Computación Distribuida

Tema 2: El API de Sockets

Introducción

- El API de sockets es un interfaz de programación para la comunicación entre procesos (IPC) originalmente disponible en el sistema operativo Unix de Berkeley.
- Ha sido traducido a todos los sistemas operativos modernos, incluyendo Solaris de Sun y sistemas Windows.
- Es el estándar *de facto* para la programación IPC y es la base de interfaces IPC más sofisticados tales como remote procedure call (RPC) y remote method invocation (RMI).

Protocolo TCP/IP

- Introducido por el DoD en ARPANET a principios de los años 80.
- Principales características:
  - Independiente del fabricante
  - Disponible desde ordenadores personales a grandes supercomputadores
  - Usado tanto en LANs como WANs
  - Actualmente es el protocolo más extendido en Internet

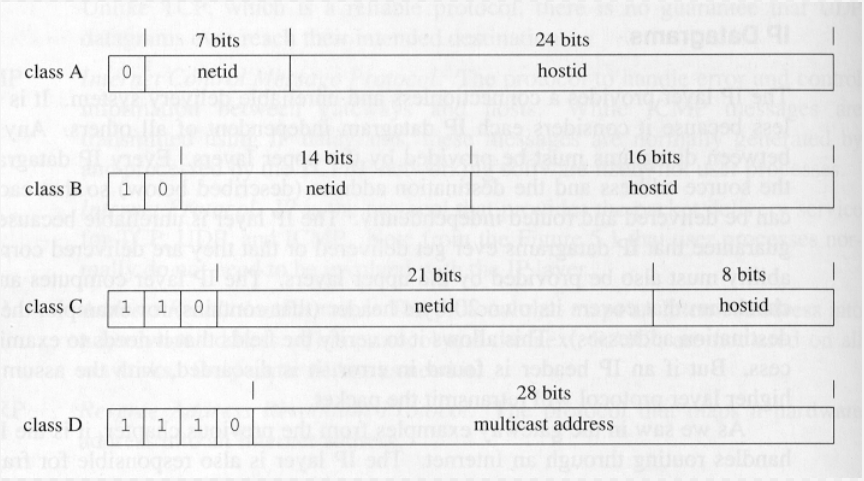
Capa de datos

- Actualmente TCP/IP soporta múltiples interfaces de comunicación
  - Líneas dedicadas de alta capacidad (T1, T3)
  - Redes locales
  - Incluso existen implementaciones sobre puertos serie (SLIP, PPP)

Capa de red

- En esta capa se proporciona un mecanismo no fiable de comunicación entre sistemas
- Se introduce el concepto de dirección IP
- Cada ordenador en la red dispone de una dirección única de 32 bits

Clasificación de las redes IP



Capa de transporte
UDP y TCP

- Un proceso interacciona con el protocolo TCP/IP mediante el envío de datos TCP o UDP
- A veces se conoce a estos protocolos bajo el nombre TCP/IP o UDP/IP

Orden de transmisión

- Los datos se pueden almacenar en la memoria de un ordenador de dos formas distintas:
  - Big endian: PowerPC, Sparc ...
  - Litle endian: Intel x86, ...
- Los datos a través de la red siempre se transmiten en formato big endian

UDP

- UDP es un protocolo no orientado a conexión sino al paso de mensajes.
- No garantiza la correcta recepción de los datos ni el orden de los mismos
- Simple = rápido
- Proporciona únicamente dos características no presentes en el protocolo IP
  - Número de puerto
  - Verificación del contenido de un paquete

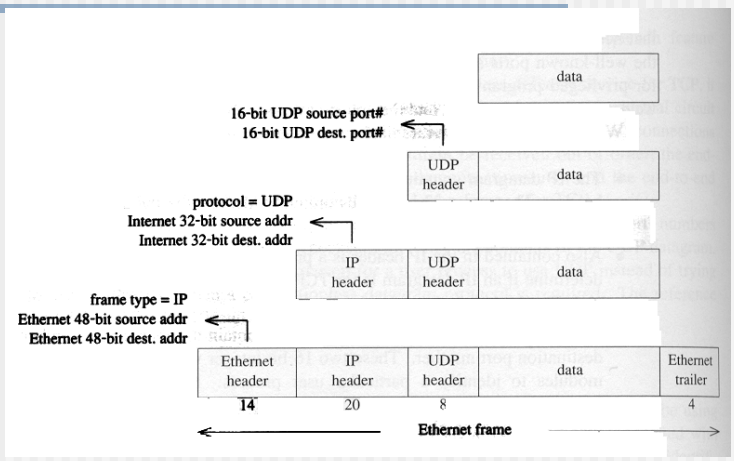
- Existen fronteras entre mensajes
- UDP se utiliza fundamentalmente en aplicaciones que requieren una alta tasa de transferencia de información y en donde las pérdidas de datos no son importantes
  - Ejemplo: videoconferencia
- No obstante, no existe el concepto de QoS (Quality of Service). No está garantizado un ancho de banda y pueden existir retardos. Esto implica que la telefonía IP necesita algo más que el protocolo TCP/IP

TCP

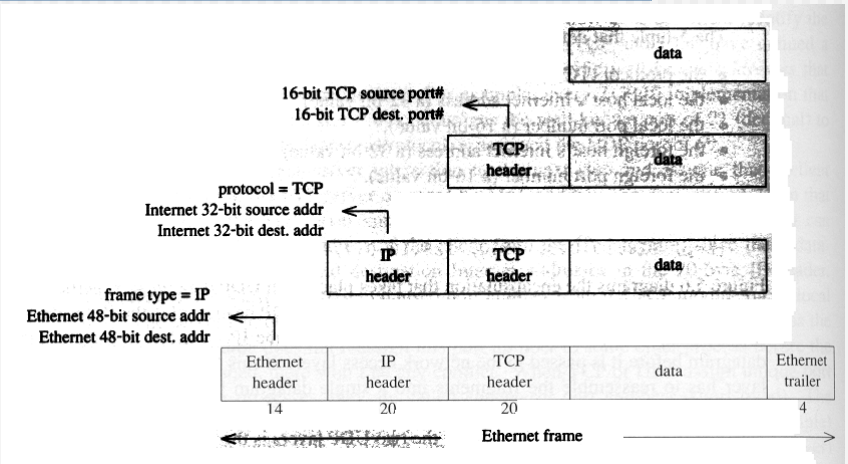
- TCP es un protocolo orientado a conexión
- Garantiza la correcta recepción de la información y el orden de los paquetes
- La comunicación es bidireccional

- TCP se usa para la transmisión fiable de información. Ejemplos:
  - telnet
  - ftp
  - Correo electrónico
  - http

Encapsulación UDP



Encapsulación TCP



Concepto de puerto

- Una máquina que dialoga con otra puede estar ejecutando varios procesos
- Es necesario identificar sin ambigüedad los procesos que intervienen en el diálogo
- Para ello se utiliza el concepto de puerto: entero de 16 bits

Asociación

- Una asociación en Internet consta de los siguientes elementos
  - Protocolo (TCP o UDP)
  - Dirección IP de la máquina local (32 bits)
  - Puerto de la máquina local (16 bits)
  - Dirección IP de la máquina remota (32 bits)
  - Puerto de la máquina remota (16 bits)
- Ejemplo: {tcp, 193.144.84.100, 1500, 193.170.2.11, 21}

Fragmentación

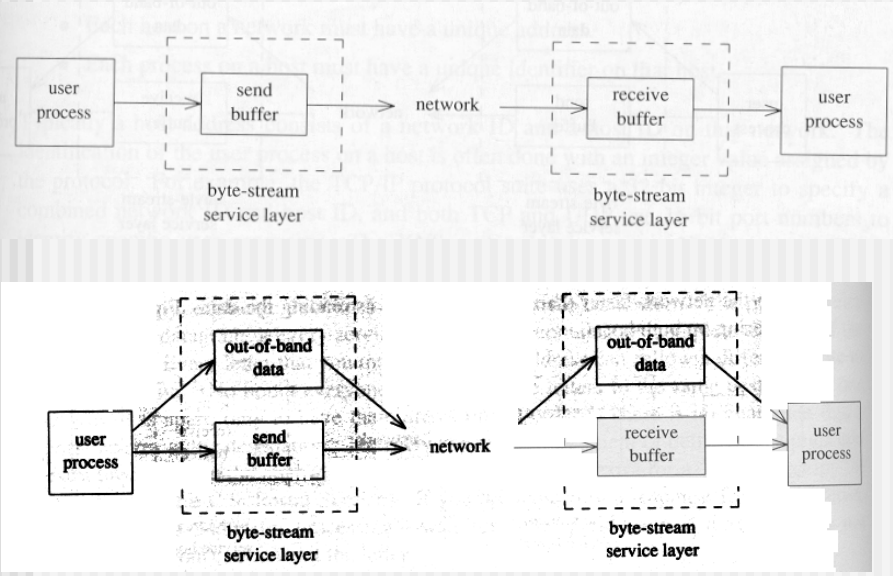
- La mayoría de las capas de red tienen un tamaño máximo de paquete de información que pueden manejar (MTU)
  - Ejemplo: ethernet = 1500 bytes
- Es necesario fragmentar la información
- Paquetes pertenecientes a un mismo mensaje pueden recibirse fuera de orden

Ensamblado

- En recepción es necesario reconstruir el mensaje ordenando los paquetes recibidos
- Se establece un número de orden de paquete. Sirve para varias cosas:
  - Reconstruir correctamente el mensaje
  - Detectar fallos en la transmisión y solicitar de nuevo la retransmisión

Out of Band (OOB)

- Generalmente la transmisión/recepción de información a través de una interfaz de comunicación se hace a través de una memoria intermedia
- A veces nos interesaría podernos saltar ese procedimiento
- Para ello surge el mecanismo de Out of Band.
- Los datos en el canal de Out of Band siempre se envían antes que el resto de los datos



IPC de UNIX BSD 4.x

- A partir de BSD 4.2 se implementa una IPC a través de llamadas al sistema
- socket: destino de mensajes
  - A través de un socket se pueden enviar y recibir mensajes
  - Los mensajes se ponen en una cola en el socket transmisor hasta que el protocolo de red los haya transmitido
  - En el socket receptor los mensajes estarán en una cola hasta que el proceso destinatario los haya extraído

- Antes de comunicarse, cada proceso debe crear explícitamente un socket
- Se crean mediante la llamada al sistema socket
- Es necesario especificar:
  - Protocolo: TCP o UDP
  - Dominio: UNIX (archivo dentro del árbol de directorios) o Internet (dirección IP+puerto)
- Devuelve un descriptor del socket susceptible de ser utilizado en llamadas al sistema posteriores

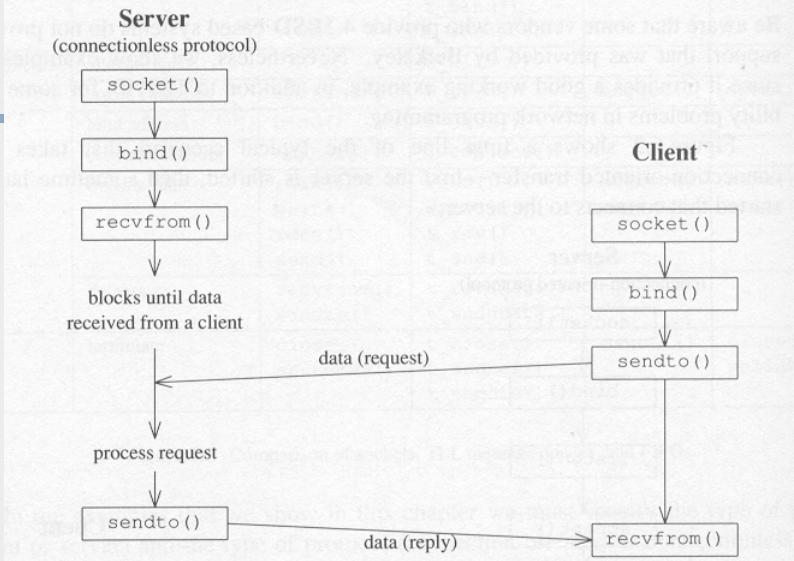
- Hay que darle un identificador que los demás procesos pueden utilizar como dirección de destino
- Para ello se utiliza la llamada al sistema bind

Comunicación por datagramas

- Primitiva de envío de mensajes: sendto
  - Además del mensaje hay que especificar
    - Descriptor de socket propio
    - Identificación del socket de destino (dirección IP+puerto)

- Primitiva de recepción de mensajes: recvfrom
  - Además del descriptor de socket propio, debe especificar el buffer en el que se almacena el mensaje y el remitente
- La comunicación tendrá lugar si se produce un emparejamiento
  - Un sendto hacia la dirección de un socket
  - Un recvfrom sobre el socket asociado a esa dirección

- Comunicación cliente-servidor
  - El servidor crea un socket, le asocia una dirección y la publica mediante algún método (NIS, p.ej.)
  - El cliente crea un socket, le asocia una dirección y obtiene la dirección del servidor
  - Mensaje de solicitud
    - El cliente lo envía con sendto
    - El servidor lo recoge con recvfrom
  - Mensaje de respuesta
    - Papeles invertidos

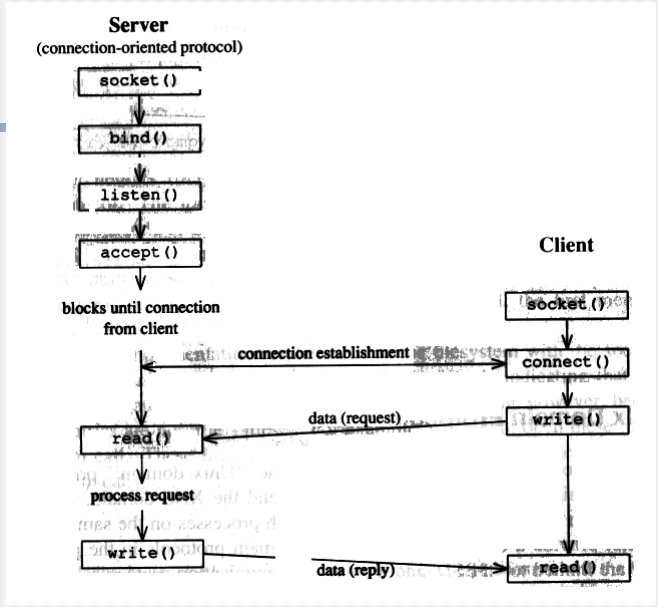


Cliente	Servidor
dsc = socket(AF_INET, SOCK_DGRAM, 0);	dss = socket(AF_INET, SOCK_DGRAM, 0);
...	
bind(dsc, &idCliente, longidCliente);	bind(dss, &idServ, longidServ);
sendto(dsc, solicitud, longSol, 0, idServ, longidServ); →	n = recvfrom(dss, &buffer, longBuf, 0, &idCliente, &longidRemite);
	...
←	sendto(dss, respuesta, longResp, 0, idCliente, longidCliente);
n = recvfrom(dsc, &buffer, longBuf, 0, &idServ, &longidServ);	
...	

Comunicación encauzada (stream)

- Primero se crea e identifica el par de sockets, de forma similar a como se hace por datagramas
- El servidor "escucha" llamadas: listen
- El cliente solicita la conexión: connect
- El servidor acepta la conexión: accept
- Conexión ⇒ se emparejan 1 connect y 1 accept
  - El Sistema Operativo crea un nuevo socket y lo conecta al del cliente
  - El servidor puede seguir escuchando en el socket original

- La comunicación entre el par de sockets conectados se hace por medio de las llamadas
  - read y write
  - send y recv
- La conexión concluye al cerrar el socket: close



Cliente	Servidor
dsc = socket(AF_INET, SOCK_STREAM, 0);	dss = socket(AF_INET, SOCK_STREAM, 0);
	bind(dss, &idServ, longServ);
connect(dsc, idServ, longServ);	
	listen(dss, 5);
... (se suspende)	... (se suspende)
	dsn = accept(dss, &remite, &long);
n = write(dsc, buffer, longBuf);	
	...
	n = read(dsn, &buffer, longBuf);

Servidor TCP en Java

```
import java.io.*;
import java.net.*;

public class Servidor {
   public static void main(String[] args) {
      try {
         ServerSocket s = new ServerSocket(8189);
         Socket socket = s.accept();
         DataInputStream input = new DataInputStream(socket.getInputStream());
         DataOutputStream output = new DataOutputStream(socket.getOutputStream());
         System.out.println(input.readByte());
         System.out.println(input.readShort());
         System.out.println(input.readInt());
         System.out.println(input.readUTF());
         output.writeByte(5);
         output.writeShort(345);
         output.writeInt(33270762);
         output.writeUTF("Se acabo correctamente");
         socket.close();
         s.close();
      }
      catch(Exception e) {
        System.out.println(e);
      }
   }
}
```

Cliente TCP en Java

```
import java.io.*;
import java.net.*;

public class Cliente {
   public static void main(String[] args) {
      try {
         Socket socket = new Socket("gsi2.dec.usc.es", 8189);
         DataInputStream input = new DataInputStream(socket.getInputStream());
         DataOutputStream output = new DataOutputStream(socket.getOutputStream());
         output.writeByte(5);
         output.writeShort(345);
         output.writeInt(33270762);
         output.writeUTF("Se acabo correctamente");
         System.out.println(input.readByte());
         System.out.println(input.readShort());
         System.out.println(input.readInt());
         System.out.println(input.readUTF());
         socket.close();
      }
      catch(Exception e) {
         System.out.println(e);
      }
   }
}
```

Algunas consideraciones

- Tanto bajo TCP como UDP existe un rango de puertos reservados [1,1023]. Únicamente los procesos con permisos de root podrán generar un socket en uno de esos puertos
- El puerto siempre se fija en la parte del servidor. El cliente genera su puerto de forma automática

- Por defecto, un socket siempre es bloqueante, de forma que cuando realizamos una operación de lectura sobre el mismo, se suspende la ejecución del proceso si no hay datos en el buffer
- Suele ser preferible generar un proceso aparte para leer datos de un socket que ponerlo en modo no bloqueante y gestionar su lectura por interrupciones. De todas formas, eso es un parámetro de diseño

Datagramas en Java

En Java tenemos dos clases proporcionadas por el API de sockets para datagramas:

1. la clase ***DatagramSocket*** para especificar el socket.
2. La clase ***DatagramPacket*** para representar al datagrama intercambiado.

Un proceso que desee enviar o recibir datos usando esta API debe crear una instancia de DatagramSocket. Cada socket esta ligado (bound) a un puerto UDP de la máquina local donde reside el proceso.

Datagramas en Java

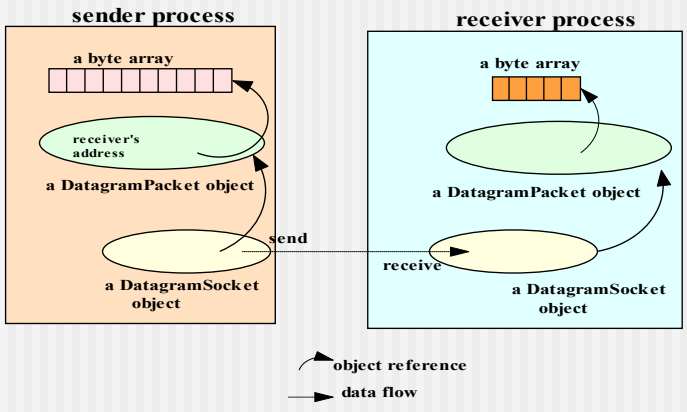
Para enviar un datagrama a otro proceso, un proceso:

- Crea un objeto que representa al datagrama. Este objeto puede ser creado como una instancia del objeto ***DatagramPacket*** que contiene
  1. (i) los datos a ser transmitidos
  2. (ii) la dirección de destino (dirección IP y puerto del socket receptor al cual está ligado).
- emite una llamada al método ***send*** del objeto ***DatagramSocket***, especificando como argumento una referencia al objeto de tipo ***DatagramPacket***.

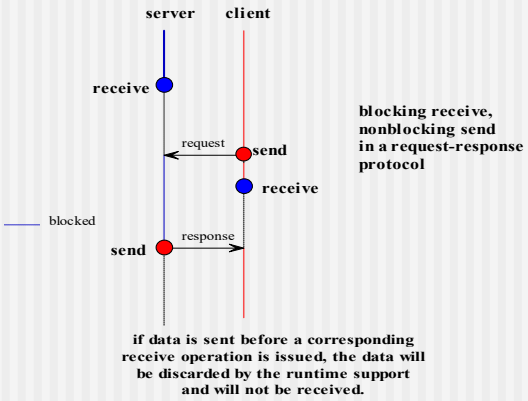
Datagramas en Java

- En el proceso receptor se debe instanciar un objeto de tipo *DatagramSocket* y ligarlo a un puerto local. Este número debe coincidir con el especificado en el datagrama del emisor.
- Para recibir datagramas enviados al socket, el proceso crea un objeto de tipo *DatagramPacket* que referencia a un array de bytes y llamar al método *receive* del objeto *DatagramSocket*, especificando como argumento una referencia al objeto *DatagramPacket*.

Estructuras de datos en los programas emisor y receptor



Sincronización de eventos con datagramas



Programando un timeout

Para evitar un bloqueo indefinido, debe programarse un timeout en un objeto socket:

void setSoTimeout(int timeout)

Programa un timeout para la recepción bloqueante de este socket en milisegundos.

Una vez programado, este timeout estará en vigor para todas las operaciones de carácter bloqueante.

Métodos clave y constructores

Method/Constructor	Description
DatagramPacket(byte[ ] buf, int length)	Construct a datagram packet for receiving packets of length *length;* data received will be stored in the byte array reference by *buf*.
DatagramPacket (byte[ ] buf, int length, InetAddress address, int port)	Construct a datagram packet for sending packets of length *length* to the socket bound to the specified port number on the specified host ; data received will be stored in the byte array reference by *buf*.
DatagramSocket ( )	Construct a datagram socket and binds it to any available port on the local host machine; this constructor can be used for a process that sends data and does not need to receive data.
DatagramSocket (int port)	Construct a datagram socket and binds it to the specified port on the local host machine; the port number can then be specified in a datagram packet sent by a sender.
void close( )	Close this datagramSocket object
void receive(DatagramPacket p)	Receive a datagram packet using this socket.
void send (DatagramPacket p)	Send a datagram packet using this socket.
void setSoTimeout (int timeout)	Set a timeout for the blocking receive from this socket, in milliseconds.

El código

```
//Excerpt from a receiver program
DatagramSocket ds = new DatagramSocket(2345);
DatagramPacket dp =
      new DatagramPacket(buffer, MAXLEN);
ds.receive(dp);
len = dp.getLength( );
System.out.Println(len + " bytes received.\n");
String s = new String(dp.getData( ), 0, len);
System.out.println(dp.getAddress( ) + " at port "
   + dp.getPort( ) + " says " + s);
```

```
// Excerpt from the sending process
InetAddress receiverHost=
    InetAddress.getByName("localHost");
DatagramSocket theSocket = new DatagramSocket( );
String message = "Hello world!";
byte[ ] data = message.getBytes( );
data = theLine.getBytes( );
DatagramPacket thePacket
    = new DatagramPacket(data, data.length,
                         receiverHost, 2345);
theSocket.send(theOutput);
```

Sockets no orientados a conexión

Con sockets no orientados a conexión es posible que múltiples procesos simultáneamente envíen datagramas al mismo socket creado por el proceso receptor, en cuyo caso el orden de recepción de los mensajes no es predecible.

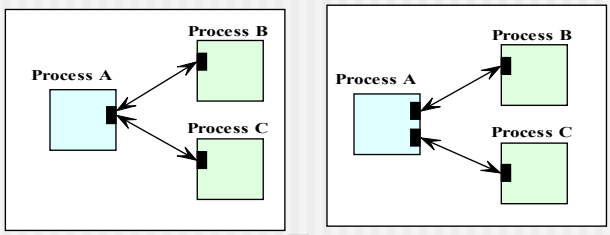
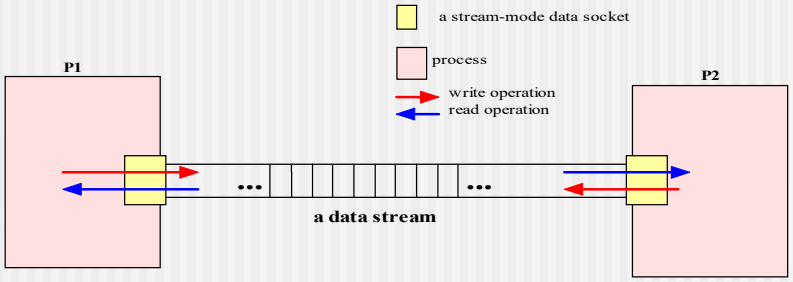


Figure 3a — a connectionless datagram socket — Figure 3b

El API de sockets en modo stream

- El API de socket para datagramas permite el intercambio de unidades de datos discretas (esto es, datagramas).
- El API de sockets en modo stream proporciona un modelo para la transmisión de datos basado en el modo encauzado de E/S proporcionado por los sistemas operativos Unix.
- Por definición, un socket en modo stream da soporte únicamente a comunicaciones orientadas a conexión.

El API de sockets en modo stream (API de sockets orientado a conexión)



El API de sockets en modo stream

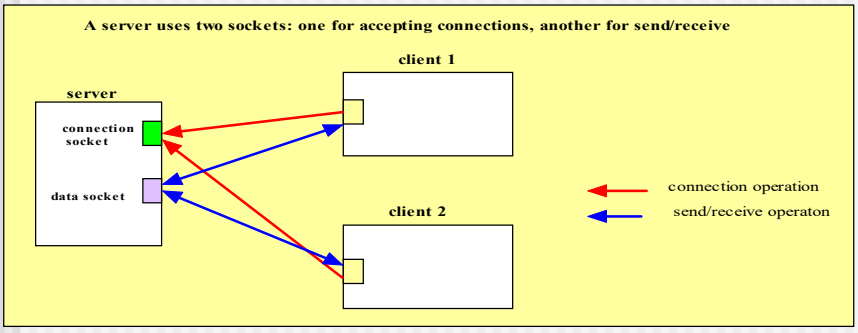
- Un socket en modo stream se crea para el intercambio de datos entre dos procesos específicos.
- Los datos es escriben en un extremo del socket y se leen en el otro extremo.
- Un socket stream no puede utilizarse para comunicarnos con más de un proceso.

El API de sockets en modo stream

En Java, el API de sockets en modo stream se proporciona mediante dos clases:

- Server socket: para aceptar conexiones; llamaremos a un objeto de esta clase un socket de conexión.
- Socket: para el intercambio de datos; llamaremos a un objeto de esta clase un socket de datos.

El servidor



Métodos clave de la clase ServerSocket

Method/constructor	Description
ServerSocket(int port)	Creates a server socket on a specified port.
Socket accept() throws IOException	Listens for a connection to be made to this socket and accepts it. The method blocks until a connection is made.
public void close() throws IOException	Closes this socket.
void setSoTimeout(int timeout) throws SocketException	Set a timeout period (in milliseconds) so that a call to accept( ) for this socket will block for only this amount of time. If the timeout expires, a java.io.InterruptedIOException is raised

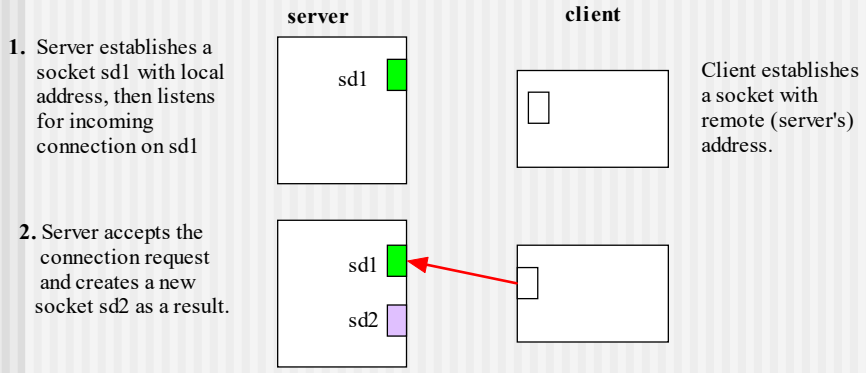
Nota: Accept es una operación bloqueante.

Métodos clave en la clase Socket

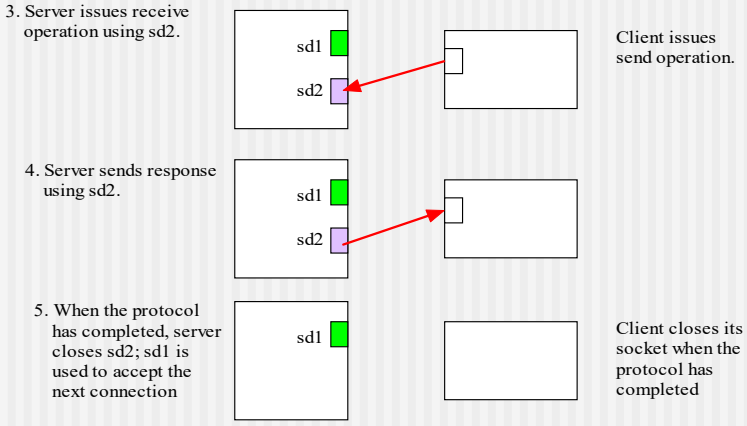
Method/constructor	**Description**
Socket(InetAddress address, int port)	Creates a stream socket and connects it to the specified port number at the specified IP address
void close() throws IOException	Closes this socket.
InputStream getInputStream( ) throws IOException	Returns an input stream so that data may be read from this socket.
OutputStream getOutputStream( )throws IOException	Returns an output stream so that data may be written to this socket.
void **setSoTimeout**(int timeout) throws SocketException	Set a timeout period for blocking so that a read( ) call on the InputStream associated with this Socket will block for only this amount of time. If the timeout expires, a **java.io.InterruptedIOException** is raised

Una operación de lectura en el InputStream es bloqueante. Una operación de escritura es no bloqueante.

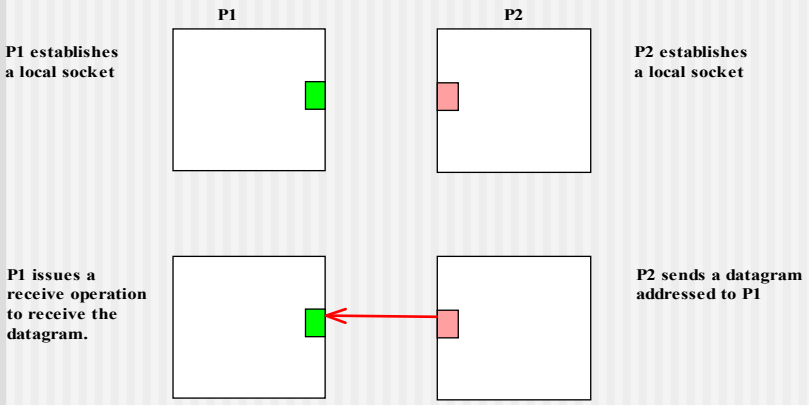
API de sockets orientados a conexión



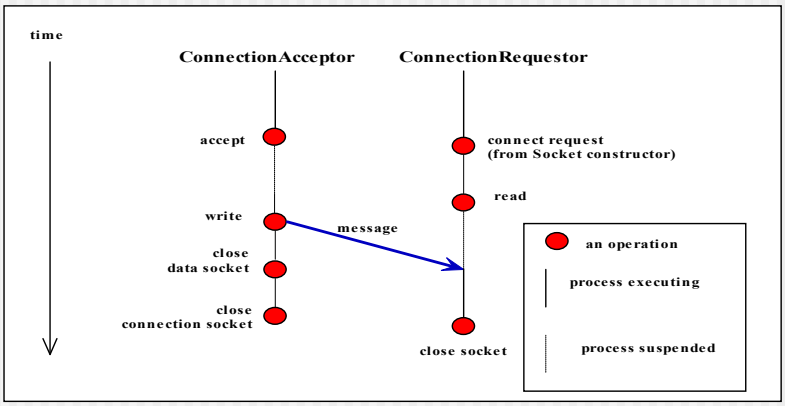
API de sockets orientados a conexión



API de sockets no orientados a conexión



Ejemplo: Diagrama de eventos



Ejemplo

Example4ConnectionAcceptor

```
try {
   nt portNo;
   String message;
   // instantiates a socket for accepting connection
   ServerSocket connectionSocket = new ServerSocket(portNo);
   Socket dataSocket = connectionSocket.accept();

   // get a output stream for writing to the data socket
   OutputStream outStream = dataSocket.getOutputStream();
   // create a PrinterWriter object for character-mode output
   PrintWriter socketOutput =
      new PrintWriter(new OutputStreamWriter(outStream));
   // write a message into the data stream
   socketOutput.println(message);
   //The ensuing flush method call is necessary for the data to
   // be written to the socket data stream before the
   // socket is closed.
   socketOutput.flush();
   dataSocket.close( );
   connectionSocket.close( );
} // end try
catch (Exception ex) {
 System.out.println(ex);
}
```

Example4ConnectionReceiver

```
try {
      InetAddress acceptorHost =
       InetAddress.getByName(args[0]);
       int acceptorPort = Integer.parseInt(args[1]);
      // instantiates a data socket
        Socket mySocket = new Socket(acceptorHost, acceptorPort);
      // get an input stream for reading from the data socket
        InputStream inStream = mySocket.getInputStream();
      // create a BufferedReader object for text-line input
       BufferedReader socketInput =
        new BufferedReader(new InputStreamReader(inStream));
      // read a line from the data stream
       String message = socketInput.readLine( );
       System.out.println("\t" + message);
       mySocket.close( );
} // end try
catch (Exception ex) {
     System.out.println(ex);
}
```

Sockets seguros

https://docs.oracle.com/javase/8/docs/technotes/guides/security/jsse/JSSERefGuide.html

- **Los sockets seguros realizan un cifrado de los datos transmitidos.**
- **El Java™ Secure Socket Extension (JSSE) es un paquete Java que permite comunicaciones seguras en Internet.**
- **Implementa una versión Java de los protocolos SSL (Secure Sockets Layer) y TLS (Transport Layer Security)**
- **Incluye funcionalidades para el cifrado de datos, autentificación del servidor, integridad de mensajes y, opcionalmente, autentificación del cliente.**

El API Java Secure Socket Extension

- **Import javax.net.ssl;**
- **La clase SSLServerSocket es una subclase de ServerSocket, y hereda todos sus métodos.**
- **La clase SSLSocket es una subclase de Socket, y hereda todos sus métodos.**

- **Para más información**
  - **https://www.owasp.org/index.php/Using_the_Java_Secure_Socket_Extensions**

Multidifusión

- La multidifusión IP se construye sobre el protocolo IP. 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 está al tanto de las identidades de los receptores y del tamaño del grupo.
- Los grupos de multidifusión se especifican utilizando direcciones Internet de la clase D.

- El convertirse en miembro de un grupo de multidifusión permite al computador recibir paquetes IP enviados al grupo.
- La pertenencia a los grupos de multidifusión es dinámica, permitiendo que los computadores se apunten o borren a un número arbitrario de grupos en cualquier instante. Es posible enviar mensajes a un grupo de multidifusión sin pertenecer al mismo.

Multidifusión en IPv4

- Routers multidifusión: los paquetes IP pueden multidifundirse tanto en una red local como en toda Internet. Si la multidifusión va dirigida a Internet, debe hacer uso de las capacidades de multidifusión de los routers, los cuales reenvían los datagramas únicamente a otros routers de redes que pertenezcan al mismo grupo. Para limitar la distancia de propagación de un datagrama multidifusión, el emisor puede especificar el número de routers que puede cruzar (TTL Time to Live).

- Reserva de direcciones multidifusión: las direcciones multidifusión se pueden reservar de forma temporal o permanente. Existen grupos permanentes, incluso cuando no tengan ningún miembro. Sus direcciones son asignadas de forma arbitraria por la autoridad de Internet en el rango de 224.0.0.1 a 224.0.0.255.
- El resto de las direcciones multidifusión están disponibles para su uso por parte de grupos temporales.

- Cuando se crea un grupo temporal, se necesita una dirección de multidifusión libre para evitar conflictos. El protocolo de multidifusión IP no resuelve el problema de reservar la dirección.
- Cuando la comunicación es a nivel local, si se pone un TTL pequeño, es difícil que entremos en conflicto con otros grupos.

- Si se necesita multidifusión a nivel de Internet, es necesario reservar previamente una dirección. El programa de directorio de sesiones (sd) sirve para arrancar o unirse a una sesión multidifusión. Proporciona una herramienta que permite a los usuarios detectar sesiones multidifusión existentes y anunciar su propia sesión, especificando el tiempo y la duración de la reserva.
- Para España quien gestiona dicha agenda es RedIris

  http://www.rediris.es/mmedia/agenda/

Multidifusión

```
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);


    // this figure continued on the next slide
```

```
            // 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();}
    }
}
```