

# Capa Aplicación: Programación de sockets

## ELO322: Redes de Computadores Agustín J. González

Este material está basado en:

- Material de apoyo al texto *Computer Networking: A Top Down Approach Featuring the Internet*. Jim Kurose, Keith Ross.

# Capítulo 2: Capa Aplicación

- ❑ 2.1 Principios de la aplicaciones de red
- ❑ 2.2 Web y HTTP
- ❑ 2.3 Correo Electrónico
  - SMTP, POP3, IMAP
- ❑ 2.4 DNS
- ❑ 2.5 P2P para archivos compartidos
- ❑ 2.6 Video streaming y redes de distribución de contenidos (Redes II)
- ❑ 2.7 Programación de sockets con UDP y TCP

# Programación de Sockets (1)

**Objetivo:** adquirir familiaridad sobre cómo construir aplicaciones cliente servidor que se comunican usando sockets

## API para sockets

- ❑ Fue introducida en BSD4.1 UNIX, 1981
- ❑ El socket es explícitamente creado, usado, y cerrado por las aplicaciones
- ❑ Sigue el modelo cliente/servidor
- ❑ Hay dos tipos de servicios de transporte vía el API de socket:
  - Datagramas no confiables (UDP)
  - Orientado a un flujo de bytes, éste es confiable (TCP)

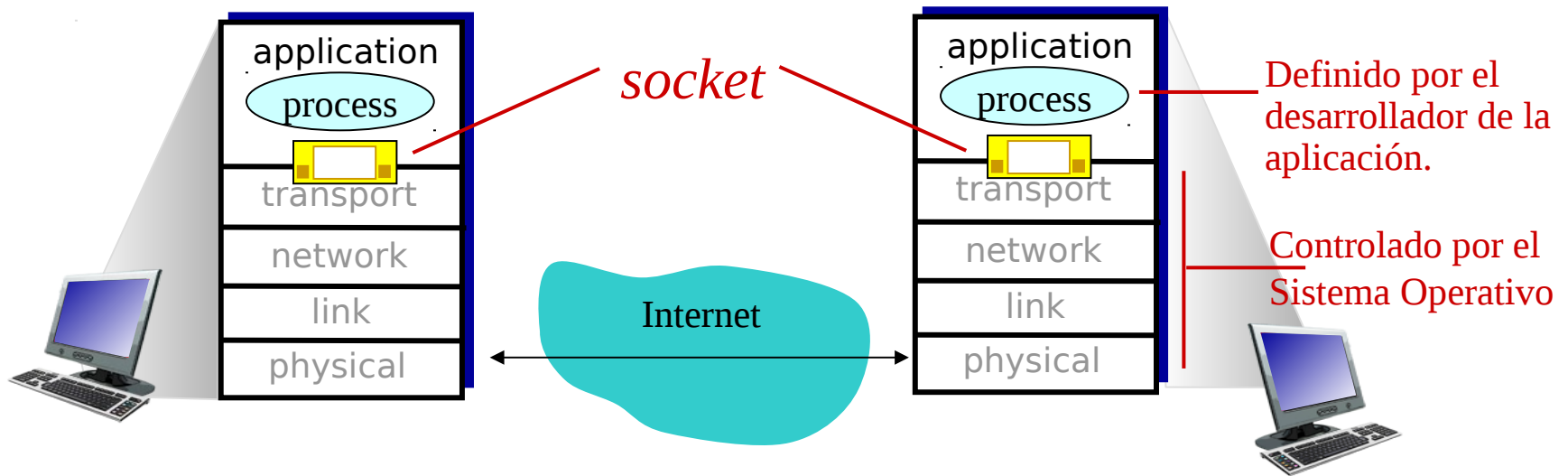
## sockets

Son *locales al host*,  
*creados por la aplicación*,  
Es una interfaz  
*controlada por el OS*  
(una “puerta”) a través  
de la cual el proceso  
aplicación puede **enviar**  
**y recibir** mensajes  
a/desde otro proceso  
remoto de la aplicación

# Programación de Sockets (2)

**objetivo:** aprender cómo construir aplicaciones cliente/servidor que se comunican usando sockets.

**socket:** puerta entre el proceso aplicación y el protocolo de transporte extremo a extremo (protocolo de capa de transporte)



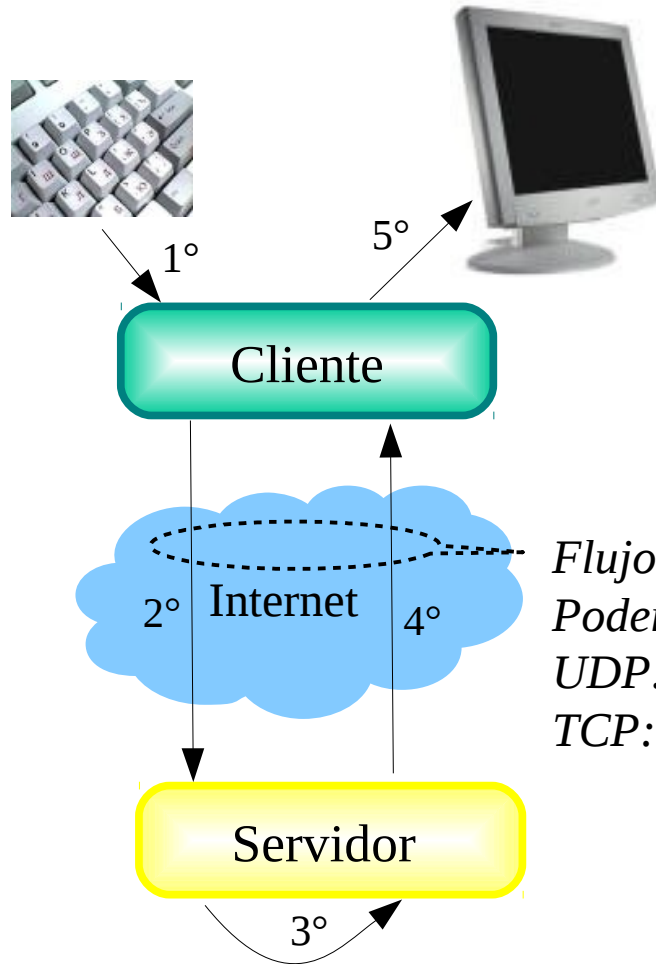
# Programación de Socket (3)

*Dos tipos de sockets para dos servicios de transporte:*

- **UDP:** datagrama (grupo de bytes) no confiable
- **TCP:** Confiable, flujo de bytes

*Ejemplo de aplicación:*

1.	Cliente lee una línea de caracteres (datos) desde su teclado y envía el texto al servidor.	
		2. El servidor recibe el texto y lleva sus letras a mayúscula.
		3. El servidor envía el texto modificado al cliente.
4.	El cliente recibe el texto modificado y muestra la línea en pantalla.	



*Flujos de ida y vuelta  
Podemos usar protocolo UDP o TCP  
UDP: sin conexión  
TCP: con conexión*

# Programación de Socket *con UDP* User Datagram Protocol

UDP: no hay “conexión” entre cliente y servidor

- ❑ No hay handshaking (establecimiento de conexión)
- ❑ Tx explícitamente adjunta dirección IP y puerto de destino en cada paquete.
- ❑ Para responder se debe extraer dirección IP y puerto del Tx desde el paquete recibido

UDP: datos transmitidos pueden llegar fuera de orden o perderse.

Congestión

Punto de vista de la aplicación

*UDP provee transferencia no confiable de grupos de bytes (“datagramas”) entre cliente y servidor*

*Distintos caminos al destino*

# Interacción Client/server vía socket UDP

## client

create socket:

`clientSocket =  
socket(AF_INET,SOCK_DGRAM)`



Create datagram with server IP and  
port=x; send datagram via `clientSocket`



read datagram from `clientSocket`



Close `clientSocket`

## server (running on serverIP)

create socket, port= x:

`serverSocket =  
socket(AF_INET,SOCK_DGRAM)`



read datagram from `serverSocket`



write reply to `serverSocket`  
specifying client address,  
port number





# Ejemplo aplicación: Cliente UDP

## *Python UDPClient*

Incluir biblioteca  
socket de Python

→ `from socket import *`

`serverName = 'hostname'`

`serverPort = 12000`

Crea socket UDP para  
cliente

→ `clientSocket = socket(AF_INET, SOCK_DGRAM)`

Obtiene entrada desde  
teclado

→ `message = raw_input('Input lowercase sentence:')`

Agregar nombre de servidor y  
puerto al mensaje; y lo envía  
usando socket

→ `clientSocket.sendto(message,(serverName, serverPort))`

`modifiedMessage, serverAddress =`

Lee en string la respuesta  
desde socket

→ `clientSocket.recvfrom(2048)`

Muestra string recibido y  
cierra el socket

→ `print modifiedMessage`

`clientSocket.close()`

# Ejemplo de aplicación: servidor UDP

## *Python UDP Server*

```
from socket import *  
serverPort = 12000  
Crea socket UDP → serverSocket = socket(AF_INET, SOCK_DGRAM)  
Vincula socket al número de  
puerto local 12000 → serverSocket.bind(("", serverPort))  
  
print "The server is ready to receive"  
  
Lazo infinito → while 1:  
Lee desde socket UDP el  
mensaje y dirección de cliente  
(IP y puerto) → message, clientAddress = serverSocket.recvfrom(2048)  
modifiedMessage = message.upper()  
Envía al cliente mensaje en  
mayúscula. → serverSocket.sendto(modifiedMessage, clientAddress)
```

# Programación de Sockets con TCP

## El cliente debe contactar al servidor

- ❑ Proceso servidor debe estar corriendo primero
- ❑ Servidor debe tener creado el socket (puerta) que acoge al cliente

## El cliente contacta al servidor a través de:

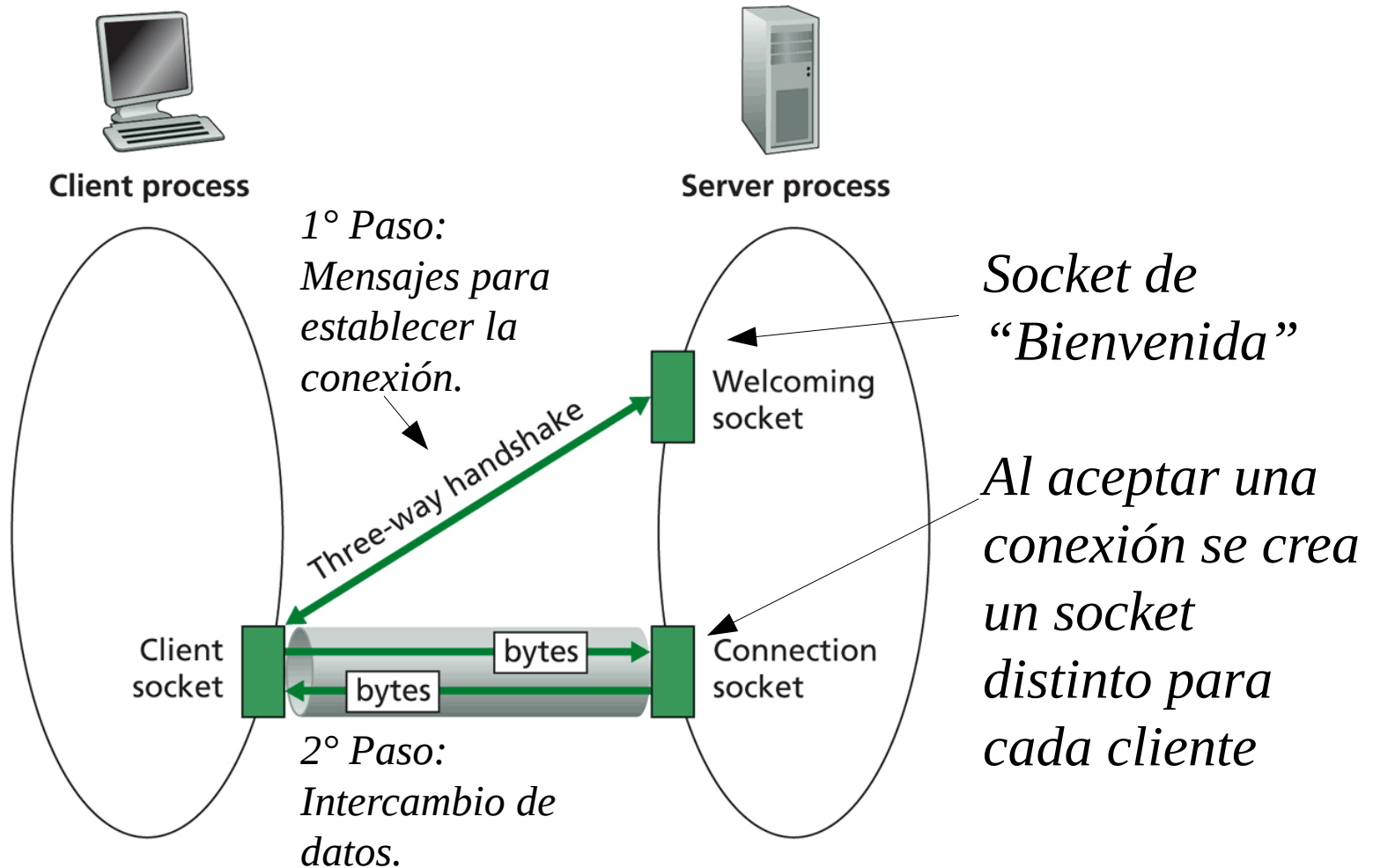
- ❑ La creación de un socket TCP local para el cliente
- ❑ Especifica la dirección IP, número de puerto del proceso servidor
- ❑ Una vez que el **cliente crea el socket**: el socket establece una conexión TCP al servidor

- ❑ Cuando el servidor es contactado por el cliente, el **servidor TCP crea otro socket** para que el proceso servidor se comunique con ese cliente, hay un socket por cliente
  - Permite que un servidor hable con múltiples clientes
  - IP y Número de puerto **fuente** (del cliente) distingue a cada cliente (**más adelante más sobre esto**)

## Punto de vista de la aplicación

*TCP provee transferencias de bytes confiables y en orden. Es un pipeline (o "tubería") de datos entre el cliente y servidor*

# Sockets creados en conexión cliente/servidor usando TCP



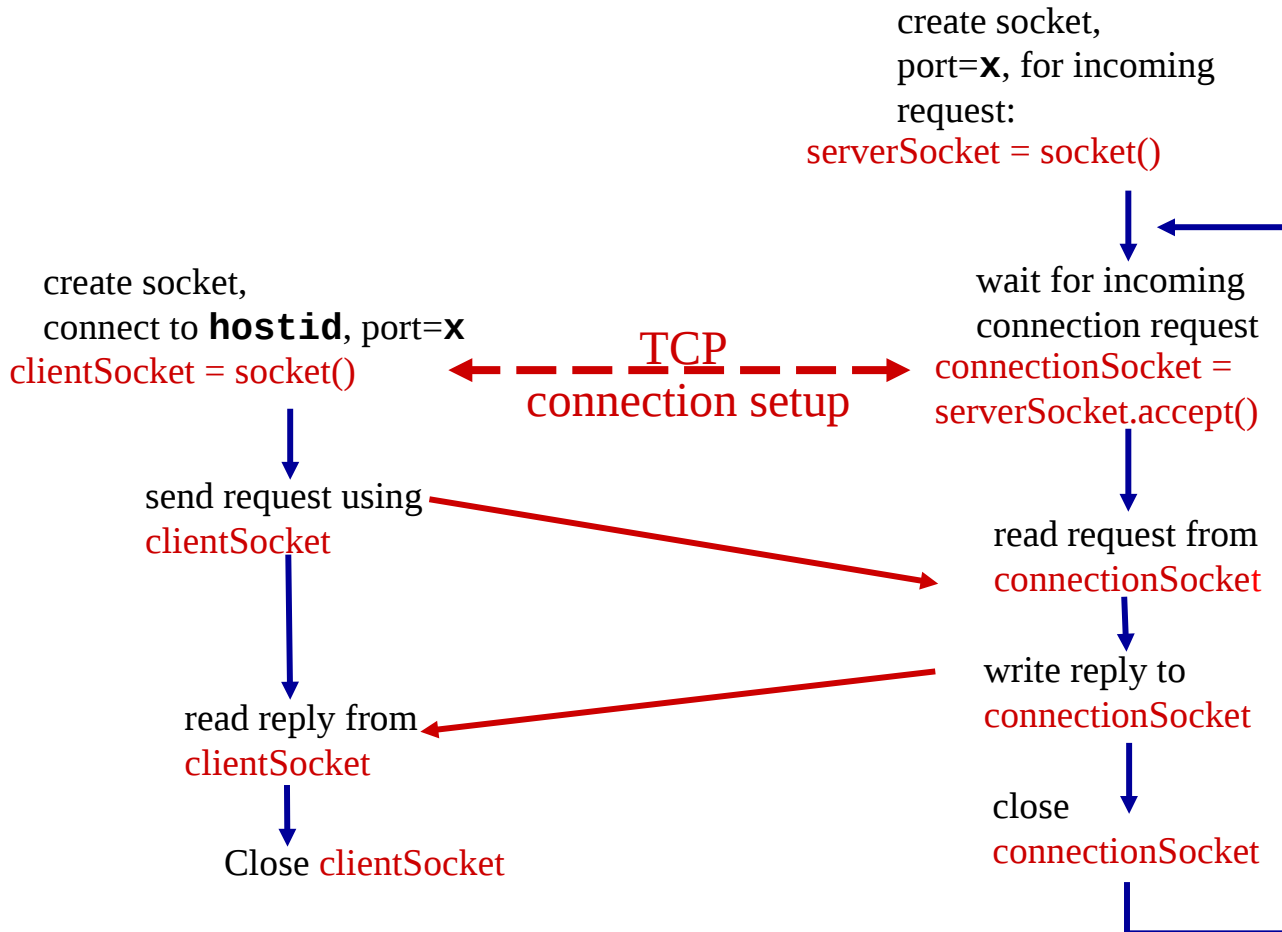
**Figure 2.27** ♦ Client socket, welcoming socket, and connection socket.

# Interacción Client/server entre sockets

## TCP

client

server (running on `hostid`)



# Ejemplo aplicación: cliente TCP

## *Python TCPClient*

```
from socket import *
```

```
serverName = 'servername'
```

```
serverPort = 12000
```

Crea socket para cliente

```
→ clientSocket = socket(AF_INET, SOCK_STREAM)
```

Conecta socket al nombre y  
puerto del servidor remoto  
12000

```
→ clientSocket.connect((serverName,serverPort))
```

```
sentence = raw_input('Input lowercase sentence:')
```

No se requiere agregar  
nombre y puerto del servidor

```
→ clientSocket.send(sentence)
```

```
modifiedSentence = clientSocket.recv(1024)
```

```
print 'From Server:', modifiedSentence
```

```
clientSocket.close()
```

# Ejemplo aplicación: servidor TCP

## *Python TCP Server*

Crea socket TCP de bienvenida  
TCP

```
from socket import *  
serverPort = 12000  
serverSocket = socket(AF_INET, SOCK_STREAM)  
serverSocket.bind(('', serverPort))
```

Servidor comienza a  
escuchar requerimientos de  
conexión

```
serverSocket.listen(1)  
print 'The server is ready to receive'
```

Lazo infinito

```
while 1:
```

Servidor espera en accept()  
por requerimientos de conexión,  
un nuevo socket es retornado  
para atender a ese cliente

```
connectionSocket, addr = serverSocket.accept()
```

Lee bytes desde el socket (no  
nos preocupamos por  
dirección como en UDP)

```
sentence = connectionSocket.recv(1024)
```

```
capitalizedSentence = sentence.upper()
```

Cierra conexión para este  
cliente (pero no socket de  
bienvenida)

```
connectionSocket.send(capitalizedSentence)  
connectionSocket.close()
```

# Resumen de Capa aplicación

## Hemos cubierto varias aplicaciones de red

- Arquitectura de la aplicaciones
  - cliente-servidor
  - P2P
  - híbridos
- Servicios requeridos por aplicaciones:
  - confiabilidad, ancho de banda, retardo
- Modelo de servicio de transporte en Internet
  - Confiable y orientada a la conexión: TCP
  - No confiable, datagramas: UDP
- Protocolos específicos:
  - HTTP
  - SMTP, POP, IMAP
  - DNS
- Aplicaciones P2P
- Programación de sockets



# Resumen de Capa aplicación

## Lo más importante aprendido sobre *protocolos*

- ❑ Intercambio de mensajes típicos requerimiento/respuesta:
  - cliente requiere info o servicio
  - servidor responde con datos, código de estatus
- ❑ Formato de mensajes:
  - **encabezado**: campos dando info sobre datos
  - **datos**: info siendo comunicada
- ❑ Centralizado vs. descentralizado
- ❑ Sin estado vs. con estado
- ❑ Transferencia confiable vs. Transferencia no confiable
- ❑ “la complejidad es puesta en los bordes de la red (en las aplicaciones)”  
Distinto a sistema telefónico clásico.