# Sistemas Distribuidos

## Grado en Ingeniería Informática

### T2: Comunicación entre procesos

Departamento de Ingeniería Informática
Universidad de Cádiz



Grado en Ingeniería Informática

# Indice

1 Introducción

2 API para los Protocolos de Interntet: Sockets

3 Comunicación cliente-servidor

4 Ejemplos de protocolos de comunicación

5 Tareas

# Sección 1 | Introducción

# Comunicación entre procesos (I)

- Características de la comunicación entre procesos:
  - Comunicación síncrona y asíncrona.
  - Destinos de los mensajes.
  - Fiabilidad.
  - Ordenación.
- Las entidades que se comunican son procesos cuyos papeles determinan cómo se comunican (“patrones de comunicación”).
- La aplicación se comunica con:
  - UDP con “paso de mensajes”.
  - TCP con “flujo de datos”.

# Comunicación entre procesos (II)

- Los patrones de comunicación principales son:
  - Comunicación cliente-servidor ⇒ sockets.
  - Comunicacion en grupo (multidifusión) ⇒ 0MQ.
- Tipos de comunicación:
  - Recurso compartido.
  - Paso de mensajes.

# Sección 2 | API para los Protocolos de Interntet: Sockets

# Protocolos TCP y UDP

## Dos tipos de conexiones

**TCP** Comunicación punto-a-punto.

**UDP** Comunicación por datagrama/paquete.

## Fiabilidad

- Comunicación punto a punto fiable:
  - Se garantiza la entrega, aunque se pierda un número razonable de paquetes.
- Comunicación no fiable:
  - La entrega no se garantiza, aunque sólo se pierda un único paquete.

## Ordenación

Algunas aplicaciones necesitan que los mensajes sean entregados en el orden de su emisión.

# Protocolos TCP y UDP

## Dos tipos de conexiones

**TCP** Comunicación punto-a-punto.

**UDP** Comunicación por datagrama/paquete.

## Fiabilidad

- Comunicación punto a punto fiable:
  - Se garantiza la entrega, aunque se pierda un número razonable de paquetes.
- Comunicación no fiable:
  - La entrega no se garantiza, aunque sólo se pierda un único paquete.

## Ordenación

Algunas aplicaciones necesitan que los mensajes sean entregados en el orden de su emisión.

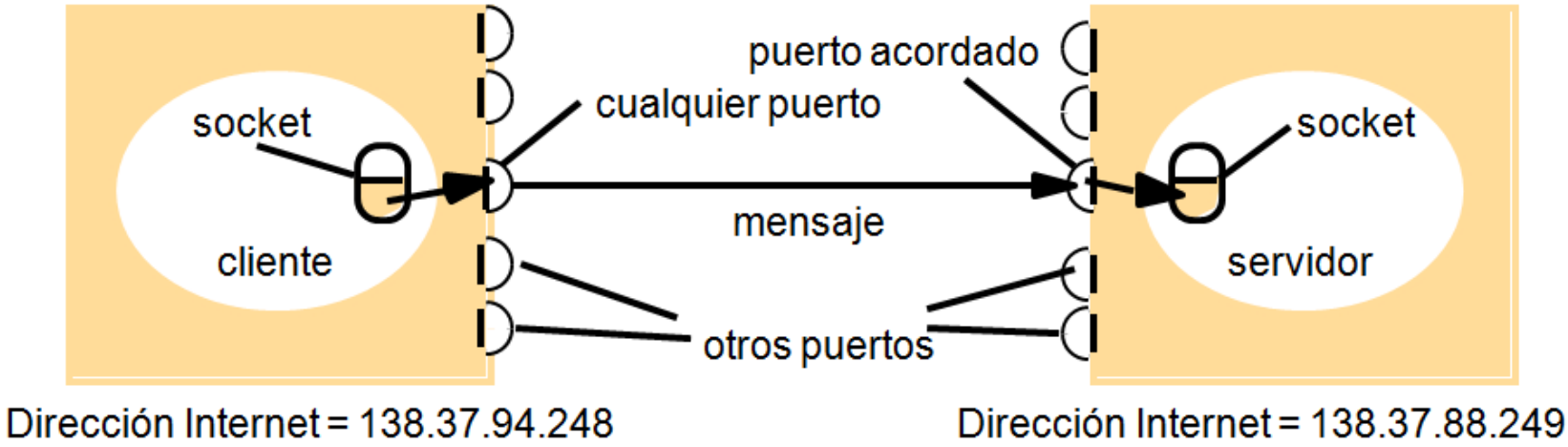
# Comunicación entre procesos

## Destino de los mensajes

- Los mensajes son enviados a direcciones construidas por pares (dirección Internet, puerto local).
- Un puerto local:
  - Es el destino de un mensaje dentro de un computador (número 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 el número de puerto puede enviarle un mensaje.

# Sockets (III)

El proceso que posee el socket es el único que puede recibir mensajes destinados al puerto asociado.



# Envío de datos

- El paso de un mensaje se puede llevar a cabo mediante 2 operaciones de comunicación:
  - *send*: Un proceso envia un mensaje a un destino.
  - *receive*: Un proceso recibe el mensaje en el destino.
- Cada destino tiene asociada una cola de mensajes:
  - Los emisores añaden mensajes a la cola.
  - Los receptores los extraen.
- Características de la comunicación entre procesos:
  - Comunicación síncrona y asíncrona.
  - Destinos de los mensajes.
  - Fiabilidad.
  - Ordenación.

# Sockets (IV)

## Interfaz Sockets: funciones UDP

- Cliente:
  - Crear socket.
  - Enviar/recibir.
  - Cerrar socket.
- Servidor:
  - Crear socket.
  - Enviar/recibir.
  - Cerrar socket.

# Sockets (V)

## Interfaz Sockets: funciones TCP

- Cliente:
  - Crear socket.
  - Conectarse.
  - Enviar/recibir.
  - Cerrar socket.
- Servidor:
  - Crear socket.
  - Enlazar a *port* y *port* (*bind*).
  - Escuchar.
  - Aceptar conexiones.
  - Enviar/recibir.
  - Cerrar socket.

# Ejemplo: Hello world en Python

## Servidor

```
import socket

MAX_CLIENTS = 10

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.bind((socket.gethostname(), 13000))
sock.listen(MAX_CLIENTS)
coding = "utf-8"

while (True):
    (client, address) = sock.accept()
    name = client.recv(4000)
    msg = "Hello, " +name.decode(coding)
    client.sendall(bytes(msg, encoding=coding))
```

# Ejemplo: Hello world en Python

## client

```
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((socket.gethostname(), 13000))
sock.sendall(b"Juan")
max_size = 1000
msg = sock.recv(max_size)
print(msg)
```

# Comunicación entre procesos (II)

## Comunicación síncrona

- El emisor y el receptor se sincronizan en cada mensaje.
- *send* y *receive* son operaciones bloqueantes:
  - El emisor se bloquea hasta que el receptor hace *receive*.
  - El receptor se bloquea hasta que le llegue un mensaje.

# Comunicación entre procesos (III)

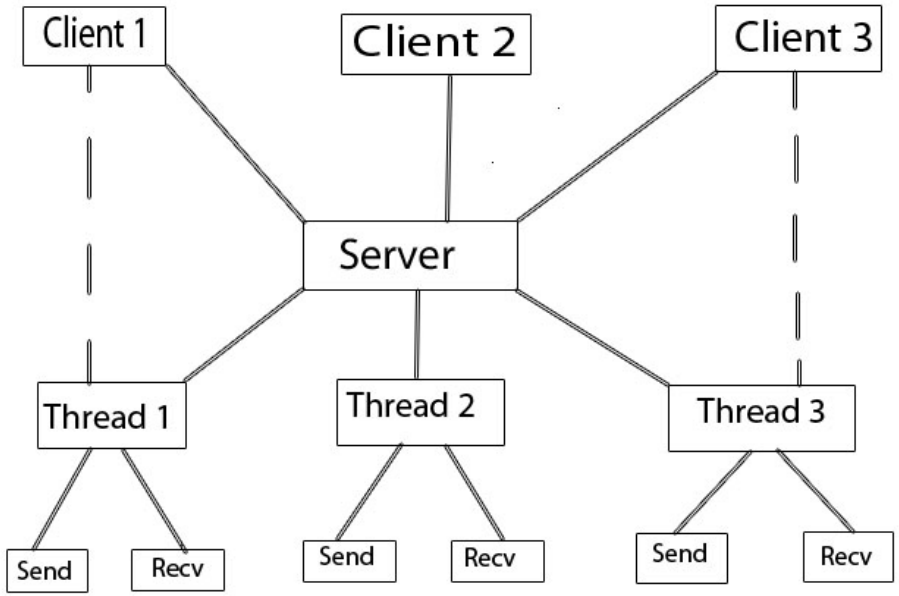
## Comunicación asíncrona (I)

- La operación *send* no bloqueante:
  - El programa emitor contigua aunque todavía el otro no se haga *receive*.
- *Receive* no bloqueante:
  - El proceso receptor sigue con su programa después de invocar la operación *receive*.
  - Proporciona un búfer que se llenarán en segundo plano.
  - El proceso debe ser informado por separado de que su búfer ha sido llenado (sondeo o interrupción).

# Comunicación entre procesos (III)

## Comunicación asíncrona en servidor (II)

- ¿El programa debe esperar a atender a un cliente para atender a otro?
  - Atender cada cliente en una hebra nueva.
  - Es más eficiente pero más complejo.



# Comunicación de datagramas UDP (I)

- Mensaje autocontenido no fiable desde un emisor a un receptor.
- Único mensaje sin confirmación o reenvío (sin garantía de entrega).
- Trasmisión entre 2 procesos: *send* y *receive*.
- Cada proceso debe crear un socket y enlazarlo a un puerto local:
  - Los clientes a cualquier puerto local libre.
  - Los servidores, a un puerto de servicio determinado.
- Se utiliza comunicación asíncrona con *receive* bloqueante:
  - *send* devuelve el control cuando ha dirigido el mensaje a las capas inferiores UDP e IP (responsables de su entrega en el destino).
  - Estas capas lo transmiten y lo dejan en la cola sel socket asociado al puerto de destino.
  - *receive* extrae el mensaje de la cola con un bloqueo indefinido (por defecto), a menos que se haya establecido un tiempo límite (*timeout*) asociado al conector.

# Comunicación de datagramas UDP (II)

## Riesgos

- No hay garantía de la recepción:
  - Los mensajes se pueden perder por errores de *checksum* o falta de espacio.
  - Los procesos deben proveer la calidad que deseen.
- Al enviarse paquete ha paquete el orden no está asegurado.
- Se puede dar un servicio fiable sobre uno no fiable, pero:
  - No es imprescindible.
  - Provoca grandes cargas administrativas:
    - Almacena información de estado en origen y destino.
    - Transmite mensajes adicionales.
    - Puede existir latencia para emisor o receptor.

# Comunicación de flujos TCP (I)

- La abstracción de flujos (*streams*) oculta:
  - Tamaño de los mensajes: los procesos leen o escriben cuanto quieren y las capas inferiores (TCP/IP) se encargan de empaquetar.
  - Mensajes perdidos: a través de asentimientos y reenvíos.
  - Control de flujo: evita desbordamiento del receptor.
  - Mensajes duplicados y/o desordenados.
  - Destinatarios de los mensajes: tras la conexión, los procesos leen y escriben del cauce sin tener que utilizar nuevamente sus respectivas direcciones.
- Se distiguen claramente las funciones del cliente y servidor:
  - Cliente: crea un socket encauzado y solicita el establecimiento de una conexión.
  - Servidor: crea un socket de escucha con una cola de peticiones de conexión, asociado a un número de puerto y se queda a la espera de peticiones de conexión.

# Comunicación de flujos TCP (II)

- Al aceptar una conexión el servidor:
  - Se crea un nuevo socket encauzado conectado al del cliente.
  - Cada proceso lee de su entrada y escribe en su salida.
  - Si un proceso cierra su socket, los datos pendientes se transmitirán y se indicará que el cauce está roto.
- Puede haber bloqueo:
  - Lectura: no hay datos disponibles.
  - Escritura: la cola del socket de destino está llena.
- Opciones para atender a múltiples clientes:
  - Escucha selectiva.
  - Múltiples hebras.
- La conexión se romperá si se detectan errores graves de red.

# Sección 3 | Comunicación cliente-servidor

# Comunicación segura

## Motivos de error

- Tiempo de espera límite.
- Eliminación de mensajes de petición duplicados.
- Pérdida de mensajes de respuesta.
- Historial.

- La comunicación cliente-servidor es síncrona (el cliente se bloquea hasta recibir una respuesta, o pasa el *timeout*).

# Fallos en la entrega

- Los mensajes se pueden perder.
- Puede haber rotura: parte de la red aislada.
- Los procesos pueden fallar.
- Dificultad de elegir un N reintentos.
- Los datos si se reciben serán correctos.
- Se incorpora un temporizador al *receive* del cliente.

# Temporizadores

- Cuando vence el temporizador del *receive* del cliente, el módulo de comunicaciones puede:
  - Retornar inmediatamente indicando el fallo ocurrido (poco común).
  - Reintentarlo repetidamente hasta obtener una respuesta o cuando exista probabilidad de que el servidor ha fallado.

# Solicitudes duplicadas

- El servidor puede recibir solicitudes duplicadas, si los reintentos llegan antes de tiempo (servidor lento o sobrecargado).
- Para evitar estas ejecuciones repetidas:
  - Reconocer los duplicados del mismo cliente (igual *idInvocacion*).
  - Filtarlos (descartarlos)

# Respuestas perdidas

- Provocan que el servidor repita una operación.
- No es un problema cuando las operaciones del servidor son idempotentes:
  - Se pueden de realizar de forma repetida.
  - Los resultados son los mismos que si se ejecutasen una sola vez.

# Historial

- Objetivo: retransmitir una respuesta sin volver a ejecutar la operación.
- El historial es el registro de las respuestas enviadas:
  - Mensaje con *idInvocacion* y su destinatario.
  - Gran consumo de memoria: se podrían descartar las respuestas tras algún tiempo.

# Protocolos de intercambio

- 3 protocolos que se suelen utilizar:
  - R (*request* o petición).
  - RR (*request-reply* o petición-respuesta).
  - RRA (*request-reply-acknowledge reply* o petición-respuesta-confirmación de la respuesta).

| Nombre | Cliente | Mensajes enviados por Servidor | Mensajes enviados por Cliente |
|---|---|---|---|
| R | Petición | | |
| RR | Petición | Respuesta | |
| RRA | Petición | Respuesta | Confirmación respuesta |

# Protocolos de intercambio RPC (II)

## Protocolo R

- Sólo es útil cuando no hay valor de retorno del procedimiento y el cliente no necesita información.
- El cliente continúa tras enviar la solicitud.

## Protocolo RR

- Común en entornos cliente-servidor.
- La respuesta asiente la solicitud.
- Una solicitud posterior del mismo cliente asiente la respuesta.

# Protocolos de intercambio RPC (III)

## Protocolo RRA

- La respuesta asiente la solicitud.
- El asentimiento de la respuesta lleva la *idInvocacion* de la respuesta a la que se refiere, asiente dicha respuesta y la de *Idvocacion* anterior, y permite vaciar entradas del historial.
- El envío del asentimiento de respuesta no bloquea al cliente pero consume recursos de procesador y red.

# Sección 4 | Ejemplos de protocolos de comunicación

# Protocolo de petición-respuesta TCP

El protocolo de petición-respuesta más conocido es HTTP:

- Es un protocolo de texto, muy utilizado.
- La longitud de UDP podría no ser adecuada.
- El TCP asegura que los datos sean entregados de forma fiable.
- Métodos: GET, POST (los más comunes), PUT, HEAD, DELETE, OPTIONS.
  - GET para peticiones (paso de parámetros en la URL).
  - POST para enviar datos (parámetros ocultos).
- Permite: autenticación y negociación del contenido.
- Establece conexiones persistentes, y abiertas durante el intercambio de mensajes.

# Petición GET HTTP

## Parámetros

- Se indica dentro de la URL: RUTA?var1=valor1&var2=valor2&...
- Distintas opciones se ponen luego de la ruta.
- Se termina la petición tras dos lineas en blanco.

## Sintaxis

```
GET <URL>
Host: <Host>
User-Agent: <User-Accept>
Accept: ...
```

| | |
|---|---|
| URL | Ruta (sin dominio). |
| Host | Máquina que responde. |
| User-Accept | Identificador del navegador/SO. |
| Accept | formato aceptado (codificación, comprimido, ...). |

# Ejemplo usando python

## Petición HTTP *a pelo*

```
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
ip = socket.gethostbyname("www.marca.com")
sock.connect((ip, 80))
sock.sendall(b"GET␣/␣HTTP/1.1\nHost:␣www.marca.com\n\n")
bufsize = 4000
output = ""
buf = sock.recv(bufsize)

while (buffer):
    output += buf.encode(utf-8)
    buf = sock.recv(bufsize)
```

# Sección 5 | Tareas

# Tareas

1. Describa dos escenarios en los que sea necesario comunicación síncrona, y dos escenarios de comunicación asíncrona.
2. ¿Resulta razonablemente útil que un puerto tenga varios receptores?
3. Un servidor crea un puerto que utiliza para recibir peticiones de sus clientes. Discuta los problemas de diseño concernientes a las relaciones entre el identificador de este puerto y los utilizados por los clientes:
   - ¿Cómo sabe el cliente qué puerto y dirección IP utilizar para acceder a un servicio?
   - Eficiencia de acceso a puertos e identificadores locales.

# Bibliografía

Coulouris, G.; Dollimore, J.; Kindberg, T.
Distributed Systems: Concepts and Design (5ª ed.)
Addison-Wesley, 2012.
(Trad. al castellano: Sistemas distribuidos: conceptos y diseño, 3ª ed., Pearson 2001)