Capítulo 3: Comunicación entre procesos

Sistemas Distribuidos

Universidad Nacional de Asunción
Facultad Politécnica
Ingeniería Informática

Prof: Fernando Mancía

Comunicación entre procesos

Protocolos para la comunicación entre procesos en un sistema distribuido.

Comunicación entre procesos en Internet a través de datagramas y comunicación de flujo. API para el manejo de estos modelos de comunicación, junto con una discusión de sus modelos de fallo.

Protocolos para la representación de colecciones de objetos de datos en mensajes y de referencias a objetos remotos.

Primitivas de comunicación entre procesos que apoyan la comunicación punto a punto, sin embargo es igualmente útil poder enviar un mensaje de un remitente a un grupo de receptores. **multicast**, que incluye **multidifusión IP.**

Interfaz de paso de mensajes (MPI), estándar desarrollado para proporcionar una API para un conjunto de operaciones de paso de mensajes con variantes síncronas y asíncronas.

Capas de Middleware

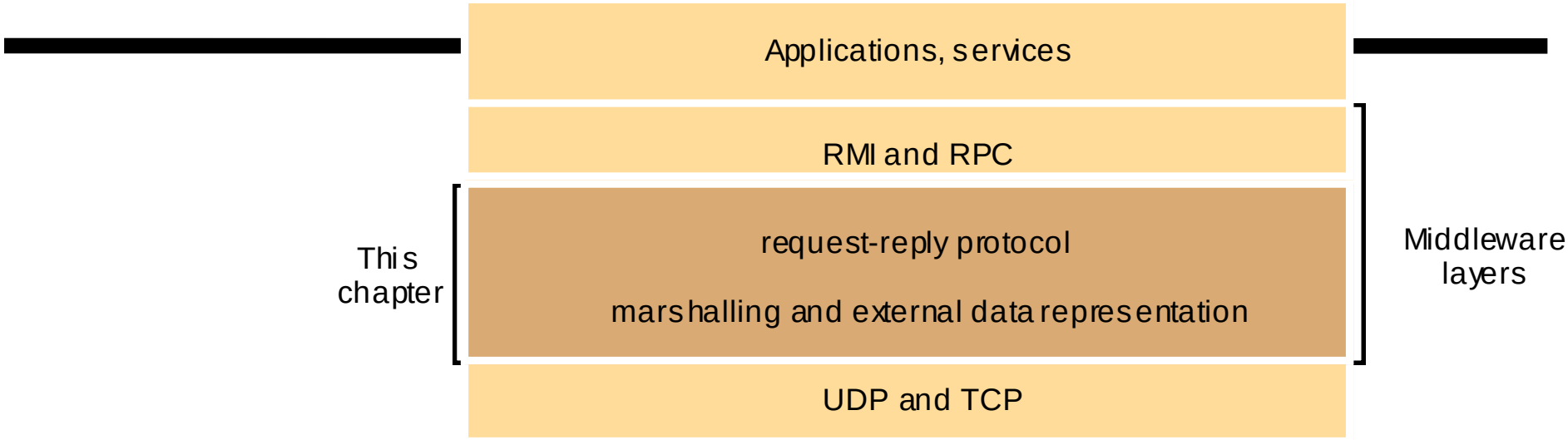
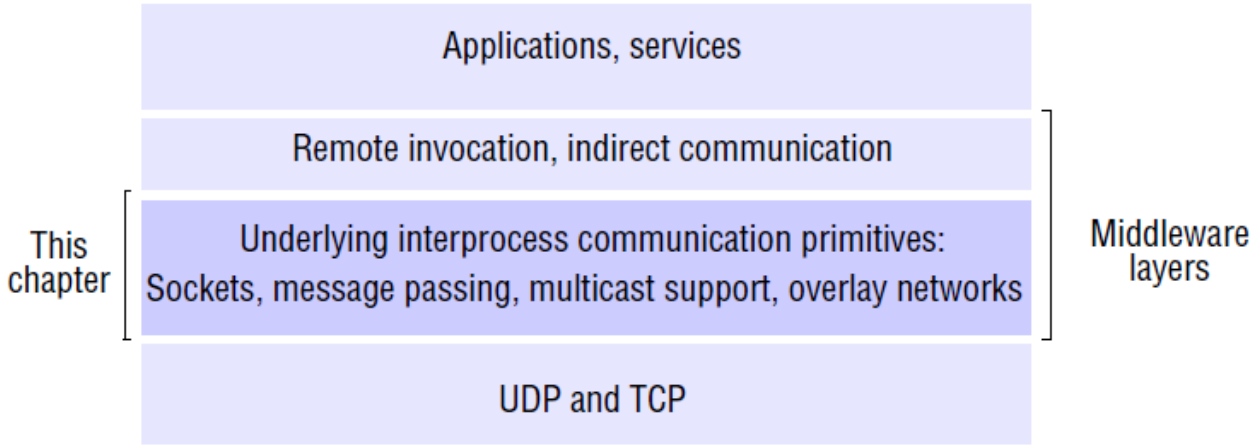


Figure 4.1 Middleware layers



Características de la comunicación entre procesos

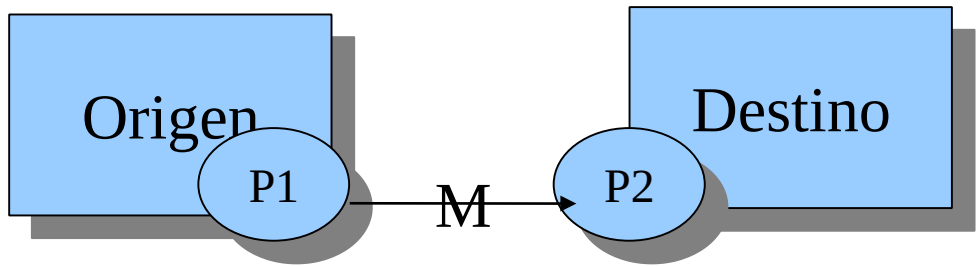
Paso de mensajes entre un par de procesos se basa en dos operaciones:

- *envía*
- *recibe*

definidas en función del destino y del mensaje.

Para que un proceso se pueda comunicar con otro, ***el proceso*** envía un mensaje (una secuencia de bytes) a un destino ***y otro proceso*** en el destino recibe el mensaje.

Esta actividad implica la comunicación de datos desde el proceso emisor al proceso receptor y puede implicar además la sincronización de los dos procesos.



Características de la comunicación entre procesos

Comunicación síncrona y asíncrona. A cada destino de mensajes se asocia una cola.

- Procesos emisores producen mensajes que serán añadidos a las colas remotas.
- Procesos receptores eliminarán mensajes de las colas locales.

La comunicación entre los procesos emisor y receptor puede ser síncrona o asíncrona. En la forma **SÍNCRONA**, los procesos receptor y emisor se sincronizan con cada mensaje. En este caso, tanto ***envía*** como ***recibe*** son operaciones ***bloqueantes***.

P.emisor, ***Envia*** *– bloquea hasta que se produce el corresp recibe.*
P.receptor, invoca un ***Recibe***, se bloquea hasta que llega un mensaje.

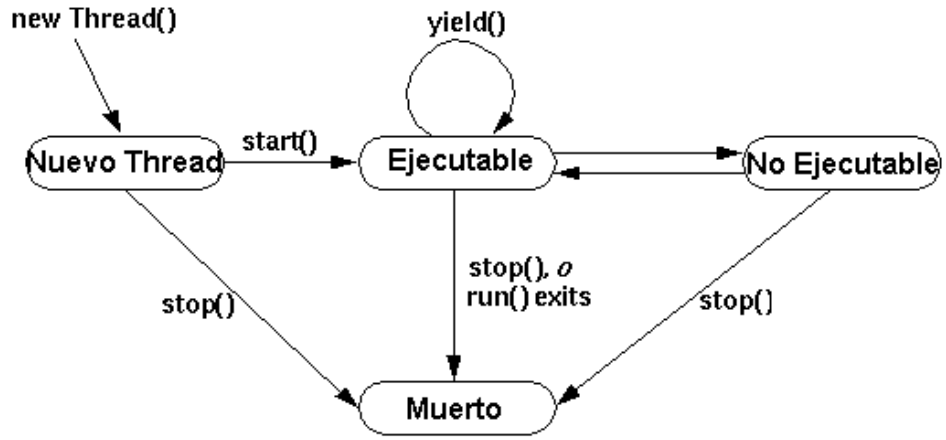
Características de la comunicación entre procesos

Comunicación *ASÍNCRONA*, la operación ***ENVÍA*** es *NO BLOQUEANTE, el p.* emisor puede continuar tan pronto como el mensaje haya sido copiado al búfer local, y la transmisión del mensaje se lleva a cabo en PARALELO con el proceso emisor. ***RECIBE*** puede tener variantes bloqueantes y no bloqueantes.

En la variante no bloqueante el proceso receptor sigue con su programa después de invocar la operación *recibe,* la cual proporciona un **BUFER** que será llenado en un segundo plano, pero el proceso **DEBE SER INFORMADO** por separado de que su búfer ha sido llenado, ya sea por el método de encuesta o mediante una interrupción.

Características de la comunicación entre procesos

En un entorno como Java, que soporta múltiples hilos en un único proceso, el *recibe* bloqueante tiene pocas desventajas, ya que puede ser invocado por un hilo mientras que el resto de hilos permanecen activos, y la simplicidad de sincronizar los hilos receptores con el mensaje, entrante es una ventaja substancial.



La comunicación no bloqueante parece ser más eficiente, implica una complejidad extra en el proceso receptor asociada con la necesidad de capturar mensaje entrante fuera de su flujo de control. Por estas razones, los sistemas actuales no proporcionan la forma no bloqueante de *recibe.*

Características de la comunicación entre procesos

Destinos de los mensajes. En los protocolos Internet, los mensajes son enviados a direcciones construidas (**Direccion_Internet**, **Puerto_local**).

Un puerto local es el destino de un mensaje dentro de un computador, especificado como un nro entero. Tiene exactamente un receptor pero puede tener muchos emisores.

Los procesos pueden utilizar múltiples puertos desde los que recibir mensajes. Cualquier proceso que conozca un número de puerto apropiado puede enviarle un mensaje.

Generalmente, los servidores hacen públicos sus números de puerto para que sean utilizados por los clientes.

Si el cliente utiliza una dirección Internet fija para referirse a un servicio, entonces ese servicio debe ejecutarse siempre en el mismo computador para que la dirección se considere válida..

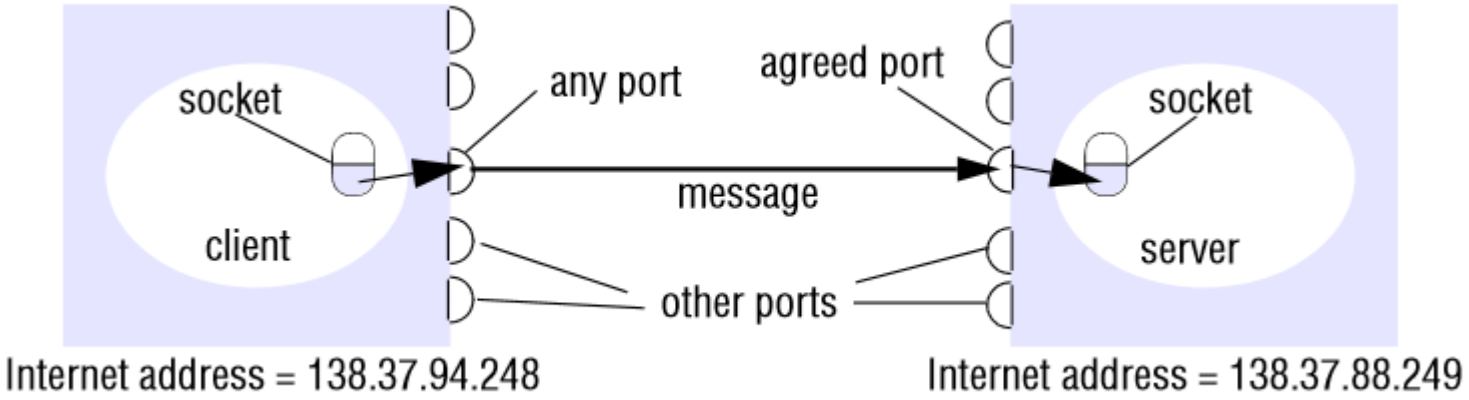
Sockets

Ambas formas de comunicación (UDP y TCP) utilizan la abstracción de *sockets,* que proporciona los **puntos extremos de la comunicación entre procesos**.

Los sockets (conectores) se originan en UNIX BSD aunque están presentes en la mayoría de las versiones de UNIX, incluido Linux y también Windows NT y Macintosh OS.

La comunicación entre procesos consiste en la transmisión de un mensaje entre un conector de un proceso y un conector de otro proceso.

Figura 4.2
Sockets y puertos



Sockets

Los procesos pueden usar el mismo socket para enviar y recibir mensajes.

Cada computadora tiene (2^16) de números de puertos posibles para ser usados por los procesos locales para recibir mensajes.

Cualquier proceso puede hacer uso de múltiples puertos para recibir mensajes, pero un proceso no puede compartir puertos con otros procesos en la misma computadora. Los procesos que usan IP multicast son una excepción porque comparten puertos.

Sin embargo, cualquier número de procesos puede enviar mensajes al mismo puerto.

Cada socket está asociado con un protocolo particular, ya sea UDP o TCP.

Sockets

RECEPTOR:

- En procesos receptores de mensajes, su conector debe estar asociado a un puerto local y a una de las direcciones del computador donde se ejecuta.
- Los mensajes enviados a una dirección de Internet y a un número de puerto concretos, **sólo pueden ser recibidos** por el proceso cuyo conector esté asociado con esa dirección y con ese puerto.

```
server {
    listen       80;
    server_name  example.org  www.example.org;
    ...
}

server {
    listen       80;
    server_name  *.example.org;
    ...
}
```

conf.d/dominio-example.conf
Variable listen en archivo de configuración de servidor web nginx.

```
listen_addresses = 'localhost'
```

```
listen_addresses = '*'
```

postgresql.conf
Variable listen en archivo de configuración de servidor de base de datos Postgresql.

API de Java para las direcciones Internet. Como los paquetes IP que subyacen a TCP y UDP se envian a direcciones Internet, Java proporciona una clase, *InetAddress,* que representa las direcciones Internet.

Los usuarios de esta clase se refieren a los computadores por sus nombres de host en el Servicio de Nombres de Dominio *(Domain Name Service,* DNS).

Por ejemplo, se pueden crear instancias de *InetAddress* que contienen direcciones de Internet invocando a un método estático de *InetAddress* con un nombre DNS de host como argumento.

El método utiliza DNS para conseguir la correspondiente dirección Internet.

Por ejemplo, para conseguir un objeto que represente la dirección Internet de un host cuyo nombre DNS es *bruno.dcs.qmw.ac.uk,* utilice:

InetAddress unComputador=InetAddress.getByName(bruno.dcs.qmw.ac.uk»);

Comunicación de datagramas UDP

Un datagrama enviado por UDP se transmite desde un proceso emisor a un proceso receptor "**sin acuse de recibo"** ni **"reintentos"**.

Si algo falla, el mensaje puede no llegar a su destino.

Se transmite un datagrama, entre procesos, cuando uno lo envía, y el otro lo recibe.

Cualquier proceso que necesite enviar o recibir mensajes debe crear, primero, un **conector asociado a una dirección Internet y a un puerto** local.

Un servidor enlazará su conector a un puerto de servidor.
Un cliente ligará su conector a cualquier puerto local libre.
El método **Recibe** devolverá, **M:**mensaje, **D:**dirección emisor; **P:**puerto emisor, permitiendo al receptor enviar la correspondiente respuesta.

Comunicación de datagramas UDP

Aspectos referentes a la comunicación de datagramas:

Tamaño del mensaje: el receptor debe especificar una cadena de bytes de un tamaño concreto para el mensaje recibido.

Si el mensaje es demasiado grande para dicha cadena, será truncado a la llegada. La capa subyacente IP permite paquetes de hasta 2^{16} bytes, incluyendo tanto las cabeceras como el mensaje.

Sin embargo, la mayoría de los entornos imponen una restricción a su talla de 8 kilobytes.

Comunicación de datagramas UDP

Bloqueo: comunicación de datagramas UDP utiliza operaciones de envío, *envía,* **no bloqueantes** y recepciones, *recibe,* **bloqueantes**.

P.origen: La operación ***envía*** devuelve el control cuando ha dirigido el mensaje a las capas inferiores UDP e IP, que son las responsables de su entrega en el destino.

P.destino: A la llegada, el mensaje será colocado **en una cola del** conector que está enlazado con el puerto de destino. El mensaje podrá obtenerse de la cola de recepción mediante una invocación pendiente o futura del método *recibe* sobre ese conector. Si no existe ningún proceso ligado al conector destino, los mensajes serán descartados.

El método *recibe* produce un bloqueo hasta que se reciba un datagrama, a menos que se haya establecido un tiempo límite *(time out)* asociado al conector.

Comunicación de datagramas UDP

Tiempo límite de espera: *recibe* con bloqueo indefinido, adecuado para servidores que están esperando recibir peticiones desde sus clientes.
En algunos programas no resulta apropiado la espera indefinida, sobretodo en aquellas situaciones en las que el potencial emisor puede haber caído o se haya podido perder el mensaje. Para esto, se pueden fijar tiempos límites de espera *(timeouts)* en los conectores.

Recibe de cualquiera: el método *recibe* no especifica el origen de los mensajes. El método *recibe* devuelve la dirección Internet y el puerto del emisor, permitiendo al receptor comprobar de dónde viene el mensaje. ***Es posible vincular un socket y una dirección Internet remotas particulares, en cuyo caso el conector sólo podrá recibir y enviar mensajes con esa dirección.***

Comunicación de datagramas UDP

Modelo de fallo. El modelo de fallo puede utilizarse para proponer un modelo de fallo para los datagramas UDP, que padece de las siguientes debilidades:

Fallos de omisión: los mensajes pueden desecharse ocasionalmente, error detectado por comprobación o porque no queda espacio en el búfer origen o destino.

Ordenación: algunas veces, los mensajes se entregan en desorden con respecto a su orden de emisión.

Las aplicaciones que utilizan datagramas UDP **dependen de sus propias comprobaciones** para conseguir la calidad que necesitan respecto a la fiabilidad de la comunicación. Puede construirse un servicio de entrega fiable a partir de uno que adolece de fallos de omisión mediante la utilización de acuses de recibo

Comunicación de datagramas UDP

Utilización de UDP. Para algunas aplicaciones, resulta aceptable utilizar un servicio que sea susceptible de sufrir fallos de omisión ocasionales.

Por ejemplo, el Servicio de Nombres de Dominio en Internet *(Domain Name Service,* DNS) está implementado sobre UDP.

Los datagramas UDP son, en algunas ocasiones, una elección atractiva porque no padecen las sobrecargas asociadas a la entrega de mensajes garantizada. Existen tres fuentes principales para esa sobrecarga:

1. La necesidad de almacenar información de estado en el origen y en el destino.
2. La transmisión de mensajes extra.
3. La latencia para el emisor.

Comunicación de datagramas UDP

API Java para datagramas UDP. La API Java proporciona una comunicación de datagramas por medio de dos clases: *DatagramPacket* y *DatagramSocket.*

DatagramPacket: esta clase proporciona un constructor que crea una instancia compuesta por:

una cadena de bytes que almacena el mensaje,
la longitud del mensaje
la dirección Internet
el número de puerto local

Paquete del datagrama

cadena de bytes conteniendo el mensaje, longitud del mensaje. dirección Internet, número de puerto

Las instancias de *DatagramPacket* podrán ser transmitidas entre procesos cuando uno las *envía,* y el otro las *recibe* .

Comunicación de datagramas UDP

La clase *DatagramSocket* proporciona varios métodos que incluyen los siguientes:

send y receive: estos métodos sirven para transmitir datagramas entre un par de conectores.

send **(**una instancia de *DatagramPacket,* conteniendo el mensaje y su destino)

receive(*DatagramPacket* vacío en el que colocar el mensaje, su longitud y su origen)

Ambos pueden lanzar una excepción *IOException.*

setSoTimeout: permite establecer un tiempo de espera límite. Cuando se fija un límite, el método *receive* se bloquea durante el tiempo fijado y después lanza una excepción *InterruptedIOException.*

connect: utilizado para conectarse a un puerto remoto y dirección Internet concretos, en cuyo caso el conector sólo podrá enviar y recibir mensajes de esa dirección.

Comunicación de streams TCP

La API para el protocolo TCP, que es originaria de UNIX BSD 4.x, proporciona la abstracción de un flujo de bytes *(stream)* en el que pueden escribirse y desde el que pueden leerse datos. La abstracción de *stream* oculta las siguientes características de la red:

- *Tamaño de los mensajes:* la aplicación puede elegir la cantidad de datos que quiere escribir o leer del stream.

- El conjunto de datos puede ser muy pequeño o muy grande. La implementación del flujo TCP subyacente decide cuántos datos recoge antes de transmitirlos como uno o más paquetes IP.

- En el destino, los datos son proporcionados a la aplicación según los va solicitando. Las aplicaciones pueden forzar, si fuera necesario, que los datos sean enviados de forma inmediata.

Comunicación de streams TCP

- *Mensajes perdidos:* el protocolo TCP utiliza un esquema de acuse de recibo de los mensajes.

Como un ejemplo de un esquema simple (no utilizado por TCP), el extremo emisor almacena un registro de cada paquete IP enviado y el extremo receptor acusa el recibo de todos los paquetes IP que le llegan.

Si el emisor no recibe dicho acuse de recibo dentro de un plazo de tiempo fijado, volverá a transmitir el mensaje.

- *Control del flujo:* el protocolo TCP intenta ajustar las velocidades de los procesos que leen y escriben en un stream.

Si el escritor es demasiado rápido para el lector, entonces será bloqueado hasta que el lector haya consumido una cantidad suficiente de datos.

- Duplicación y ordenación de los mensajes: a cada paquete IP se le asocia un identificador, que hace posible que el receptor pueda detectar y rechazar mensajes duplicados, o que pueda reordenar los mensajes que lleguen desordenados.

- Destinos de los mensajes: un par de procesos en comunicación establecen una conexión antes de que puedan comunicarse mediante un stream. Una establecida la comunicación, los procesos simplemente leen o escriben en el stream sin tener que preocuparse de las direcciones Internet ni de los números de puerto. La conexión implica una petición de conexión, *connect,* desde el cliente al servidor, seguida de una aceptación,accept, desde el servidor al cliente antes de que cualquier comunicación pueda tener lugar.

Esto puede suponer una sobrecarga considerable para una única petición y una única respuesta.

Comunicación de streams TCP

El API para la comunicación por streams supone que en el momento de establecer una conexión **uno de ellos es cliente** y el otro **es 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.

- Concordancia de ítems de datos: los dos procesos que se comunican necesitan estar de acuerdo en el ***tipo de datos*** transmitidos por el stream. Por ejemplo, si un proceso escribe un int seguido de un *double,* el proceso receptor debe interpretarlo como un *int* seguido de un double. Cuando un par de procesos no coopera correctamente en el uso del stream, el proceso lector puede encontrarse con problemas cuando interprete los datos o puede bloquearse debido a que se encuentre una cantidad insuficiente de datos en el stream.

Comunicación de streams TCP

***Bloqueo**:* los datos escritos en un stream se almacenan en un búfer en el conector destino. Cuando un proceso intenta leer datos de un canal de entrada:

- extrae los datos de la cola o
- se bloquea hasta que existan datos disponibles.

El proceso que escribe los datos en el stream resultará bloqueado por el mecanismo de control de stream de TCP si el conector del otro lado intenta almacenar en la cola de entrada más información de la permitida.

Hilos: cuando un servidor acepta una conexión, generalmente crea un nuevo hilo con el que comunicarse con el nuevo cliente. La **ventaja** de utilizar un hilo separado para cada cliente es que el servidor puede bloquearse a la espera de entradas sin afectar a los otros clientes. En un entorno en el cual no se disponga de hilos, una alternativa es comprobar si existen datos accesibles en el stream antes de intentar leerlos.

Comunicación de streams TCP

Modelo de fallo. los streams TCP utilizan:
-suma de comprobación para detectar y rechazar los paquetes corruptos
-número de secuencia para detectar y eliminar los paquetes duplicados.

Con respecto a la propiedad de validez, los streamsTCP utilizan:
- ***timeouts*** y
- **retransmisión** de los paquetes perdidos.

Se tienen garantizo la entrega incluso cuando alguno de los paquetes subyacentes se haya perdido.

Conexion Rota:
Si la pérdida de paquetes sobrepasa un cierto límite, o la red que conecta un par de procesos en comunicación está severamente congestionada, el software TCP responsable de enviar los mensajes no recibirá acuses de recibo de los paquetes enviados y después de un tiempo declarará rota la conexión.

Comunicación de streams TCP

Cuando una conexión está rota, se notificará al proceso que la utiliza siempre que intente leer o escribir. Esto tiene los siguientes efectos:

. Los procesos que utilizan la conexión no distinguen entre un fallo en la red y un fallo en el proceso que está en el otro extremo de la conexión.

. Los procesos comunicantes no pueden saber si sus mensajes recientes han sido recibidos o no.

Comunicación de streams TCP

Utilización de TCP. Muchos de los servicios utilizados se ejecutan sobre conexiones TCP , con números de puerto reservados. Entre ellos se encuentran los siguientes:

HITP: El protocolo de transferencia de hipertexto se utiliza en comunicación entre un navegador y un servidor web
FTP: El protocolo de transferencia de archivos permite leer los directorios de un computador remoto y transferir archivos entre los computadores de una conexión.
Telnet: La herramienta Telnet proporciona acceso a un terminal en un computador remoto.
SMTP: El protocolo simple de transferencia de correo se utiliza para mandar correos electrónicos entre computadores.

Comunicación de streams TCP

El API Java para los streams TCP . La interfaz Java para los streams TCP está consituida por las clases *ServerSocket* y *Socket.*

ServerSocket*:* está diseñada para ser utilizada por un servidor para crear un conector en el puerto de servidor que escucha las peticiones de conexión de los clientes. Su método *accept* toma una petición *connect* de la cola, o si la cola está vacía, se bloquea hasta que llega una petición. El resultado de ejecutar *accept* es una instancia de *Socket,* un conector que da acceso a streams para comunicarse con el cliente.

Socket: es utilizada por el par de procesos de una conexión. El cliente utiliza un constructor para crear un conector, especificando el nombre DNS de host y el puerto del servidor. Este constructor no sólo crea el conector asociado con el puerto local, sino que también se conecta con el computador remoto especificado en el puerto indicado.

Representación Externa de Datos y Empaquetado

La información almacenada en los programas en ejecución se representa como estructuras de datos (por ejemplo, conjuntos de objetos interconectados) mientras que la información en los mensajes consiste en secuencias de bytes.

Independientemente de la forma de comunicación utilizado, las estructuras de datos deben ser aplanadas (convertido a una secuencia de bytes) antes de la transmisión y reconstruido en destino.

Las datos primitivos en la transmisión de los mensajes pueden tener valores de diferentes tipos, y no todos los computadores almacenan valores, como enteros, en el mismo orden. La representación de números en coma flotante también difiere entre las arquitecturas.

Representación Externa de Datos y Empaquetado

Dos métodos se pueden utilizar para realizar entre dos ordenadores el intercambio de valores binarios de datos:

• Los valores se convierten a un formato convenido externo antes de la transmisión y convertido a la forma local en recepción. Si los dos ordenadores conocen el mismo tipo, la conversión a formato externo puede omitirse.

• Los valores se transmiten en formato del remitente, junto con una indicación del formato utilizado, y el receptor convierte los valores si es necesario.

Representación Externa de Datos y Empaquetado

Alternativas para la representación externa de datos y empaquetado:

- Representación común de datos CORBA: una representación externa de los tipos estructurados y primitivos que se pueden pasar como argumentos y resultados de invocaciones de métodos remotos en CORBA. Puede ser utilizado por una gran variedad de lenguajes de programación.

- Serialización de objetos Java, representación “aplanada” y externa de datos de cualquier objeto o árbol de objetos que pueden necesitar ser transmitidos en un mensaje o almacenado en un disco. Es para uso exclusivo de Java.

Representación Externa de Datos y Empaquetado

- XML (Extensible Markup Language), define un fomato para la representación de datos estructurados. Fue pensado originalmente para los documentos que contienen datos estructurados auto-descriptos.
  - Documentos accesibles en la Web.
  - Representar datos enviados y recibidos en mensajes intercambiados por los clientes y servidores de los Servicios Web.

Otras alternativas:

- Protocol Buffers (Google):
  - **https://developers.google.com/protocol-buffers/**

- JSON
  - **http://www.json.org/**

http://jsonlint.com/ JSONLint The JSON Validator

Representación Externa de Datos y Empaquetado

```
{"menu": {
    "header": "SVG Viewer",
    "items": [
        {"id": "Open"},
        {"id": "OpenNew", "label": "Open New"},
        null,
        {"id": "ZoomIn", "label": "Zoom In"},
        {"id": "ZoomOut", "label": "Zoom Out"},
        {"id": "OriginalView", "label": "Original View"},
        null,
        {"id": "Quality"},
        {"id": "Pause"},
        {"id": "Mute"},
        null,
        {"id": "Find", "label": "Find..."},
        {"id": "FindAgain", "label": "Find Again"},
        {"id": "Copy"},
        {"id": "CopyAgain", "label": "Copy Again"},
        {"id": "CopySVG", "label": "Copy SVG"},
        {"id": "ViewSVG", "label": "View SVG"},
        {"id": "ViewSource", "label": "View Source"},
        {"id": "SaveAs", "label": "Save As"},
        null,
        {"id": "Help"},
        {"id": "About", "label": "About Adobe CVG Viewer..."}
    ]
}}
```

Multidifusión IP. La multidifusión IP se construye sobre el protocolo Internet, IP. Los paquetes IP se dirigen a los computadores; mientras que los puertos pertenecen a niveles TCP y UDP.

La multidifusión IP permite que el emisor transmita un único paquete IP a un conjunto de computadores que forman un grupo de multidifusión.

El emisor no tiene que estar al tanto de las identidades de los receptores individuales y del tamaño del grupo. Los grupos de multidifusión se especifican utilizando las direcciones Internet de la clase D, esto es, una dirección cuyos primeros cuatro bits son 1110 en IPv4.

El convertirse en miembro de un grupo de multidifusión permite al computador recibir los paquetes IP enviados al grupo. La pertenencia a los grupos de multidifusión es dinámica, permitiendose que los computadores se apunten o se borren a un número arbitrario de grupos en cualquier instante.

Los siguientes detalles son específicos de IPv4:

***Routers multidifusión**:* los paquetes IP pueden multidifundirse tanto en la red local como toda Internet. La multidifusión local utiliza la capacidad de multidifusión de la red local, por ejemplo una Ethernet. La multidifusión dirigida a Internet hace uso de las posibilidades de multidifusión de los routers (encaminadores), los cuales reenvían los datagramas únicamente a otros routers de redes con miembros de ese grupo, donde serán multidifundidos a los miembros locales. Para limitar la distancia de propagación de un datagrama de multidifusión, el emisor puede especificar el número de routers que puede cruzar; llamado tiempo de vida *(time to* TTL).

Reserva de direcciones de multidifusión: las direcciones de multidifusión se pueden reservar deforma temporal o permanentemente. Existen grupos permanentes incluso cuando no existe ningún miembro; y sus direcciones son asignadas arbitrariamente por la autoridad de Internet en el rango 224.0.0.1 a 224.0.0.255.

Un participante en multidifusión se apunta a un grupo y envía y recibe datagramas

```
import java.net.*;
import java.io.*;
public class MulticastPeer{
        public static void main(String args[]){
         // args give message contents & destination multicast group (e.g. "228.5.6.7")
        MulticastSocket s =null;
         try {
                InetAddress group = InetAddress.getByName(args[1]);
                s = new MulticastSocket(6789);
                s.joinGroup(group);
                byte [] m = args[0].getBytes();
                DatagramPacket messageOut =
                        new DatagramPacket(m, m.length, group, 6789);
                s.send(messageOut);
```

```
            // get messages from others in group
                    byte[] buffer = new byte[1000];
                    for(int i=0; i< 3; i++) {
                       DatagramPacket messageIn =
                                new DatagramPacket(buffer, buffer.length);
                       s.receive(messageIn);
                       System.out.println("Received:" + new String(messageIn.getData()));
                    }
                    s.leaveGroup(group);
                }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
                }catch (IOException e){System.out.println("IO: " + e.getMessage());}
            }finally {if(s != null) s.close();}
    }
}
```

Virtualización de red

Se ocupa de la construcción de redes virtuales diferentes sobre una red existente tal como el Internet.

Cada red virtual puede ser diseñada para soportar una aplicación distribuida en particular.

Por ejemplo, puede soportar streaming multimedia, como en BBC iPlayer, BoxeeTV [boxee.tv] o Hulu [hulu.com], y coexistir con otro que admita un juego en línea multijugador, ambos corriendo sobre la misma red subyacente.

Esto sugiere una respuesta al dilema:
“una red virtual específica de aplicación puede ser construida sobre una red existente y optimizada para esa aplicación particular, sin cambiar las características de la Red subyacente.”

Virtualización de red: (***overlay networks***)

Una **red de superposición (*overlay networks*)** es una red virtual que consta de nodos y enlaces virtuales, que se encuentran encima de una red subyacente (como una red IP) y ofrece algo que no se proporciona de otra manera:

• un servicio adaptado a las necesidades de una clase de aplicación o de un servicio particular de alto nivel; por ejemplo, distribución de contenido multimedia;

• funcionamiento más eficiente en un entorno de red dado - por ejemplo encaminamiento en una red ad hoc;

• una característica adicional - por ejemplo, comunicación multicast o segura.

Virtualización de red: (***overlay networks***)

De manera similar, cada red virtual tiene su propio esquema de direccionamiento particular, protocolos y algoritmos de enrutamiento, pero redefinido para satisfacer las necesidades particulares de las diferentes clases de aplicaciones.

Skype: Un ejemplo de *overlay network*