# Tema 3: COMUNICACIÓN ENTRE PROCESOS

**E. U. Informática en Segovia**
**Departamento de Informática**
**Universidad de Valladolid**

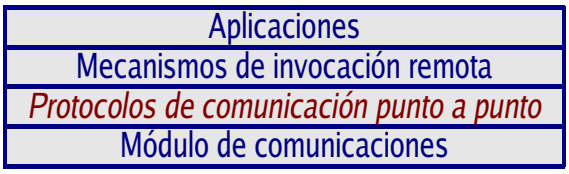




## Introducción

Estudia las comunicaciones punto a punto entre los lados de un enlace

Posibles capas de comunicación:




- Por debajo comunicación básica: TCP/IP
- Por encima
  - APIs de servicios usadas por aplicaciones
  - Otros protocolos de comunicación (RPC, ORB, mensajería...)

## Protocolos de comunicación (1)

Los procesos se comunican intercambiando mensajes ⇒ dos operaciones diferenciadas:

- send: el emisor envía un mensaje hacia el destino
- receive: el receptor recibe el mensaje en destino

Dos tipos:

- *Comunicación síncrona*
- *Comunicación asíncrona*





## Protocolos de comunicación (2)

- Comunicación síncrona: send y receive bloqueantes
  - Emisor bloqueado hasta que el receptor hace receive
  - Receptor bloqueado hasta que llega el mensaje

  Comunicación síncrona ⇒ sincronización entre emisor y receptor

- Comunicación asíncrona: send no bloqueante ⇒ buffer en emisor

Dos posibilidades:

  - *Recepción no bloqueante*: sondeo o interrupción de buffer en receptor
  - *Recepción bloqueante*: receptor bloqueado esperando por el mensaje ⇒ necesario receptor multihilo

  Receptor no bloqueante más eficiente, pero más complejo de implementar

## Protocolos de comunicación (3)

Especificación del destino del mensaje mediante dos campos

- Identificación de la máquina receptora. En internet IP o nombre (transparencia)
- Puerto: identifica una aplicación (número de 16 bits)

Además, tenemos:

- Comunicación *fiable*: se garantiza la entrega
- Comunicación *no fiable*: la entrega no está garantizada





## *Protocolos de comunicación: Sockets (1)*

Comunicación entre procesos al principio con *pipes*:

```
gunzip paquete.tar.gz | tar xvf -
```

Inconveniente: procesos en la misma máquina ⇒ *sockets*

*sockets* = ½ conexión para comunicación entre procesos en modo cliente/servidor

Cada *socket* caracterizado por:

- Una dirección IP asociada
- Un número de puerto asociado
- Un protocolo (UDP o TCP)

Cada conexión de transporte caracterizada por:

$$(IP_{local},\ puerto_{local},\ protocolo,\ IP_{remota},\ puerto_{remoto})$$

Protocolos de comunicaciones en *sockets*:

- Comunicación por *datagramas*
- Comunicación por cauces, canales o flujos (*streams*)

## Comunicación mediante datagramas

Características:

- Servicio de transporte sin conexión orientado a paquetes
- Eficiencia sin garantía de fiabilidad
- Posible duplicación, pérdida o desorden en los paquetes
- Protocolo sin conexión ⇒ descriptor del socket local y dirección socket receptor en cada envío
- Datos adicionales en cada comunicación

Cada proceso crea un socket y lo enlaza a un puerto local:

- Cliente a cualquier puerto local
- Servidor a un puerto conocido

Comunicación asíncrona bloqueante no garantizada

⇩

Fiabilidad ⇒ algún tipo de asentimiento





## Comunicación mediante streams

Características

- Conexión similar a una conexión física: una tubería entre procesos
- Comunicación fiable y garantizada: confirmaciones y reenvío de paquetes perdidos; se mantiene el orden
- Procesos leen y escriben del cauce sin especificar direcciones

Papeles diferenciados en cliente y servidor:

- Servidor: crea un socket de escucha con una cola de peticiones de conexión. Aceptación ⇒ creación de un nuevo socket encauzado
- Cliente: crea un socket encauzado y solicita el establecimiento de una conexión

## Diferencias datagramas-streams

- En UDP descriptor del socket local y dirección del socket receptor en cada envío el datagrama ⇒ mayor tamaño que en TCP
- En TCP conexión entre sockets antes de nada ⇒ tiempo de establecimiento
- En UDP límite de tamaño (64 kilobytes); TCP sin límite
- UDP es desordenado; TCP es ordenado
- UDP soporta multicast
- TCP indicado para servicios de red (telnet) o transmisión (ftp): longitud de datos indefinida
- UDP indicado para algunas aplicaciones de sistemas distribuidos

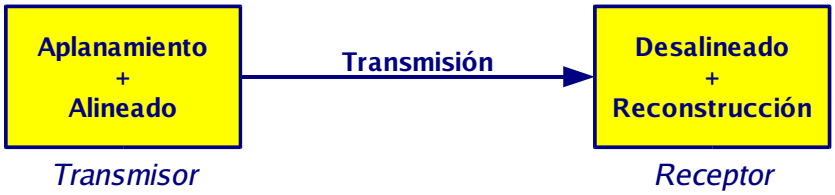




## *Protocolos de comunicación: Representación externa de los datos*

Datos estructurados, mensajes secuenciales ⇒ *Aplanado de los datos*

Desde el punto de vista de los datos, transmisión es:




Diferentes arquitecturas ⇒ distinta representación de los datos
- Consistencia: trasmisión sin modificación
- Negociación de la transmisión; usual en el formato del servidor
- Envío: datos + información de estructural

Normas de control de la representación de los datos:
- Corba CDR
- Sun XDR
- Java OS para Java RMI
- HTTP utiliza texto ASCII

Cada vez más importante: XML

# Comunicación por sockets

## *Sockets en* C (1)

### Creación de los sockets

Lo primero es usar socket:

```
int socket(int dominio, int tipo, int protocolo);
```

Crea un socket:

- dominio: local o a través de internet (PF_UNIX, PF_INET o PF_INET6)
- tipo: cauces (SOCK_STREAM), datagramas (SOCK_DGRAM), ...
- protocolo: identificador numérico de protocolo (0 si el tipo soporta solamente uno)
- Devuelve un descriptor de socket (utilizado en el resto de operaciones)

Con bind se asocia una dirección IP local y un puerto local al socket mediante el uso de una estructrura sockaddr

```
int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
```

Si el campo sin_addr especifica la macro INADDR_ANY el SO se encarga de asociar una de las direcciones IP disponibles al socket de forma automática

Si el campo sin_port especifica el valor 0, el SO se encarga de asociar un número de puerto de forma automática, y si es distinto de 0 uno específico (este último caso necesario para los servidores, que deben especificar un puerto bien conocido)





## *Sockets en* C (2)

### Comunicación por datagramas

| Sockets UDP | |
|---|---|
| Cliente | Servidor |
| socket( ) | socket( ) |
| bind( ) | bind( ) |
| sendto( ) | recvfrom( ) |

Pasos

- Ambos crean un socket con socket()
- El servidor establece el número de puerto
- Se transfieren datos con sendto() y recvfrom()

## *Sockets en* C (3)

| Sockets UDP | |
|---|---|
| Cliente | Servidor |
| socket( ) | socket( ) |
| bind( ) | bind( ) |
| sendto( ) | recvfrom( ) |

Prototipo de sendto:

```
int sendto(int sockfd, const void *msg, int len, unsigned int flags,
    const struct sockaddr *to, int tolen)
```

Argumentos:

- sockfd: descriptor socket local a través del cual se envían los datos
- msg: puntero a los datos a ser enviados
- len: longitud de los datos en bytes
- flags: varias funciones como limitar el envío a máquinas “a este lado” de un gateway
- to: puntero a estructura que contiene la dirección IP y número de puerto destino
- tolen: tamaño de struct sockaddr (sizeof (struct sockaddr))





## *Sockets en* C (4)

| Sockets UDP | |
|---|---|
| Cliente | Servidor |
| socket( ) | socket( ) |
| bind( ) | bind( ) |
| sendto( ) | recvfrom( ) |

Prototipo de recvfrom:

```
int recvfrom (int sockfd, void *buf, int len, unsigned int flags,
    struct sockaddr *from, int *fromlen)
```

Argumentos:

- sockfd: descriptor socket local a través del cual se leen los datos
- buf: buffer de almacenamiento de datos
- len: longitud de buf
- flags: varias funciones como recibir datos sin que se eliminen de la cola
- to: puntero a estructura que contiene la dirección IP y número de puerto del host origen de los datos
- tolen: tamaño de struct sockaddr (sizeof (struct sockaddr))

# *Sockets en* C (5)

## Comunicación por streams

| Sockets TCP | |
|---|---|
| Cliente | Servidor |
| socket( ) | socket( ) |
| bind( ) | bind( ) |
| | listen( ) |
| | accept( ) |
| connect( ) | |
| send( ) | recv( ) |

Pasos

- Ambos crean un socket con socket()
- El servidor establece el número de puerto
- El servidor habilita su socket para recepción con listen() y ejecuta accept() para quedar en espera
- El cliente usa connect() ⇒ accept() retorna con un nuevo descriptor de sockts para la transferencia
- Se transfieren datos con send() y recv()





# *Sockets en* C (6)

Prototipo de listen:

```
int listen (int sockfd, int backlog)
```

Argumentos:

- sockfd: descriptor devuelto por la funcion socket() que se usa para recibir conexiones
- backlog: número máximo en la cola de entrada de conexiones (independientemente de si se tratan de conexiones ya establecidas o pendientes de establecer

Prototipo de accept:

```
int accept (int sockfd, void *addr, int *addrlen)
```

Argumentos:

- sockfd: descriptor de socket habilitado para recibir conexiones
- addr: puntero a una estructura que almacena información de la conexión entrante

addrlen: tamaño de sockaddr (sizeof(struct sockaddr))

Retorna un nuevo descriptor de socket sobre el que se establece la conexión con el cliente y será la que haya que utilizar en la comunicación con éste

| Sockets TCP | |
|---|---|
| Cliente | Servidor |
| socket( ) | socket( ) |
| bind( ) | bind( ) |
| bind( ) | listen( ) |
| | accept( ) |
| connect( ) | accept( ) |
| send( ) | recv( ) |
| send( ) | recv( ) |

## *Sockets en C (7)*

| Sockets TCP | |
|---|---|
| Cliente | Servidor |
| socket( ) | socket( ) |
| bind( ) | bind( ) |
| | listen( ) |
| | accept( ) |
| connect( ) | |
| send( ) | recv( ) |

Prototipo de connect:

```
int connect (int sockfd, struct sockaddr *serv_addr, int addrlen)
```

Argumentos:

- sockfd: descriptor de socket devuelto socket( )
- serv_addr: estructura que contiene dirección IP y número de puerto destino
- addrlen: tamaño de sockaddr (sizeof (struct sockaddr))





## *Sockets en C (8)*

| Sockets TCP | |
|---|---|
| Cliente | Servidor |
| socket( ) | socket( ) |
| bind( ) | bind( ) |
| | listen( ) |
| | accept( ) |
| connect( ) | |
| send( ) | recv( ) |

send( ) y recv( ) son como sendto( ) y recvfrom( ) pero simplificadas:

```
send (int sockfd, const void *msg, int len, int flags)
```

Argumentos:

- sockfd: socket por donde se enviarán los datos
- msg: puntero a los datos
- len: longitud de los datos en bytes
- flags: igual que sendto( )

```
recv (int sockfd, void *buf, int len, unsigned int flags)
```

Argumentos:

- sockfd: socket por donde se reciben los datos
- buf: buffer donde se almacenarán los datos
- len: longitud de buf
- flags: igual que recvfrom( )

# *Sockets en* Java (1)

En *Java* más sencilla que en *C* por las facilidades del lenguaje

Sockets TCP

| Sockets TCP | | |
|---|---|---|
| Cliente | | Servidor |
| Socket(host,port#) | | ServerSocket(port#,backlog) |
| | | accept( ) |
| InputStream | → | OutputStream |
| close( ) | | close( ) |

- El servidor establece un puerto y el tamaño de la cola de conexiones
- El servidor hace una operación accept sobre el ServerSocket para quedar a la espera de conexiones
- El cliente establece una conexión
- El cliente y el servidor se comunican con manejadores InputStream y OutputStream
- Los applets sólo se conectan con el nodo desde el que se transfirió su código (importante por seguridad)
- En ServerSocket se puede especificar el número máximo de conexiones (backlog)
- Cada conexión en servidor ⇒ hilo para atender conexiones simultáneas





# *Sockets en* Java (2)

Implementación del servidor:

```
import java.net.*;
import java.io.*;
public class sockettcpser {
   public static void main(String argv[]) {
      System.out.println("Prueba de sockets TCP (servidor)");
      ServerSocket socket;
      boolean fin = false;

      try {
         socket = new ServerSocket(6001);
         Socket socket_cli = socket.accept();
         DataInputStream in =
            new DataInputStream(socket_cli.getInputStream());
         do {
            String mensaje ="";
            mensaje = in.readUTF();
            System.out.println(mensaje);
         } while (1>0);
      }
      catch (Exception e) {
         System.err.println(e.getMessage());
         System.exit(1);
      }
   }
}
```

Crea un socket en el puerto 6001

Pone el socket en escucha y devuelve una referencia a un nuevo objeto Socket

Obtiene un stream de entrada asociado al socket

Lee líneas del canal de entrada

# *Sockets en* Java (3)

## Implementación del cliente:

```
...
      try {
         System.out.print("Capturando dirección de host... ");
         address=InetAddress.getByName(argv[0]);
         System.out.println("ok");

         System.out.print("Creando socket... ");
         socket = new Socket(address,6001);
         System.out.println("ok");

         DataOutputStream out =
            new DataOutputStream(socket.getOutputStream());

         System.out.println("Introduce mensajes a enviar:");

         do {
            mensaje = in.readLine();
            out.writeUTF(mensaje);
         } while (!mensaje.startsWith("fin"));
      }
      catch (Exception e) {
         System.err.println(e.getMessage());
         System.exit(1);
      }
...
```

Obtiene la dirección IP del servidor

Busca un socket en el puerto 6001 del servidor

Crea un canal de salida asociado al socket

Escribe en el canal de salida





# *Sockets en* Java (4)

## Sockets UDP

| Sockets UDP | | |
|---|---|---|
| Cliente | | Servidor |
| DatagramSocket( ) | | DatagramSocket(port#) |
| send( ) | → | receive( ) |

- El servidor establece un puerto y espera por los clientes
- El cliente crea un mensaje en el que especifica la dirección y el puerto del servidor
- El cliente manda el mensaje y el servidor lo recibe

# *Sockets en* Java (5)

## Implementación del servidor:

```
...
      try {
         System.out.print("Creando socket... ");
         socket = new DatagramSocket(6000);
         System.out.println("ok");

         System.out.println("Recibiendo mensajes... ");
         do {
           byte[] mensaje_bytes = new byte[256];
           DatagramPacket paquete =
              new DatagramPacket(mensaje_bytes,256);
           socket.receive(paquete);
           String mensaje ="";
           mensaje = new String(mensaje_bytes);
           System.out.println(mensaje);
           if (mensaje.startsWith("fin")) fin=true;
         } while (!fin);
      }
      catch (Exception e) {
         System.err.println(e.getMessage());
         System.exit(1);
      }
...
```

Crea un socket en el puerto 6000

Crea una estructura para recibir el mensaje

Recibe el mensaje





# *Sockets en* Java (6)

## Implementación del cliente:

```
...
      try {
         System.out.print("Creando socket... ");
         socket = new DatagramSocket();
         System.out.println("ok");

         System.out.print("Capturando dirección de host... ");
         address=InetAddress.getByName(argv[0]);
         System.out.println("ok");

         System.out.println("Introduce mensajes a enviar:");

         do {
            mensaje = in.readLine();
            mensaje_bytes = mensaje.getBytes();
            paquete = new DatagramPacket(mensaje_bytes
               ,mensaje.length(),address,6000);
            socket.send(paquete);
         } while (!mensaje.startsWith("fin"));
      }
      catch (Exception e) {
         System.err.println(e.getMessage());
         System.exit(1);
      }

...
```

Crea un socket

Obtiene la dirección IP del servidor

Crea el mensaje incluyendo la dirección del servidor y el puerto

Envía el mensaje