existen muchos tipos de sockets, sin embargo, los más populares son:
  - Stream (TCP)
  - Datagram (UDP)
  - Raw (acceso directo al protocolo: root)

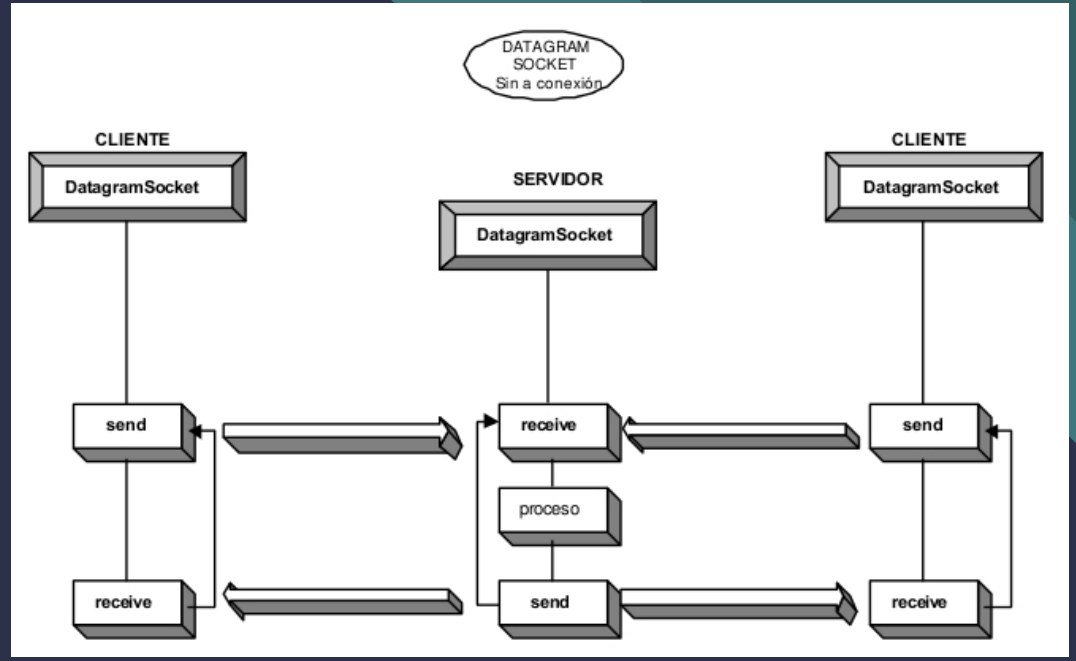
Socket. Stream (TCP)

- Son un servicio orientado a la conexión, donde los datos se transfieren sin encuadrarlos en registros o bloques, asegurándose de esta manera que los datos lleguen al destino en el orden de transmisión. Si se rompe la conexión entre los procesos, éstos serán informados de tal suceso para que tomen las medidas oportunas, por eso se dice que están libres de errores.
- El protocolo de comunicaciones con streams es un protocolo orientado a conexión, ya que para establecer una comunicación utilizando el protocolo TCP (Transmission Control Protocol), hay que establecer en primer lugar una conexión entre un par de sockets. Mientras uno de los sockets atiende peticiones de conexión (servidor), el otro solicita una conexión (cliente). Una vez que los dos sockets estén conectados, se pueden utilizar para transmitir datos en ambas direcciones.



Socket. Datagram (UDP)

- Son un servicio de transporte no orientado a la conexión. Son más eficientes que TCP, pero en su utilización no está garantizada la fiabilidad. Los datos se envían y reciben en paquetes, cuya entrega no está garantizada. Los paquetes pueden ser duplicados, perdidos o llegar en un orden diferente al que se envió.
- Las comunicaciones a través de datagramas usan UDP (User Datagram Protocol), lo que significa que, cada vez que se envíen datagramas es necesario enviar el descriptor del socket local y la dirección del socket que debe recibir el datagrama. Como se puede ver, hay que enviar datos adicionales cada vez que se realice una comunicación, aunque tiene la ventaja de que se pueden indicar direcciones globales y el mismo mensaje llegará a un muchas máquinas a la vez.



Socket. Raw

- Son sockets que dan acceso directo a la capa de software de red subyacente o a protocolos de más bajo nivel. Se utilizan sobre todo para la depuración del código de los protocolos.

Socket. Socket Stream vs Datagrama

- El problema aparece al momento de decidir por **cual protocolo o tipo de socket usar**. La decisión depende de la aplicación cliente/servidor que se esté desarrollando; aunque hay algunas diferencias entre los protocolos que sirven para ayudar en la decisión y utilizar un determinado tipo de socket.
- En UDP, cada vez que se envía un datagrama, hay que enviar también el descriptor del socket local y la dirección del socket que va a recibir el datagrama, luego los mensajes son más grandes que los TCP. Como el protocolo TCP está orientado a conexión, hay que establecer esta conexión entre los dos sockets antes de nada, lo que implica un cierto tiempo empleado en el establecimiento de la conexión, que no es necesario emplear en UDP.
- En UDP hay un límite de tamaño de los datagramas, establecido en 64 kilobytes, que se pueden enviar a una localización determinada, mientras que TCP no tiene límite; una vez que se ha establecido la conexión, el par de sockets funciona como los streams: todos los datos se leen inmediatamente, en el mismo orden en que se van recibiendo.
- UDP es un protocolo desordenado, no garantiza que los datagramas que se hayan enviado sean recibidos en el mismo orden por el socket de recepción. Al contrario, TCP es un protocolo ordenado, garantiza que todos los paquetes que se envíen serán recibidos en el socket destino en el mismo orden en que se han enviado.
- Los datagramas son bloques de información del tipo lanzar y olvidar, es decir, no hay la seguridad de que el paquete llegue o no al destino. Para la mayoría de los programas que utilicen la red, el usar un flujo TCP en vez de un datagrama UDP es más sencillo y hay menos posibilidades de tener problemas. Sin embargo, cuando se requiere un rendimiento óptimo, y está justificado el tiempo adicional que supone realizar la verificación de los datos, la comunicación a través de sockets TCP es un mecanismo realmente útil.
- En resumen, TCP parece más indicado para la implementación de servicios de red como un control remoto (rlogin, telnet) y transmisión de ficheros (ftp); que necesitan transmitir datos de longitud indefinida. UDP es menos complejo y tiene una menor sobrecarga sobre la conexión; esto hace que sea el indicado en la implementación de aplicaciones cliente/servidor en sistemas distribuidos montados sobre redes de área local.

Programación de Sockets

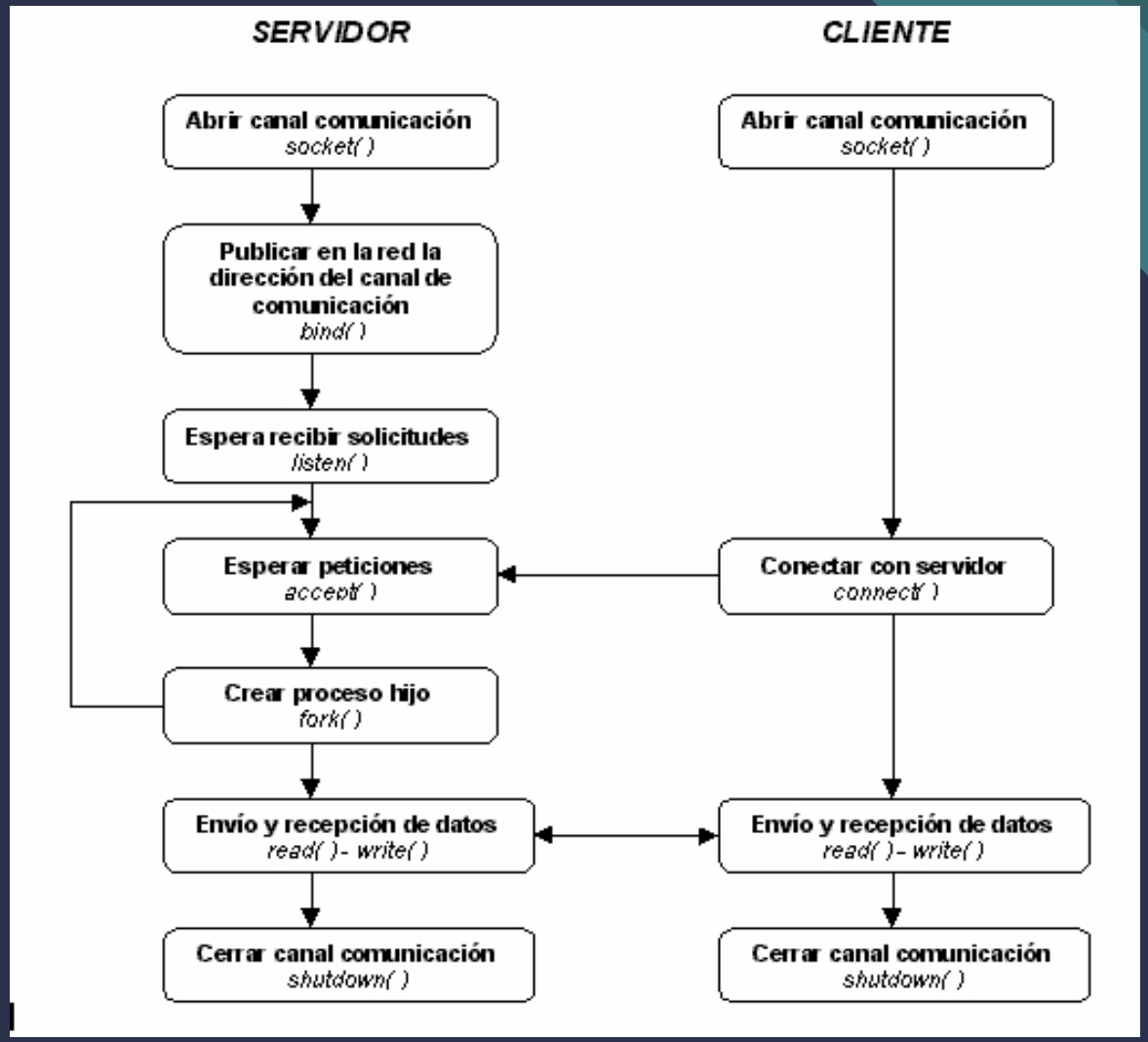
Aplicando JAVA a nuestras comunicaciones

Programando Sockets

- Como ya se menciono, los sockets extienden el concepto de descriptores de archivos para usarlos en conexiones remotas, en otras palabras, una conexión mediante sockets es idéntica a una comunicación mediante un pipe bidireccional, lo único que cambia es la forma de “abrir” el descriptor de archivo.
- Como se ha mencionado, la comunicación con sockets hace uso de una serie de primitivas, entre las que destacan socket para establecer el punto de comunicación, connect para conectarse a una maquina remota en un determinado puerto que esté disponible, bind para escribir en él y publicar información, read para leer de él, shutdown o close para desconectarse, entre otras.
- Normalmente, un servidor se ejecuta sobre una computadora específica y tiene un socket que responde en un puerto específico. El servidor únicamente espera, escuchando a través del socket a que un cliente haga una petición.
- En el lado del cliente: el cliente conoce el nombre de host de la máquina en la cual el servidor se encuentra ejecutando y el número de puerto en el cual el servidor está conectado. Para realizar una petición de conexión, el cliente intenta encontrar al servidor en la máquina servidora en el puerto especificado.

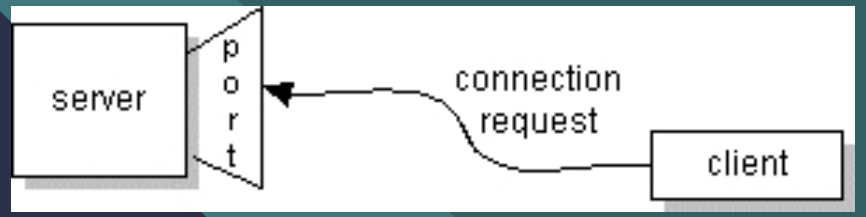
socket	crea un descriptor para usarlo en Tx sobre redes. Toma como parámetro la familia de protocolos, y el tipo de servicio (stream o datagram en en el caso de TCP/IP).
connect	establece una conexión activa, recibe como parámetro la dirección y puerto de destino.
write	generalmente copia los datos a un buffer y los envía a medida que puede. Si los buffers del S.O. están llenos, se bloquea.
read	lee de la conexión, se bloquea si no hay datos, o entrega a lo más length datos (length es un parámetro a la función). En UDP, si hay más datos que length en el datagrama, el resto se pierde, ya que no tiene sentido hacer otro read si no existe el concepto de conexión.
bind	especifica dirección (número IP + puerto local) al cual se asocia el socket.
listen	pone el socket en modo pasivo, y setea el número máximo de conexiones que se encolarán (cuando llegan conexiones simultáneas).
close	termina la conexión y libera el socket. Si es un socket compartido, `ref_count -;`
shutdown	Termina la conexión TCP/IP en una o ambas direcciones.
getpeername	retorna dirección remota del socket.
getsockopt	ver opciones del socket.
setsockopt	cambiar opciones del socket.

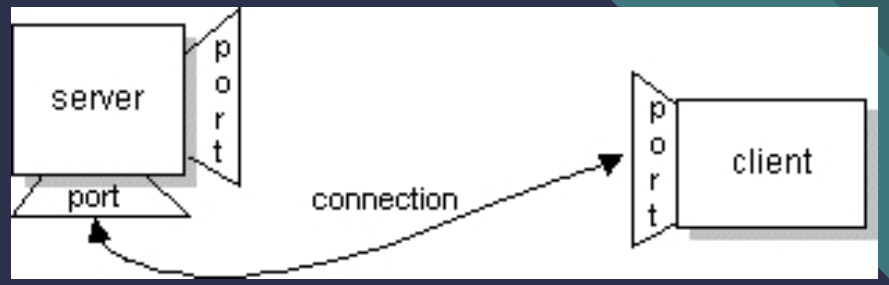
Programando Sockets



Programando Sockets

- Como se puede observar en la figura, un sistema de comunicación necesita de dos entidades bien diferenciadas: el Servidor y el Cliente.
- Cliente realiza petición de conexión al servidor.
- Servidor acepta la solicitud y establece la conexión con el cliente.
- Si todo va bien, el servidor acepta la conexión. Además de aceptar, el servidor obtiene un nuevo socket sobre un puerto diferente. Esto se debe a que necesita un nuevo socket (y, en consecuencia, un número de puerto diferente) para seguir atendiendo al socket original para peticiones de conexión mientras atiende las necesidades del cliente que se conectó.
- Por la parte del cliente, si la conexión es aceptada, un socket se crea de forma satisfactoria y puede usarlo para comunicarse con el servidor. Es importante darse cuenta que el socket en el cliente no está utilizando el número de puerto usado para realizar la petición al servidor. En lugar de éste, el cliente asigna un número de puerto local a la máquina en la cual está siendo ejecutado. Ahora el cliente y el servidor pueden comunicarse escribiendo o leyendo en o desde sus respectivos sockets.



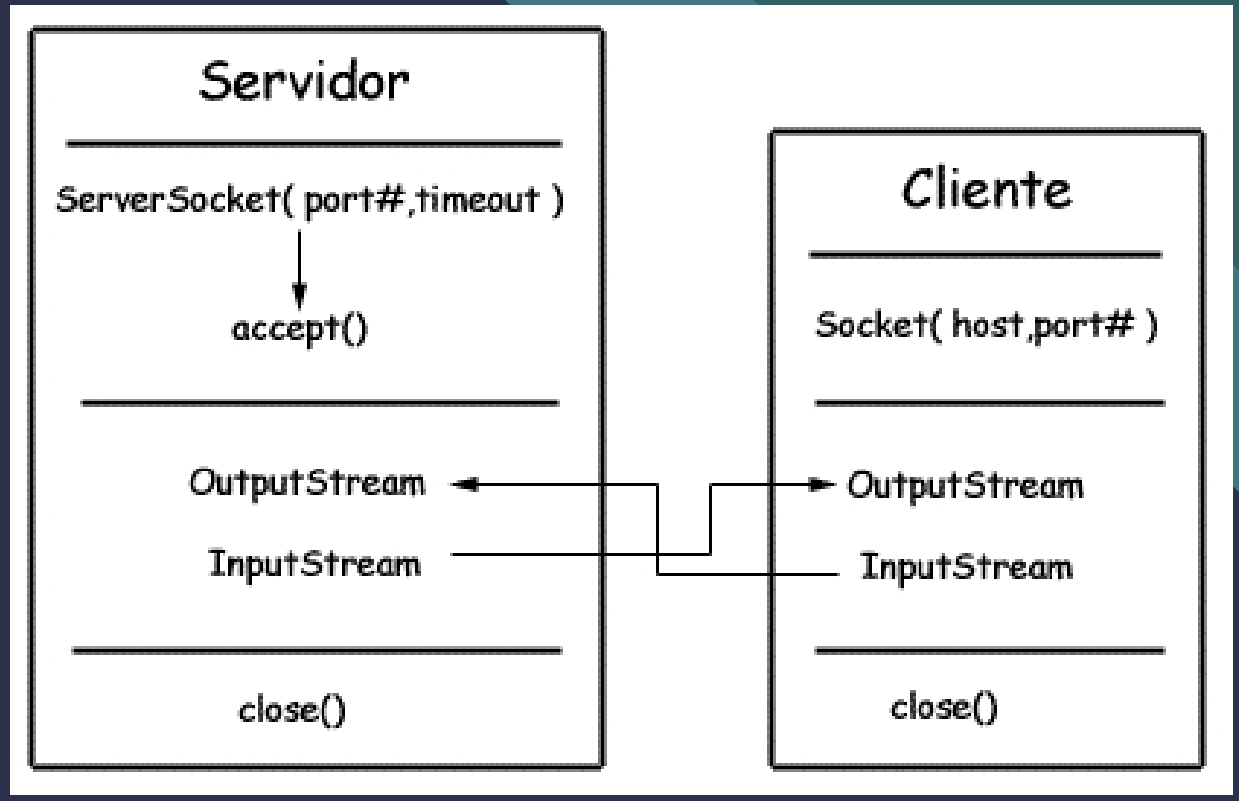


Programando Sockets. Clases

- La programación utilizando sockets involucra principalmente a dos clases: **Socket y DatagramSocket**, a la que se incorpora una tercera no tan empleada, ServerSocket, que solamente se utiliza para implementar servidores, mientras que las dos primeras se pueden usar para crear tanto clientes como servidores, representando comunicaciones TCP la primera y comunicaciones UDP la segunda.
- El paquete java.net de la plataforma Java proporciona una clase Socket, la cual implementa una de las partes de la comunicación bidireccional entre un programa Java y otro programa en la red.
- La clase Socket se sitúa en la parte más alta de una implementación dependiente de la plataforma, ocultando los detalles de cualquier sistema particular al programa Java. Usando la clase java.net.Socket en lugar de utilizar código nativo de la plataforma, los programas Java pueden comunicarse a través de la red de una forma totalmente independiente de la plataforma.
- De forma adicional, java.net incluye la clase ServerSocket, la cual implementa un socket el cual los servidores pueden utilizar para escuchar y aceptar peticiones de conexión de clientes. Por otra parte, si intentamos conectar a través de la Web, la clase URL y clases relacionadas (URLConnection, URLEncoder) son probablemente más apropiadas que las clases de sockets.
- Pero, de hecho, las clases URL no son más que una conexión a un nivel más alto a la web y utilizan como parte de su implementación interna a los sockets.

Programando Sockets. Modelo de Comunicaciones

- El modelo de comunicaciones más simple es el siguiente:
- El servidor establece un puerto y espera durante un cierto tiempo (timeout) a que el cliente establezca la conexión. Cuando el cliente solicite una conexión, el servidor abrirá la conexión socket con el método accept().
- El cliente establece una conexión con la máquina host a través del puerto que se designe en el parámetro respectivo.
- El cliente y el servidor se comunican con manejadores InputStream y OutputStream.



Programando Sockets. Apertura de sockets

- Si estamos implementando un Cliente, el socket se abre de la forma:
- Donde maquina es el nombre de la máquina en donde estamos intentando abrir la conexión y numeroPuerto es el puerto (un número) del servidor que está corriendo sobre el cual nos queremos conectar. Cuando se selecciona un número de puerto, se debe tener en cuenta que los puertos en el rango 0-1023 están reservados para usuarios con muchos privilegios (superusuarios o root). Estos puertos son los que utilizan los servicios estándar del sistema como email, ftp o http. Para las aplicaciones que se desarrollen, asegurarse de seleccionar un puerto por encima del 1023.

```
Socket miCliente;
try {
miCliente = new
Socket
(
"maquina",numeroPuerto );
}
catch( IOException e )
{
System.out.println( e );
}
```

Programando Sockets. Apertura de sockets

- A la hora de la implementación de un servidor también necesitamos crear un objeto socket desde el ServerSocket para que esté atento a las conexiones que le puedan realizar clientes potenciales y poder aceptar esas conexiones:

```
Socket socketServicio =
null;
try {
socketServicio =
miServicio.accept();
}
catch( IOException e )
{
System.out.println( e );
}
```

Programando Sockets. Creación de Streams. Entrada

- En la parte Cliente de la aplicación, se puede utilizar la clase DataInputStream para crear un stream de entrada que esté listo a recibir todas las respuestas que el servidor le envíe.
- La clase DataInputStream permite la lectura de líneas de texto y tipos de datos primitivos de Java de un modo altamente portable; dispone de métodos para leer todos esos tipos como: read(), readChar(), readInt(), readDouble() y readLine(). Deberemos utilizar la función que creamos necesaria dependiendo del tipo de dato que esperemos recibir del servidor.
- En el lado del Servidor, también usaremos DataInputStream, pero en este caso para recibir las entradas que se produzcan de los clientes que se hayan conectado.

```
DataInputStream entrada;
try {
entrada = new
DataInputStream
(
socketServicio.getInputStr
eam() );
}
catch( IOException e )
{
System.out.println( e );
}
```

Programando Sockets. Creación de Streams. Salida

- En la parte del Cliente podemos crear un stream de salida para enviar información al socket del servidor utilizando las clases PrintStream o DataOutputStream.
- La clase PrintStream tiene métodos para la representación textual de todos los datos primitivos de Java. Sus métodos write y println() tienen una especial importancia en este aspecto. No obstante, para el envío de información al servidor también podemos utilizar DataOutputStream.
- La clase DataOutputStream permite escribir cualquiera de los tipos primitivos de Java, muchos de sus métodos escriben un tipo de dato primitivo en el stream de salida. De todos esos métodos, el más útil quizás sea writeBytes().
- En el lado del Servidor, podemos utilizar la clase PrintStream para enviar información al cliente.
- Pero también podemos utilizar la clase DataOutputStream como en el caso de envío de información desde el cliente.

```
PrintStream salida;
try {
salida = new
PrintStream
(
socketServicio.getOutputSt
ream() );
}
catch( IOException e )
{
System.out.println( e );
}
```

Programando Sockets. Cierre de sockets

- Siempre deberemos cerrar los canales de entrada y salida que se hayan abierto durante la ejecución de la aplicación. En la parte del cliente y del servidor
- Es importante destacar que el orden de cierre es relevante. Es decir, se deben cerrar primero los streams relacionados con un socket antes que el propio socket, ya que de esta forma evitamos posibles errores de escrituras o lecturas sobre descriptores ya cerrados.

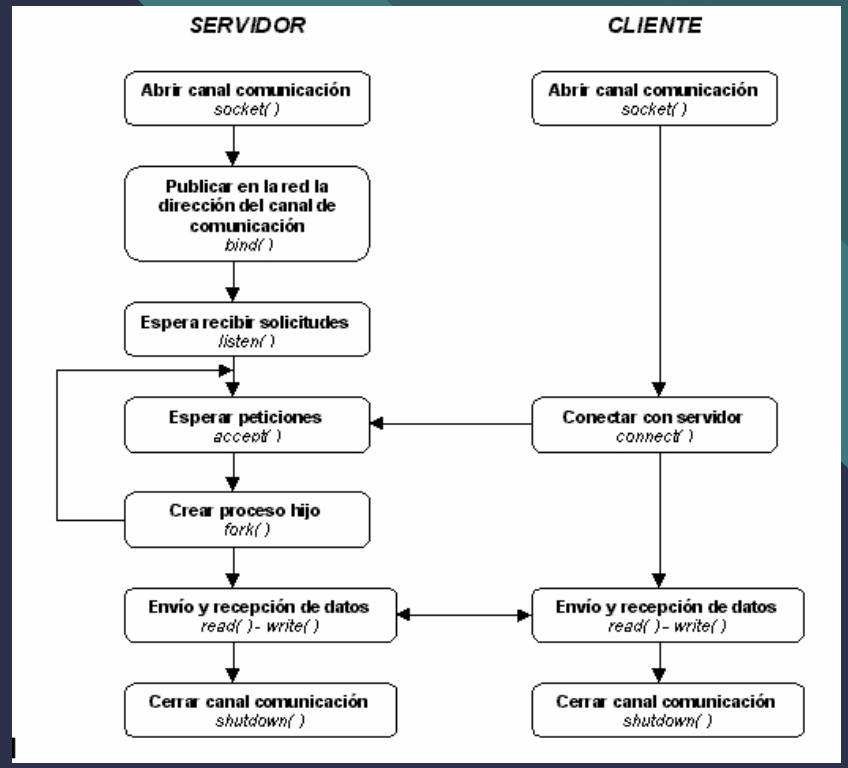
```
try {
salida.close();
entrada.close();
socketServicio.close();
miServicio.close();
}
catch( IOException e )
{
System.out.println( e );
}
```

Aplicación Cliente-Servidor

Arquitectura distribuida

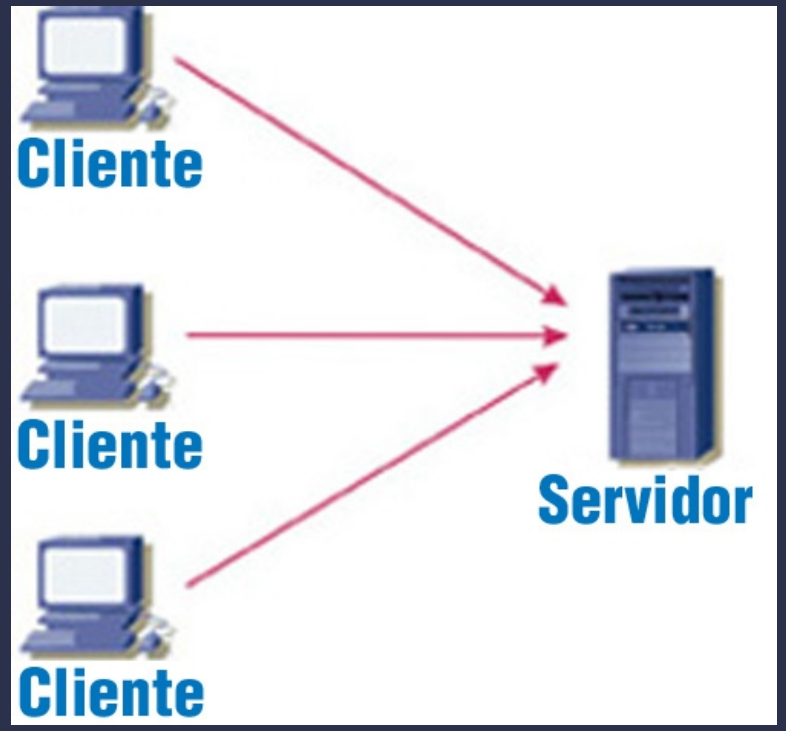
Cliente-Servidor

- Los elementos que componen el modelo son:
- Cliente. Es el proceso que permite interactuar con el usuario, realizar las peticiones, enviarlas al servidor y mostrar los datos al cliente. En definitiva, se comporta como la interfaz (front-end) que utiliza el usuario para interactuar con el servidor. Las funciones que lleva a cabo el proceso cliente se resumen en los siguientes puntos:
  - Interactuar con el usuario.
  - Procesar las peticiones para ver si son válidas y evitar peticiones maliciosas al servidor.
  - Recibir los resultados del servidor.
  - Formatear y mostrar los resultados.
- Servidor. Es el proceso encargado de recibir y procesar las peticiones de los clientes para permitir el acceso a algún recurso (back-end). Las funciones del servidor son:
  - Aceptar las peticiones de los clientes.
  - Procesar las peticiones.
  - Formatear y enviar el resultado a los clientes.
  - Procesar la lógica de la aplicación y realizar validaciones de datos.
  - Asegurar la consistencia de la información.
  - Evitar que las peticiones de los clientes interfieran entre sí.
  - Mantener la seguridad del sistema.



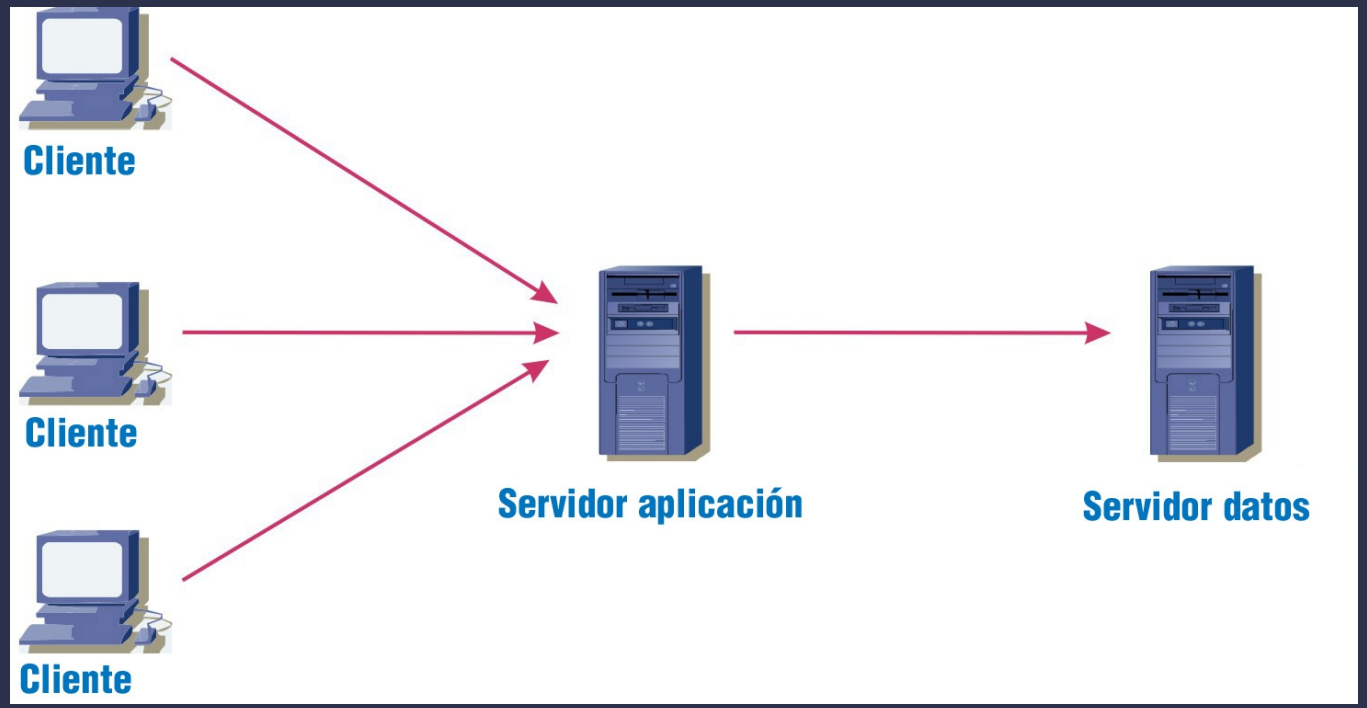
Cliente-Servidor. Modelos

- La principal forma de clasificar los modelos Cliente/Servidor es a partir del número de capas (tiers) que tiene la infraestructura del sistema. De ésta forma podemos tener los siguientes modelos:
  - 1 capa (1-tier). El proceso cliente/servidor se encuentra en el mismo equipo y realmente no se considera un modelo cliente/servidor ya que no se realizan comunicaciones por la red.
  - 2 capas (2-tiers). Es el modelo tradicional en el que existe un servidor y unos clientes bien diferenciados. El principal problema de éste modelo es que no permite escalabilidad del sistema y puede sobrecargarse con un número alto de peticiones por parte de los clientes.



Cliente-Servidor. Modelos

- 3 capas (3-tiers). Para mejorar el rendimiento del sistema en el modelo de dos capas se añade una nueva capa de servidores. En este caso se dispone de:
  - Servidor de aplicación. Es el encargado de interactuar con los diferentes clientes y enviar las peticiones de procesamiento al servidor de datos.
  - Servidor de datos. Recibe las peticiones del servidor de aplicación, las procesa y le devuelve su resultado al servidor de aplicación para que éste los envíe al cliente. Para mejorar el rendimiento del sistema, es posible añadir los servidores de datos que sean necesarios.
- n capas (n-tiers). A partir del modelo anterior, se pueden añadir capas adiciones de servidores con el objetivo de separar la funcionalidad de cada servidor y de mejorar el rendimiento del sistema.



Cliente-Servidor. Optimización

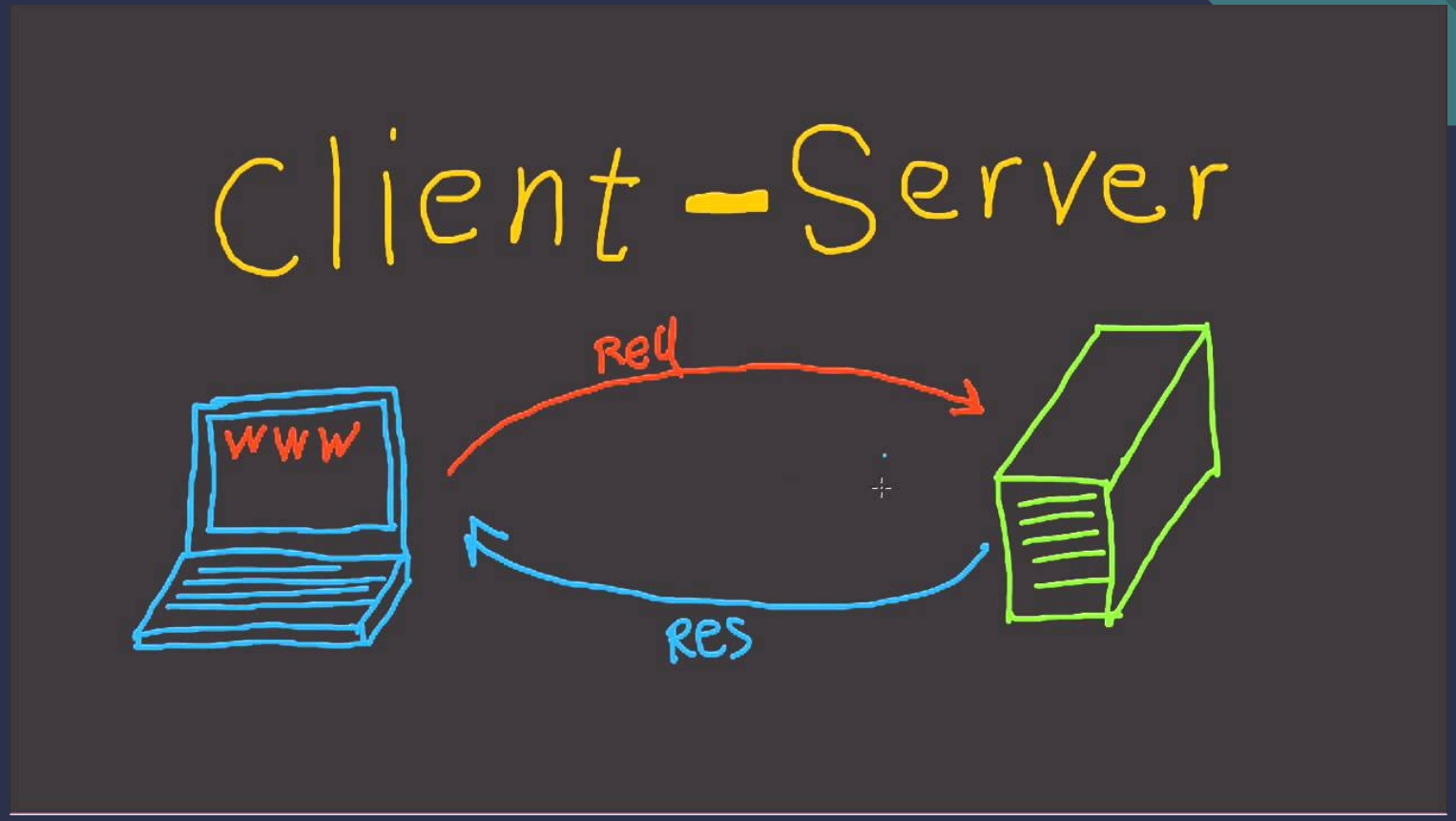
- A la hora de utilizar los sockets es muy importante optimizar su funcionamiento y garantizar la seguridad del sistema. Como la información reside en el servidor y existen múltiples clientes que realizan peticiones es totalmente indispensable permitir que la aplicación cliente/servidor cuente con las siguientes características que veremos más adelante:
- **Atender múltiples peticiones simultáneamente**. El servidor debe permitir el acceso de forma simultánea al servidor para acceder a los recursos o servicios que éste ofrece. Para ello, en vez de ejecutar todo el código del servidor de forma secuencial, vamos a tener un bucle while para que cada vez que se realice la conexión de un cliente se cree una hebra de ejecución (thread) que será la encargada de atender al cliente. De ésta forma, tendremos tantas hebras de ejecución como clientes se conecten a nuestro servidor de forma simultánea.
  - La función public Servidor permite inicializar los valores iniciales que recibe la hebra.
  - La función run() es la encargada de realizar las tareas de la hebra.
  - Para iniciar la hebra se crea el objeto Servidor y se inicia.
- **Seguridad.** Para asegurar el sistema, como mínimo, el servidor debe ser capaz de evitar la pérdida de información, filtrar las peticiones de los clientes para asegurar que éstas están bien formadas y llevar un control sobre las diferentes transacciones de los clientes.
- Por último, es necesario dotar a nuestro sistema de mecanismos para **monitorizar los tiempos de respuesta de los clientes** para ver el comportamiento del sistema.

Servicios de Sistema

Servicios de Sistema

Servicios

- Un servicio es una apliación del sistema encargada de realizar una serie de acciones según las peticiones que recibe. Estas aplicaciones tienen la particularidad que se encuentran corriendo en segundo plano.



Servicios en Windows

- Haremos uso de NSSM: http://nssm.cc/
- Debemos indicar el jar de nuestra app:

  pushd <path-to-jar>

  nssm.exe install "<service-name>" "<path-to-java.exe>" "-jar <name-of-jar>"

  nssm.exe set "<service-name>" AppDirectory "<path-to-jar>"

- Tambiñen podemos usar winsw: https://github.com/winsw/winsw. Cuyo fichero de ejecución es:

```
<service>
        <id>myapp</id>
        <name>myapp</name>
        <description>This service runs myapp project.</description>
        <executable>java</executable>
        <arguments>-jar "myapp.jar"</arguments>
        <logmode>rotate</logmode>
</service>
```

Servicios en Linux

- Creamos el servicio:
  - sudo vim /etc/systemd/system/my-webapp.service
- Copiamos el contenido

```
[Unit]
Description=My App Java Service
[Service]
User=bhupesh
# The configuration file application.properties should be here:
#change this to your workspace
WorkingDirectory=/home/bhupesh/GitHub
#path to executable.
#executable is a bash script which calls jar fileExecStart=/home/bhupesh/GitHub/myapp
SuccessExitStatus=143
TimeoutStopSec=10
Restart=on-failure
RestartSec=5
[Install]
WantedBy=multi-user.target
```

Servicios en Linux

- Creamos el script bash para llamar a la apliación

  #!/bin/sh

  sudo /usr/bin/java -jar myapp.jar server config.yml

  sudo chmod u+x myapp

- Iniciamos el servicio
  - sudo systemctl daemon-reload
  - sudo systemctl enable myapp.service
  - sudo systemctl start myapp
  - sudo systemctl status myapp
- Si queremos iniciarlo desde el inicio
  - sudo journalctl --unit=myapp
- Parar el servicio
  - sudo systemctl stop myapp

Servicios en Docker

- Creamos el Dockerfile

  FROM java:11

  WORKDIR /

  ADD HelloWorld.jar HelloWorld.jar

  EXPOSE 8080

  CMD java - jar HelloWorld.jar

- Construímos la imagen

  docker build -t helloworld

- Si la queremos la subimos al registro añadiéndole una tag

  docker pull /hello-world

  docker tag 4b795844c7ab /hello-world

  docker push /hello-world:latest

- Si no una vez subida o no (sola la imagen) la ejecutamos
  - docker run /hello-world

"La mejor forma de predecir el futuro es implementarlo"

-- David Heinemeier Hansson

Recursos

- Twitter: https://twitter.com/joseluisgonsan
- GitHub: https://github.com/joseluisgs
- Web: https://joseluisgs.github.io
- Discord: https://discord.gg/kyhx7kGN
- Aula Virtual: https://aulavirtual33.educa.madrid.org/ies.luisvives.leganes/course/view.php?id=248




Gracias

José Luis González Sánchez