2 Modelo Cliente Servidor

2.2 Modelos y arquitecturas cliente - servidor

Modelos y arquitecturas cliente - servidor

Una definición **simple** …

"El software del servidor acepta peticiones de servicio desde el software del cliente, calcula el resultado y lo devuelve al cliente"

Modelos y arquitecturas cliente - servidor

Participantes

- Elementos de computación:
  - Cliente
  - Servidor
  - Red de interconexión

Modelos y arquitecturas cliente - servidor

Ejemplo: HTTP

1) Petición:

```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: nombre-cliente
[Línea en blanco]
```

2) Respuesta:

```
HTTP/1.1 200 OK
Date: Fri, 31 Dec 2003 23:59:59 GMT
Content-Type: text/html
Content-Length: 1221

<html>
<body>
 <h1>Página www.uc3m.es</h1>
(Contenido) . . .
</body>
</html>
```

Acceso distribuido vs. Computación distribuida

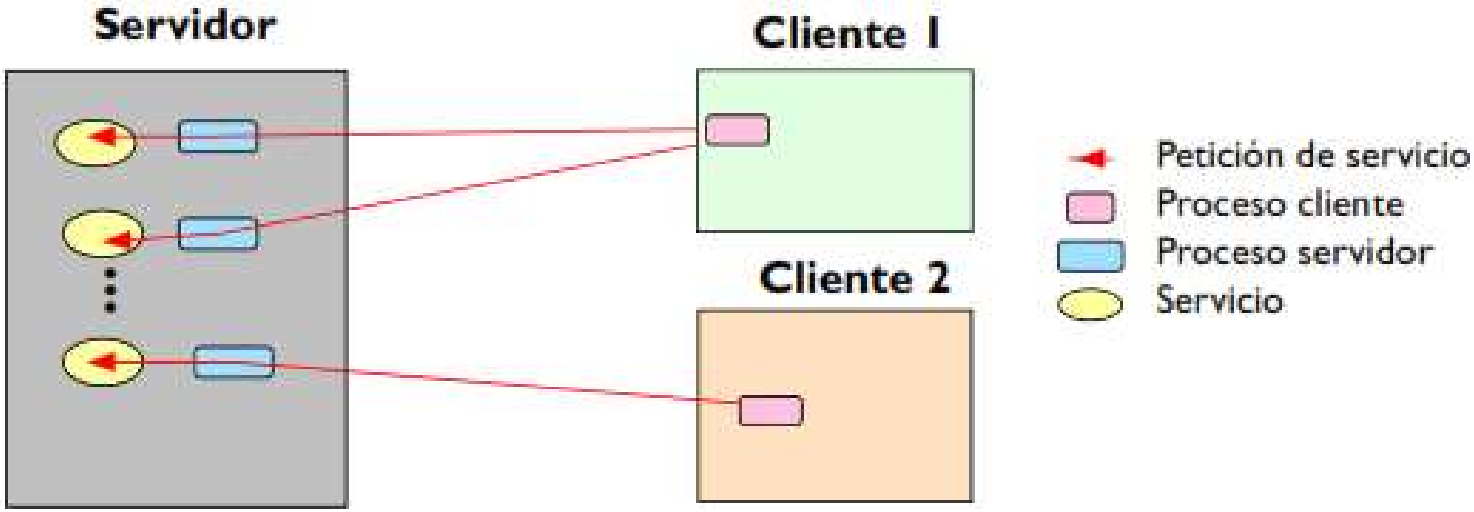
La computación de tipo cliente-servidor

es acceso distribuido

no computación distribuida !!!

Cliente - Servidor

- Asigna roles diferentes a los procesos que comunican: cliente y servidor
- Servidor:
  - Ofrece un servicio
  - Elemento pasivo: espera la llegada de peticiones
- Cliente:
  - Solicita el servicio
  - Elemento activo: invoca peticiones



Conceptos previos

- El modelo cliente-servidor es una abstracción eficiente para facilitar los servicios de red
- La asignación de roles asimétricos simplifica la sincronización
- Implementación mediante:
  - **Sockets**
  - Llamada a procedimientos remotos (**RPC**)
  - Invocación de **métodos remotos** (RMI, CORBA, …).
- Paradigma principalmente adecuado para servicios centralizados
- **Ejemplos**: servicios de Internet (HTTP, FTP, DNS, …)

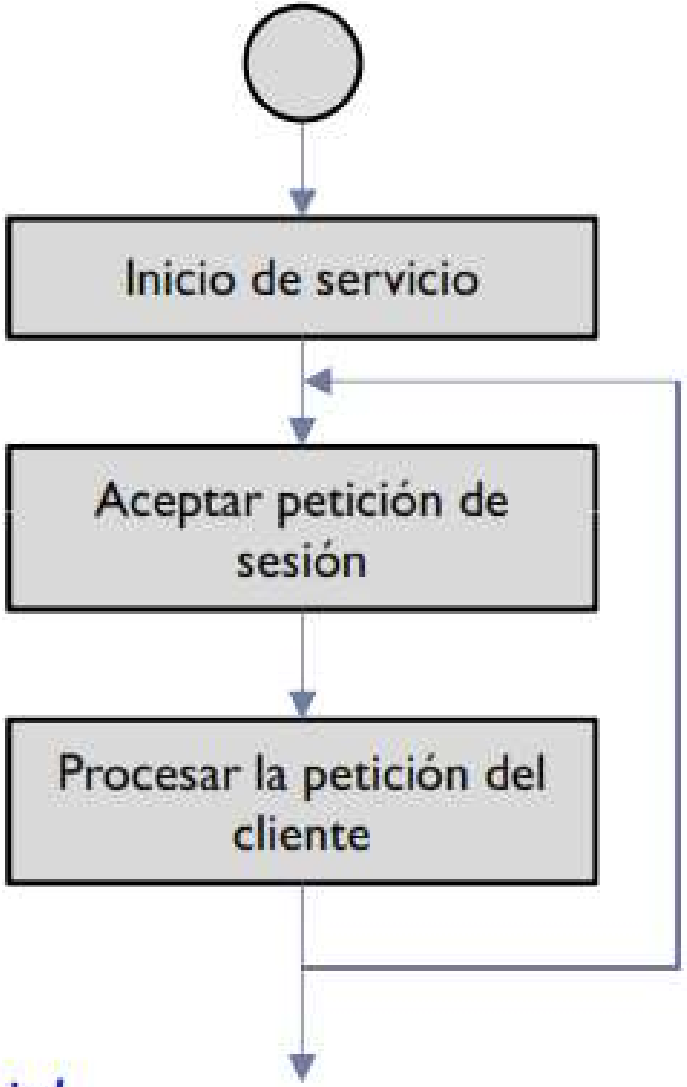
Tipos de servidores de aplicaciones

- En función del número de peticiones que es capaz de atender:
  - Secuencial: una petición
  - Concurrente: múltiples peticiones atendidas al mismo tiempo
- En función de si existe una conexión preestablecida con el cliente
  - Servidores orientados a conexión
  - Servidores NO orientados a conexión
- En función de si almacena o no el estado de la comunicación
  - Servidores con estado
  - Servidores sin estado

Modelo de servidor secuencial

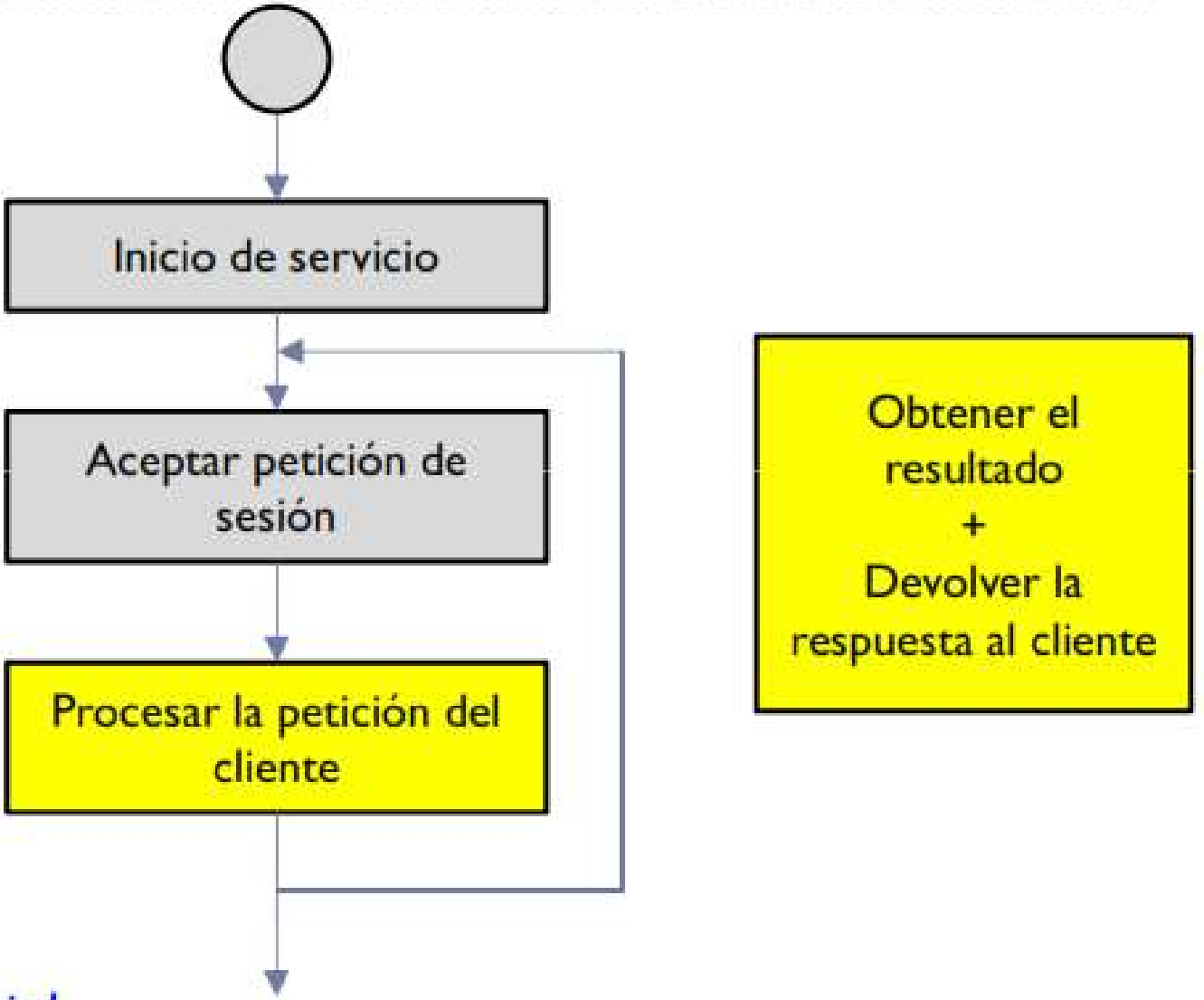
- El servidor sirve las peticiones de forma secuencial
- Mientras está atendiendo a un cliente **no** puede aceptar peticiones de más clientes

Flujo de ejecución de un servidor secuencial



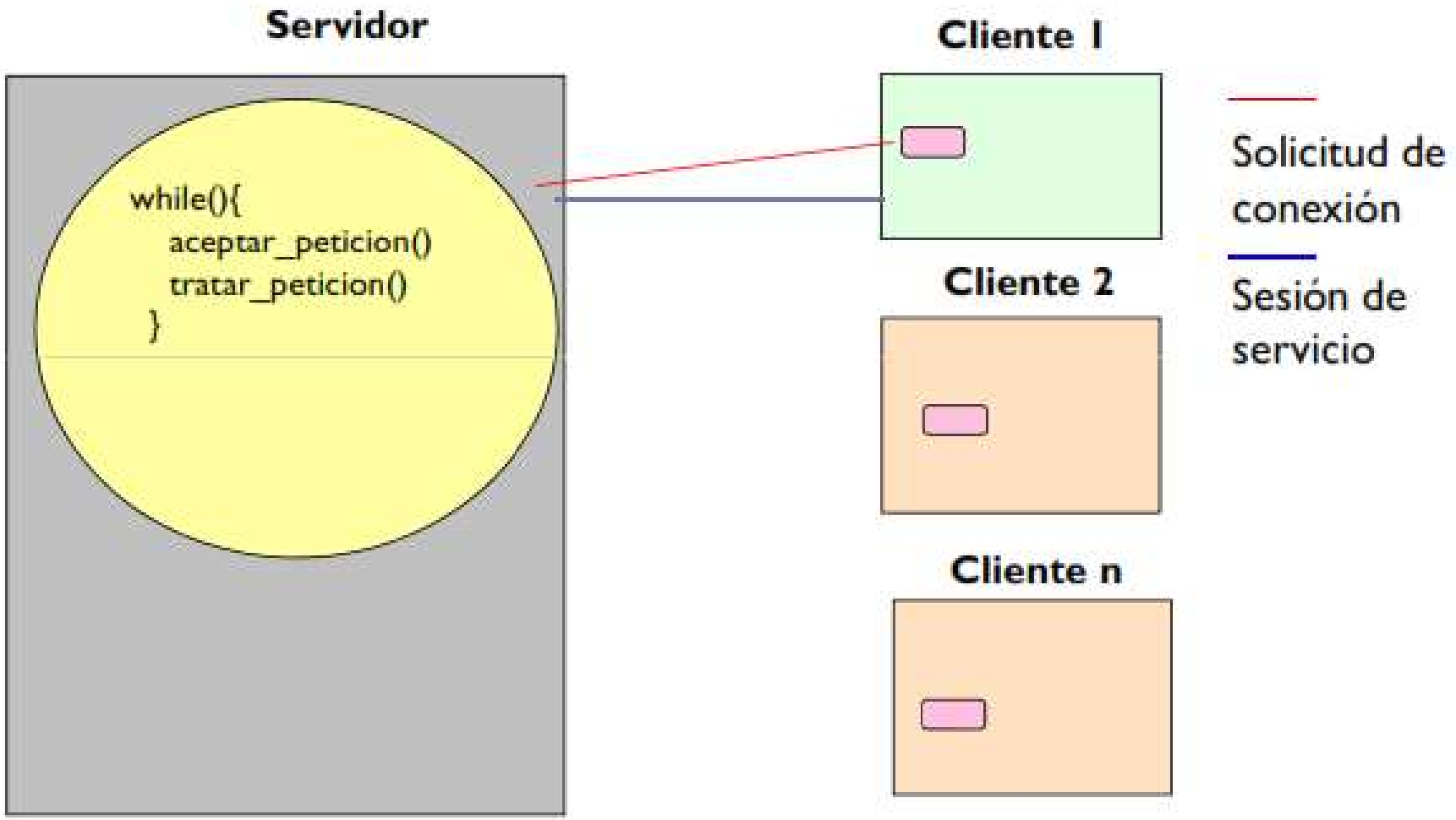
Servidor secuencial

Flujo de ejecución de un servidor secuencial

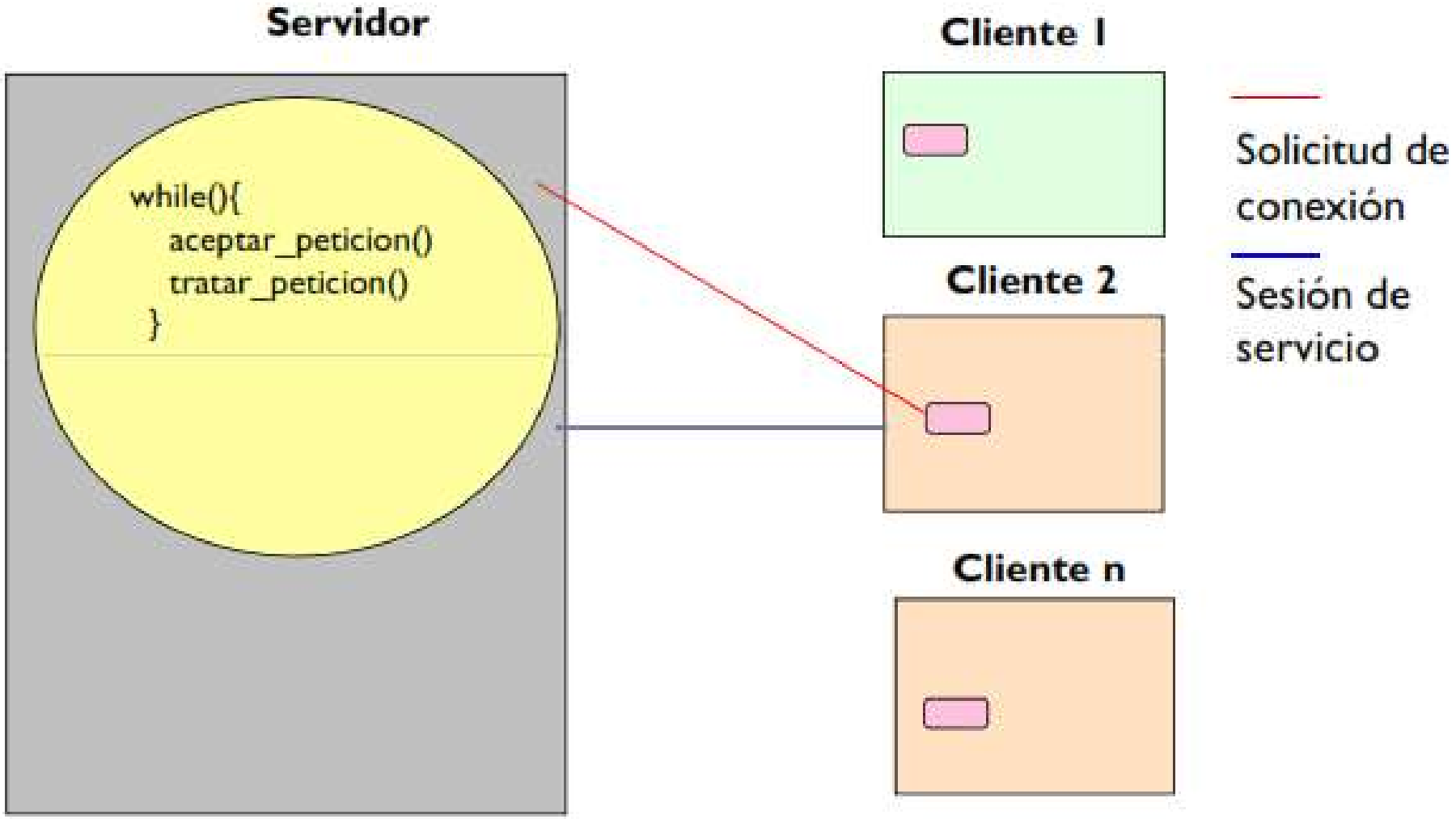


Servidor secuencial

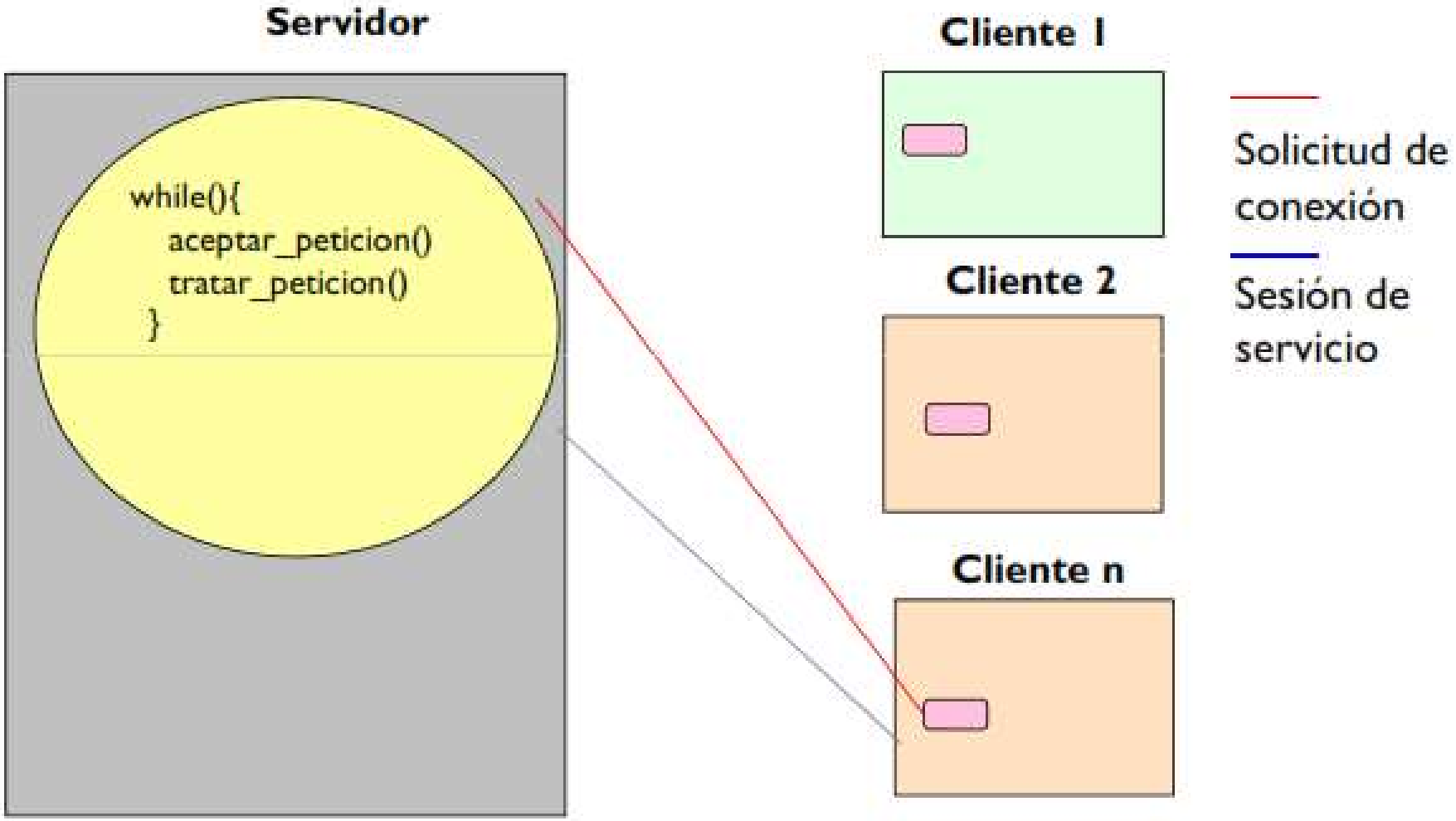
Cliente-Servidor secuencial



Cliente-Servidor secuencial

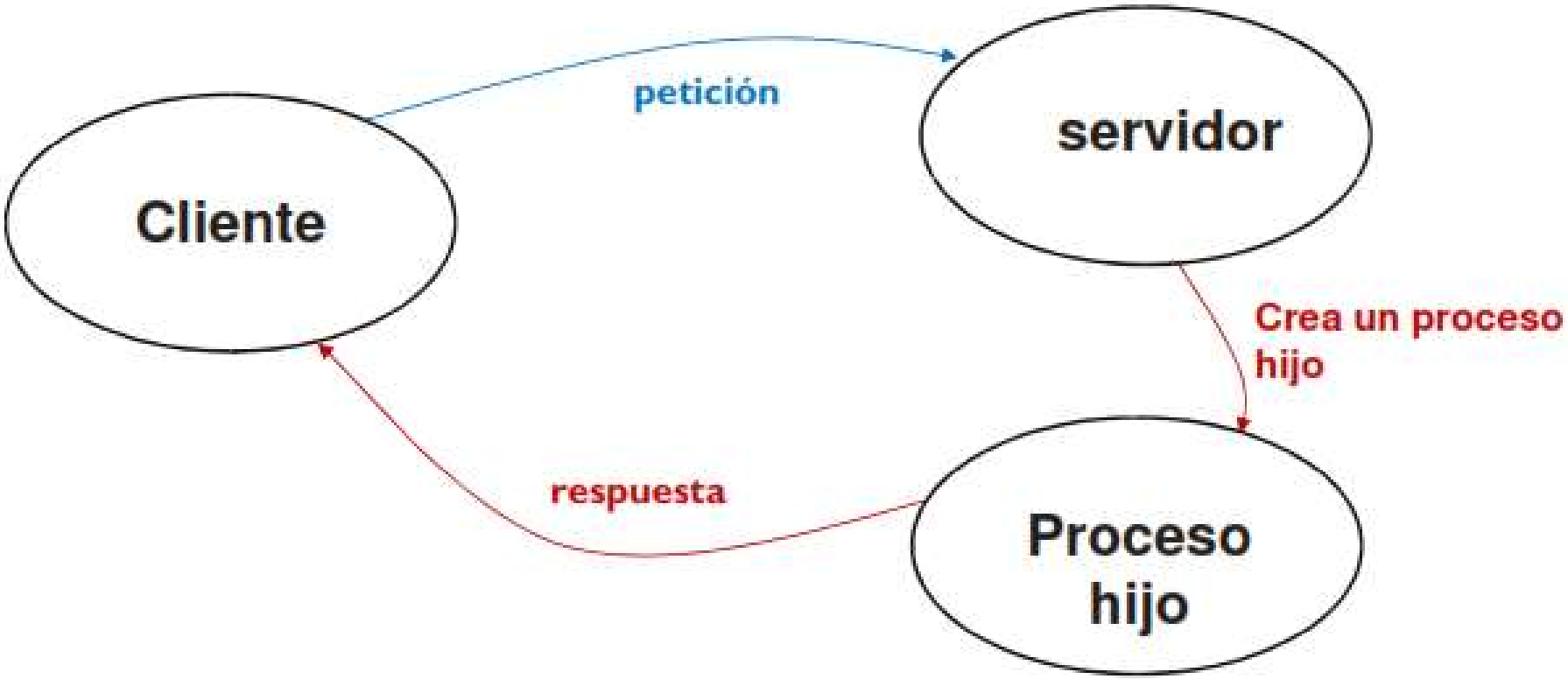


Cliente-Servidor secuencial

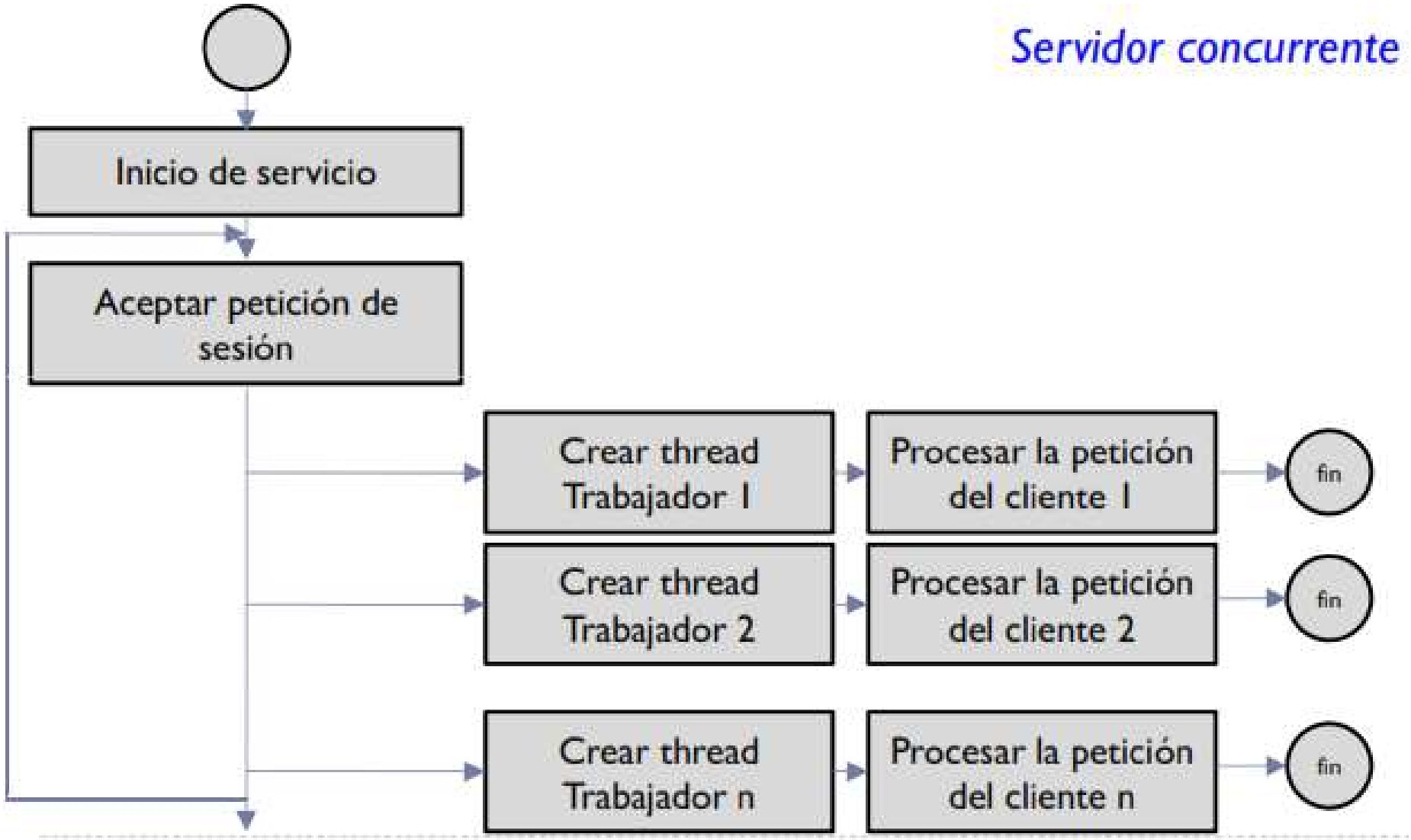


Modelo de servidor concurrente

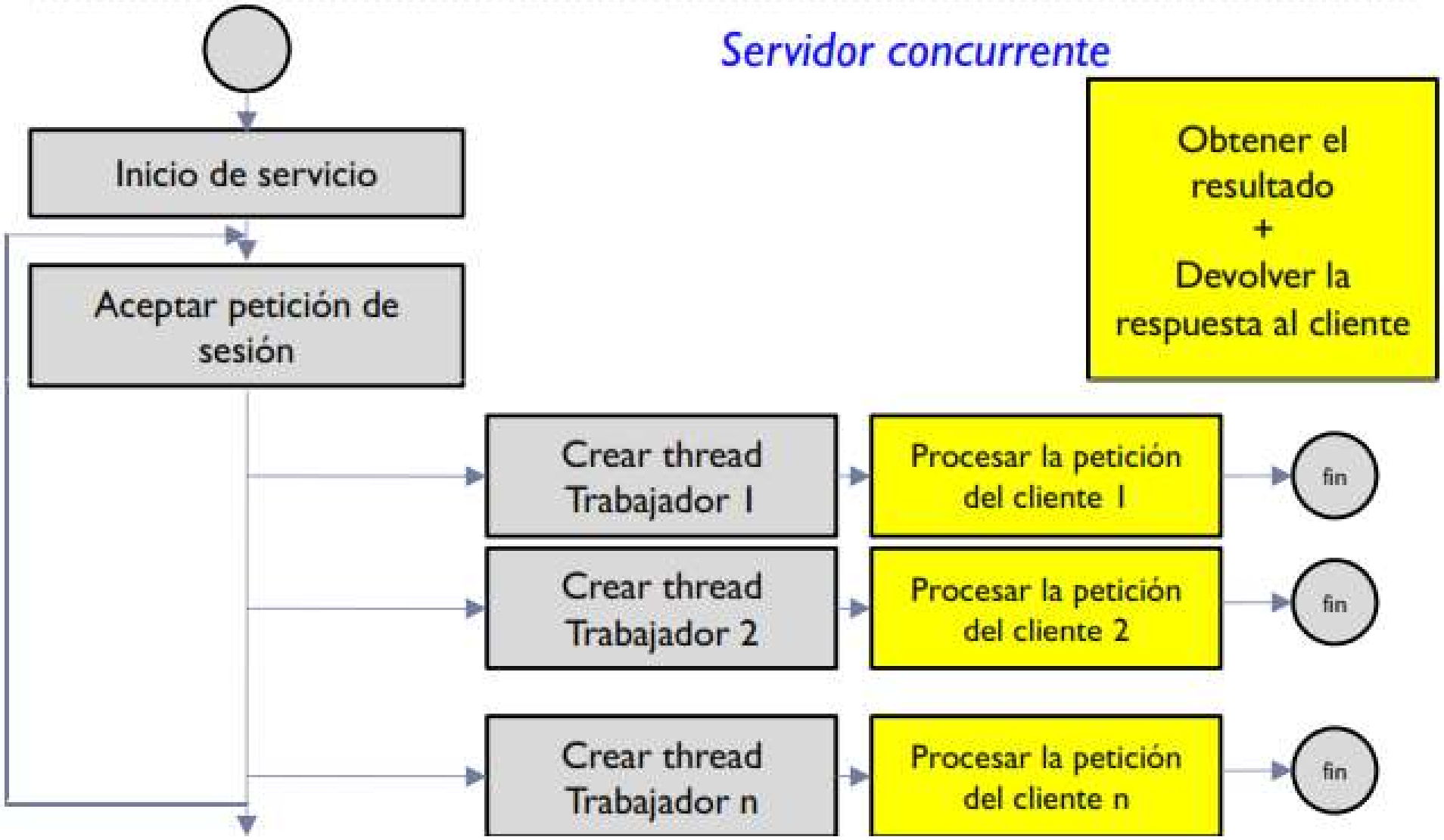
- El servidor crea un hijo que atiende la petición y envía la respuesta al cliente
- Se pueden atender múltiples peticiones de forma concurrente



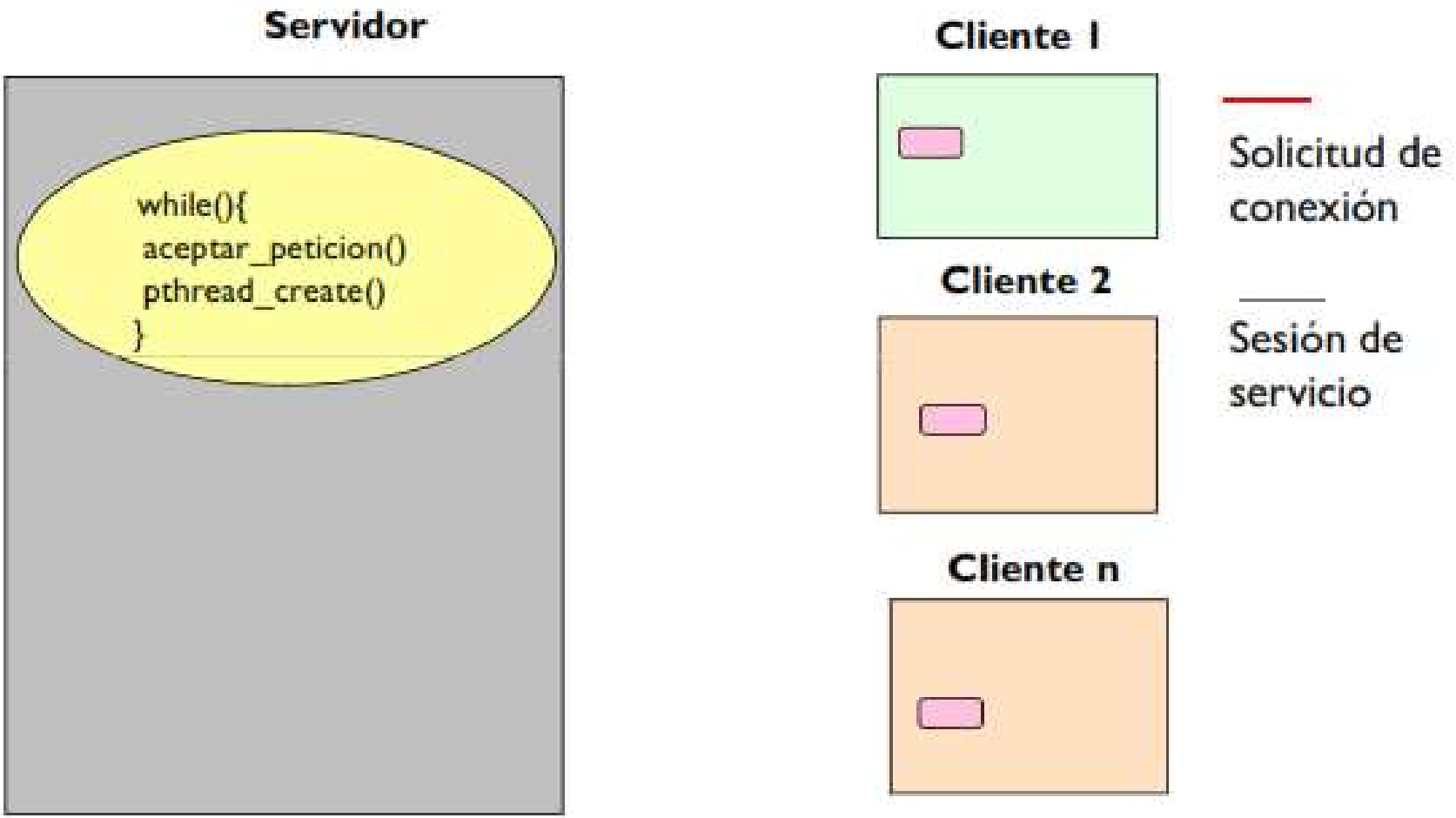
Flujo de ejecución de un servidor concurrente



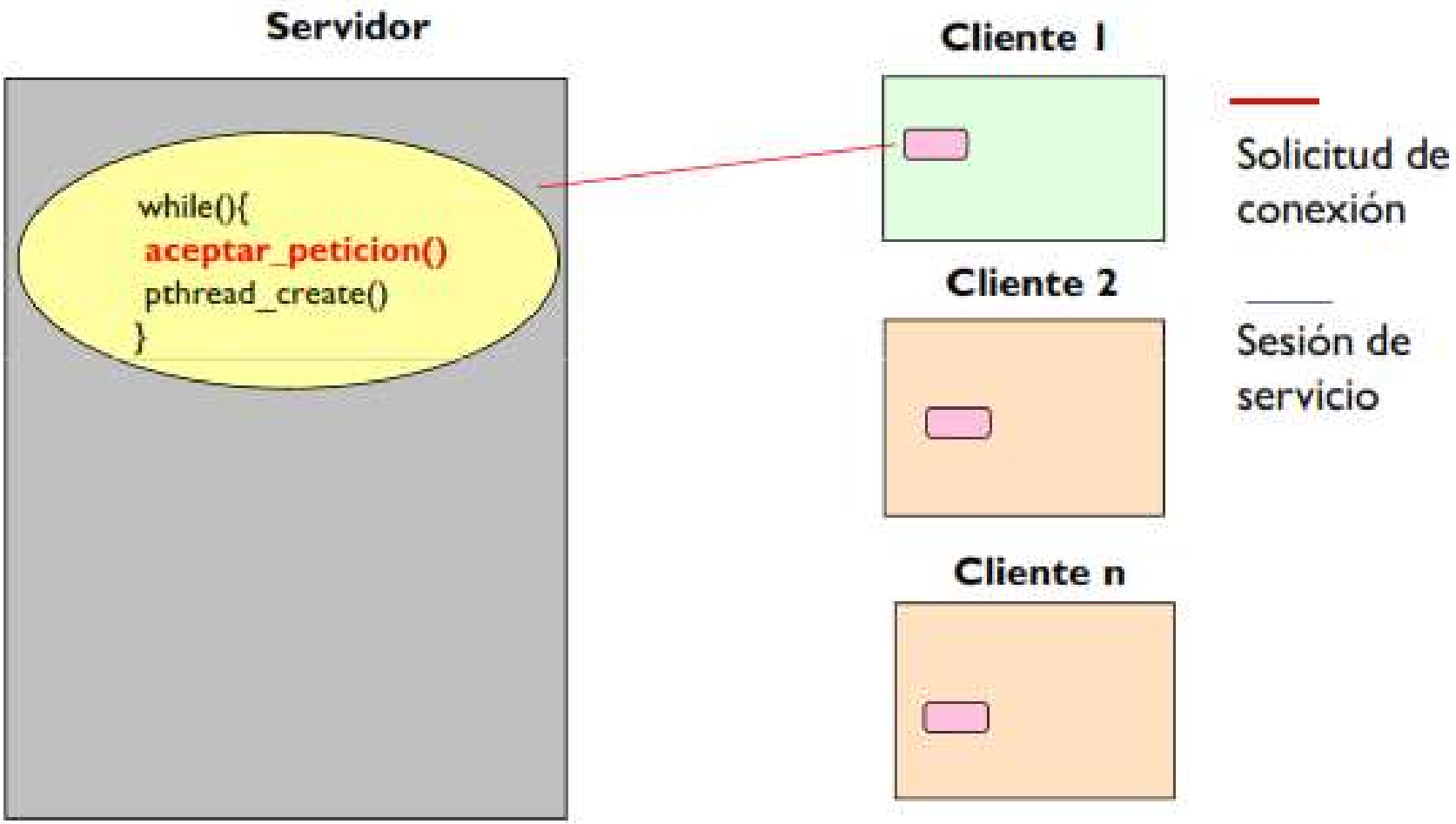
Flujo de ejecución de un servidor concurrente



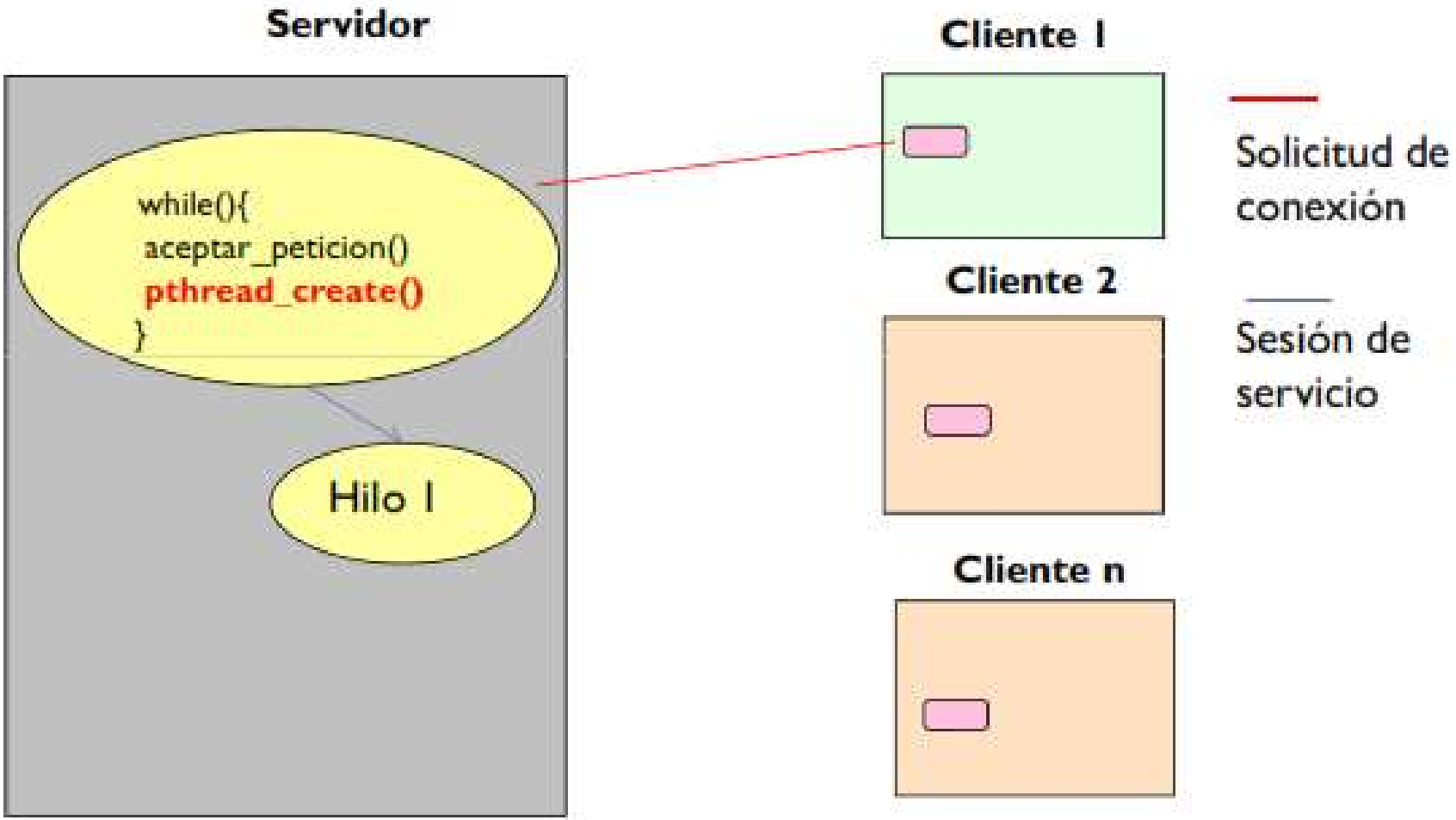
Cliente-Servidor concurrente



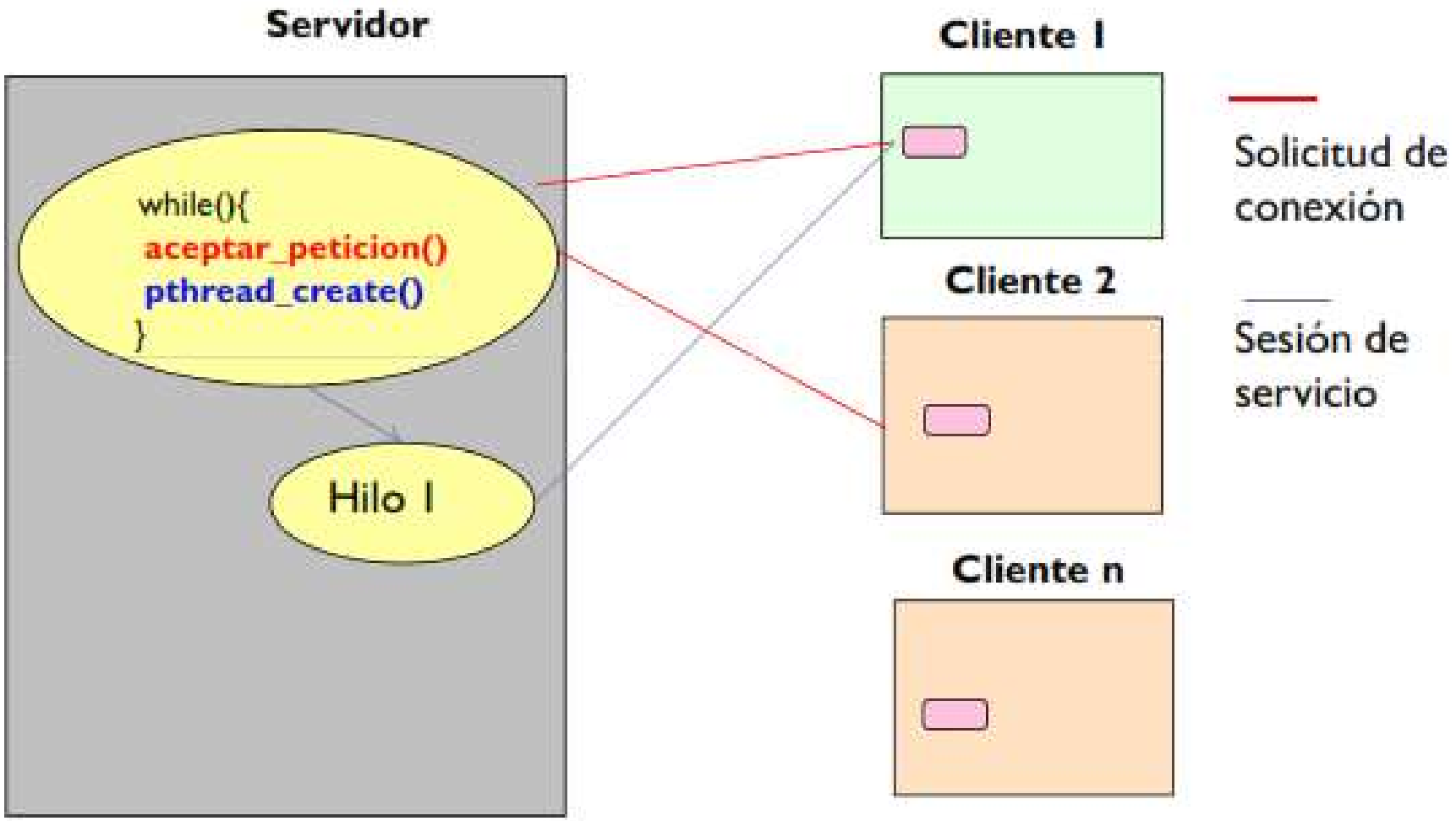
Cliente-Servidor concurrente



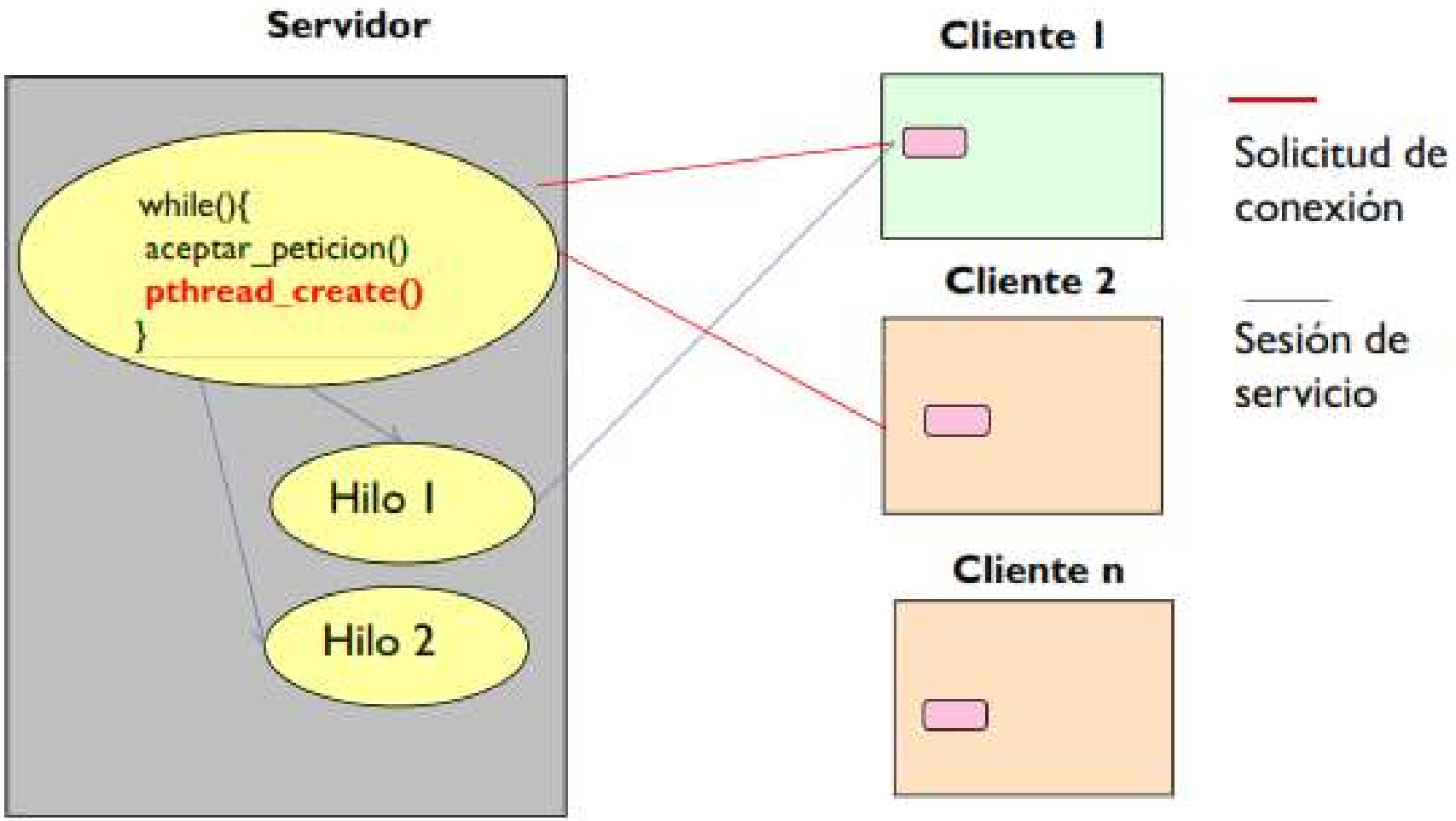
Cliente-Servidor concurrente



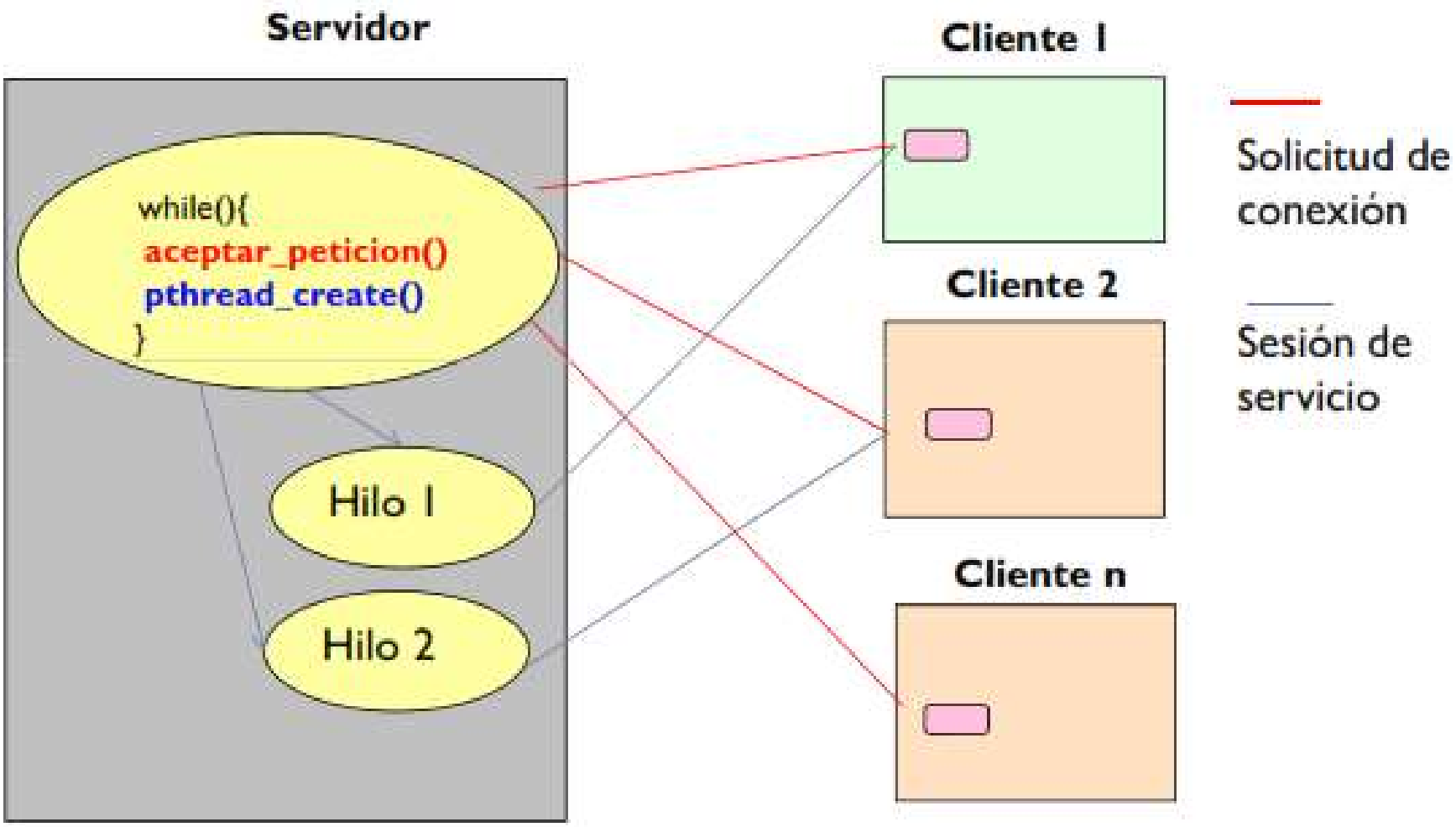
Cliente-Servidor concurrente



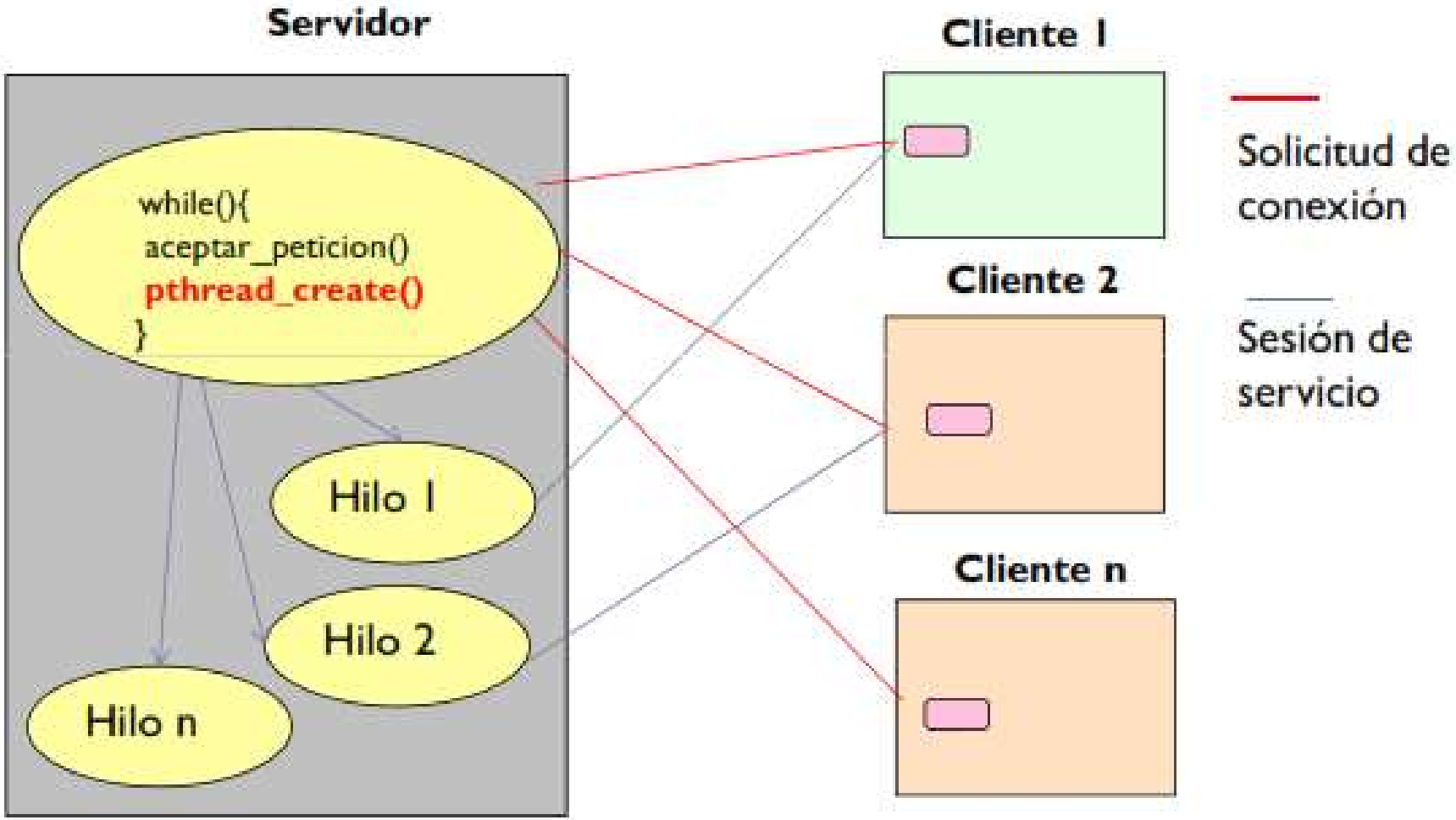
Cliente-Servidor concurrente



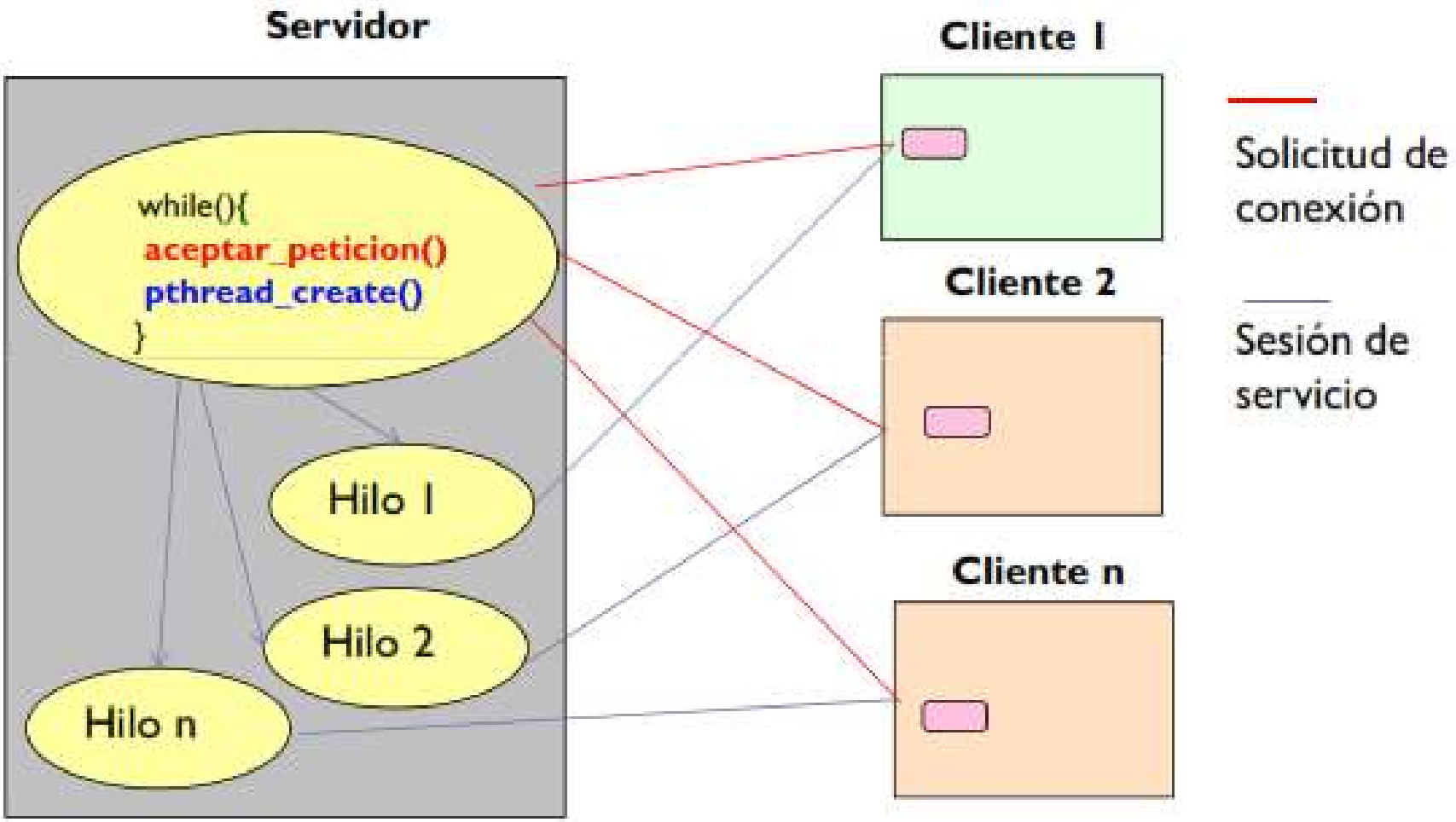
Cliente-Servidor concurrente



Cliente-Servidor concurrente

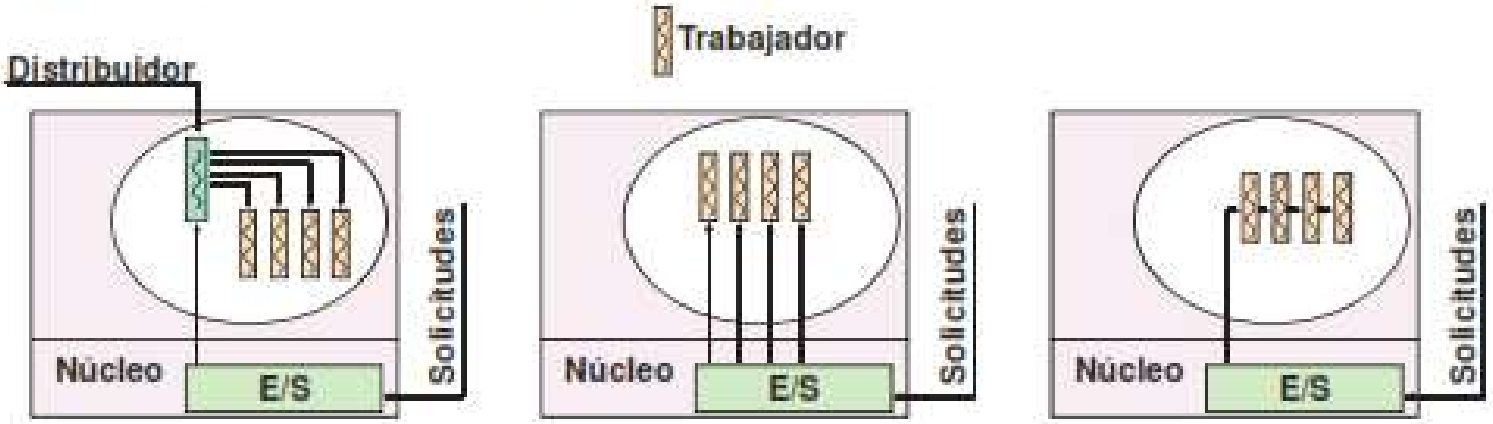


Cliente-Servidor concurrente



Diseño de servidores concurrentes mediante threads

- Distintas arquitecturas de SW para construir servidores paralelos:
  - Un proceso distribuidor que acepta peticiones y las distribuye entre un *pool* de procesos ligeros
  - Cada proceso ligero realiza las mismas tareas: aceptar peticiones, procesarlas y devolver su resultado
  - Segmentación: cada trabajo se divide en una serie de fases, cada una de ellas se procesa por un proceso ligero especializado



@Fuente: *Jesús Carretero, Félix García, Pedro de Miguel y Fernando Pérez.* **Mc Graw Hill**

Servidores orientados a conexión

- En un servicio orientado a conexión, el cliente y el servidor establecen una conexión (que puede ser lógica), posteriormente insertan o extraen datos desde esa conexión, y finalmente la liberan
  - El flujo de tráfico se representa mediante un identificador de conexión
- Los datos **no** incluyen información sobre la conexión establecida
  - Direcciones origen y destino
- Ejemplo: **TCP**

Servidores sin conexión

- En un protocolo **no orientado a conexión** los datos son intercambiados usando paquetes independientes, auto-contenidos, cada uno de los cuales necesita explícitamente la información de conexión
  - No existe acuerdo previo
- Ejemplo: **IP**, **UDP**

Concepto de sesión

- **Sesión**: Interacción entre cliente y servidor
- Cada cliente entabla una sesión separada e independiente con el servidor
  - El cliente conduce un diálogo con el servidor hasta obtener el servicio deseado
- El servidor ejecuta indefinidamente:
  - **Bucle continuo** para aceptar peticiones de las sesiones de los clientes
  - Para cada cliente el servidor conduce una sesión de servicio

Protocolo de servicio

- Se necesita un **protocolo** para especificar las reglas que deben observar el cliente y el servidor durante una sesión de servicio
  - En cada sesión el diálogo sigue un patrón especificado por el protocolo
  - Los protocolos de Internet están publicados en las RFCs

- Definición del protocolo de servicio:
  - Localización del servicio
  - Secuencia de comunicación entre procesos
  - Representación en interpretación de los datos

Tipos de servidores

- Servidores **sin estado:**
  - Cada mensaje de petición y respuesta es independiente de las demás
  - Ejemplo: HTTP
- Servidores **con estado:**
  - Debe mantener información de estado (por ej. anteriores conexiones de clientes) para proporcionar su servicio
  - Cada petición/respuesta puede depender de otras anteriores
  - Ejemplo: Telnet

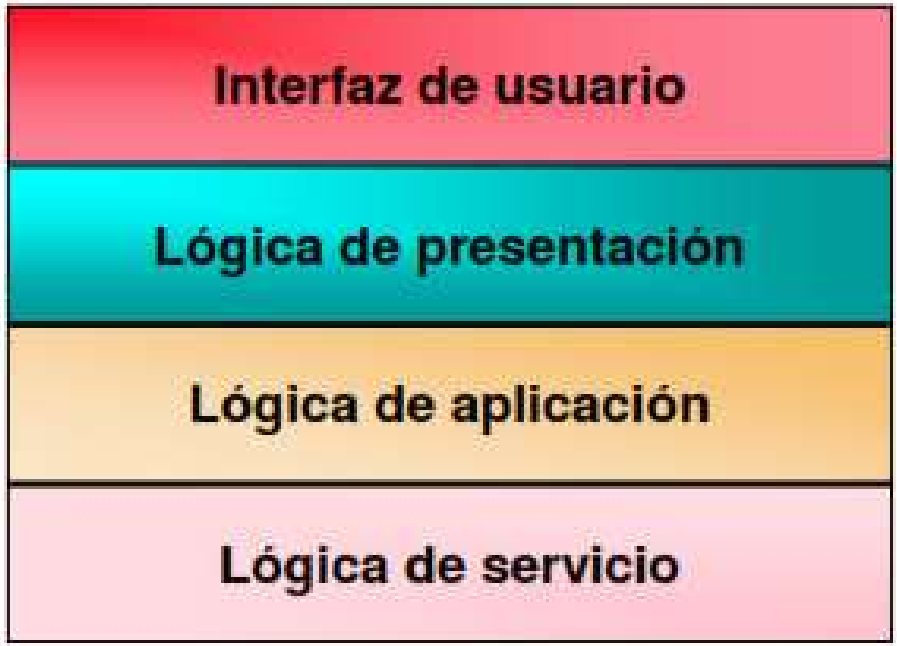
Información de estado

- Información de estado global
  - El servidor mantiene información para todos los clientes durante la vida del servidor
  - Ejemplo: servidor de tiempo
- Información de estado de sesión
  - El servidor mantiene información específica para cada sesión iniciada por los clientes
  - Ejemplo: **FTP** (File Transfer Protocol)

Arquitectura de SW

- La **arquitectura de SW** de una aplicación cliente-servidor consta de tres niveles:
  - Nivel de presentación: cliente y servidor precisan una interfaz de usuario
  - Nivel de lógica de aplicación: en el lado del servidor necesita procesarse la petición del cliente, calcular el resultado y devolverlo al cliente. En el lado del cliente se necesita enviar al servicio la petición del usuario y procesar el resultado (por ejemplo, mostrarlo por pantalla)
  - Nivel de servicio: los servicios requeridos para dar soporte a la aplicación son (1) en el servidor aquellos que permiten procesar la petición y 2) el mecanismo de IPC

Arquitectura de las Aplicaciones



¿Dónde se ejecutan las tareas?

- En el software del cliente (lado del cliente)
- En el software del servidor (lado del servidor)

Responsabilidades en el cliente

- Cliente:
  - Genera un mensaje de petición de servicio
  - Se conecta al servidor (dirección IP y puerto) [Solo orientado a conexión]
  - Envía el mensaje de petición de servicio
  - Espera por la respuesta
  - Procesa la respuesta: imprimir, almacenar, etc.
  - Desconexión [Solo orientado a conexión]

Responsabilidades en el servidor

- Servidor:
  1. Espera conexiones entrantes de los clientes
     - Una conexión entrante es una petición de servicio
  2. Por cada conexión:
     - Genera un *thread* de servicio [Solo servidores concurrentes]
     - El proceso principal:
       - Vuelve a esperar por nuevas conexiones entrantes
     - El *thread* de servicio:
       1. Procesa la petición
       2. Calcula el resultado
       3. Devuelve la respuesta al cliente
       4. Finaliza su ejecución

2.3 API Sockets

Sockets: introducción

- Mecanismo de IPC que proporciona comunicación entre procesos que ejecutan en máquinas distintas
- La primera implementación apareció en 1983 en UNIX BSD 4.2
  - Objetivo: incluir TCP/IP en UNIX
  - Diseño independiente del protocolo de comunicación

- Un **socket** es un descriptor de un punto final de comunicación:
  - dirección IP
  - puerto

- Abstracción que:
  - Representa un extremo de una comunicación bidireccional con una dirección asociada
  - Ofrece **interfaz de acceso a la capa de transporte** del protocolo TCP/IP
    - Protocolo TCP y UDP

Sockets: introducción

- API formalmente especificado en el estándar POSIX.1g (2000)
- Actualmente disponibles en:
  - Casi todos los sistemas UNIX
  - Prácticamente todos los sistemas operativos
    - WinSock: API de sockets de Windows
    - Macintosh
  - En Java como clase nativa

Sockets: UNIX

- Dominios de comunicación
  - AF_UNIX
  - AF_INET
  - AF_INET6
- Tipos de sockets
  - **Stream** (SOCK_STREAM)
  - Datragrama (SOCK_DGRAM)
- Direcciones de sockets

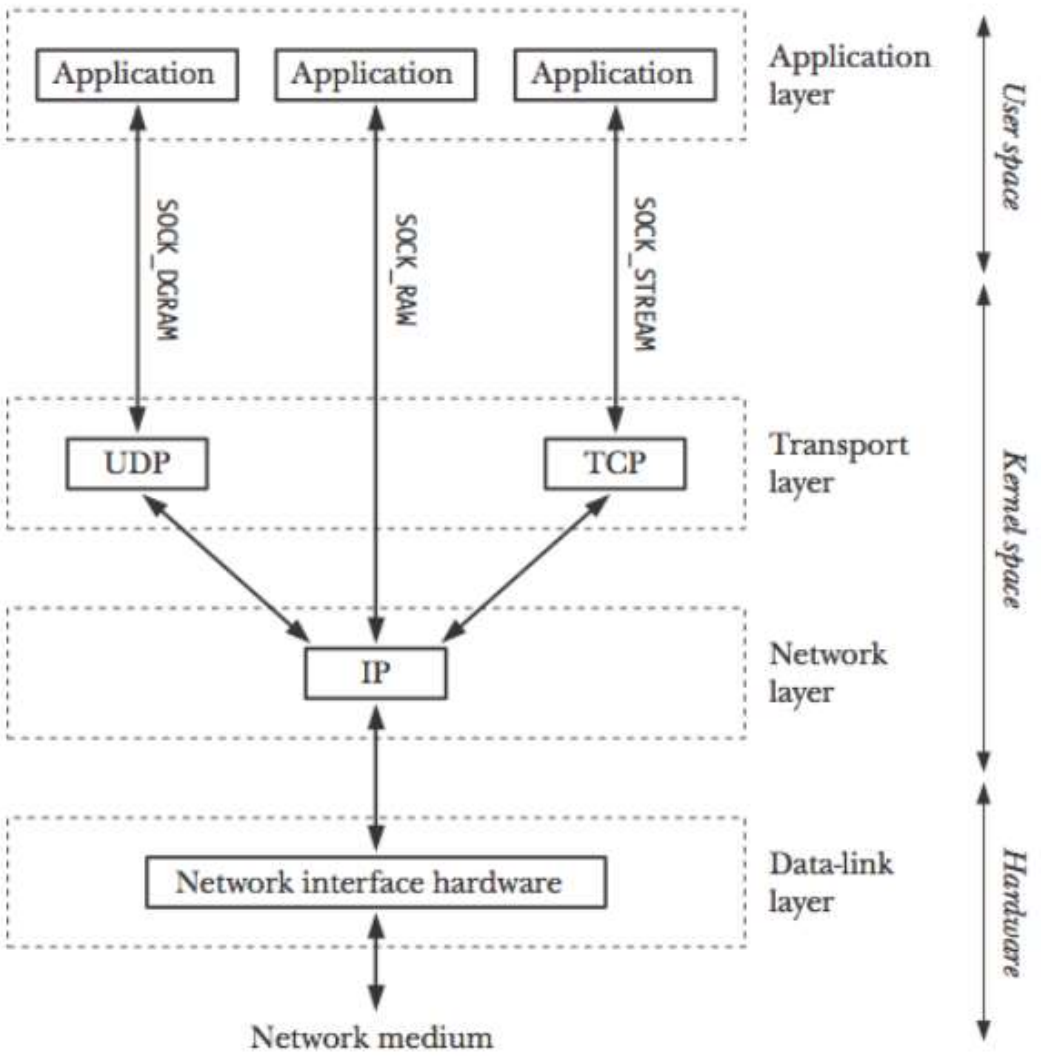
Dominios de comunicación

- Un dominio representa una familia de protocolos
  - Un socket está asociado a un dominio desde su creación
  - Sólo se pueden comunicar sockets del mismo dominio
  - Los servicios de sockets son independientes del dominio

- Ejemplos de dominios:
  - AF_UNIX: comunicación dentro de una máquina
  - AF_INET: comunicación usando protocolos TCP/IP (IPv4)
  - AF_INET6: comunicación usando protocolos TCP/IP (IPv6)

Tipos de sockets

- **Stream** (SOCK_STREAM)
  - Protocolo TCP
- **Datagrama** (SOCK_DGRAM)
  - Protocolo UDP
- **Raw** (SOCK_RAW)
  - Sockets sin protocolo de transporte (IP)



© The Linux Programming Interface. M. Kerrisk

Sockets **streams**

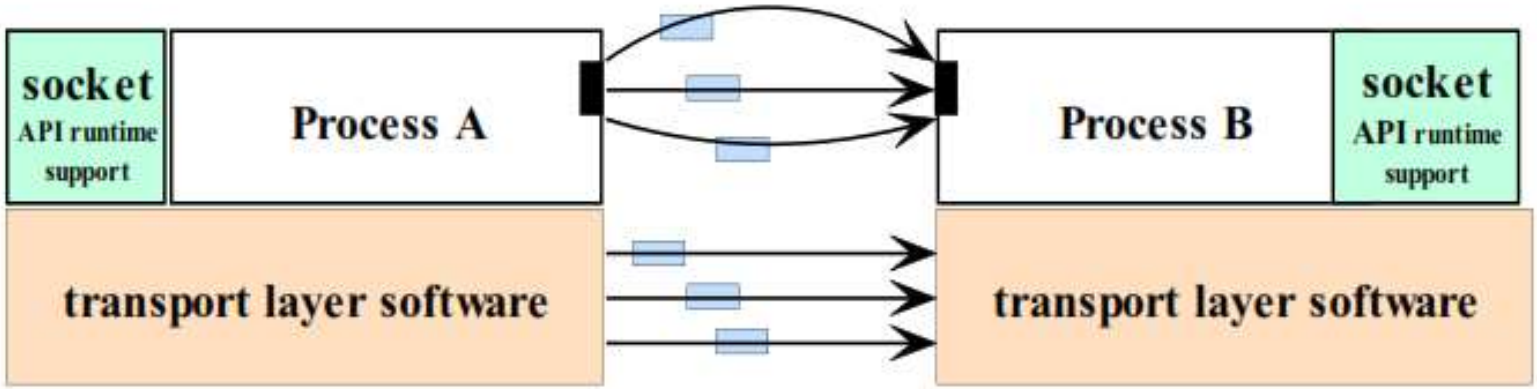
- Protocolo TCP
- Flujo de datos bidireccional
- Orientado a conexión
  - Debe establecerse una conexión extremo-a-extremo antes del envío y recepción de datos
  - Flujo de bytes (no preserva el límite entre mensajes)
  - Proporciona fiabilidad
    - Paquetes ordenados por secuencia, sin duplicación de paquetes, libre de errores, notifica errores

- Ejemplos:
  - HTTP, Telnet, FTP, SMTP

Sockets **datagrama**

- Protocolo **UDP**
- Flujo de datos bidireccional
- No orientado a conexión
  - No se establece/mantiene una conexión entre los procesos que comunican
  - Un datagrama es una entidad autocontenida
    - Se preservan los límites entre mensajes
  - Longitud máxima de un datagrama (datos **y** cabeceras) es **64 KB**
    - Cabecera IP+cabecera UDP = 28 bytes
  - Mantiene separación entre paquetes
  - No proporcionan fiabilidad
    - Paquetes desordenados, duplicados, pérdidas
- Ejemplos:
  - DNS

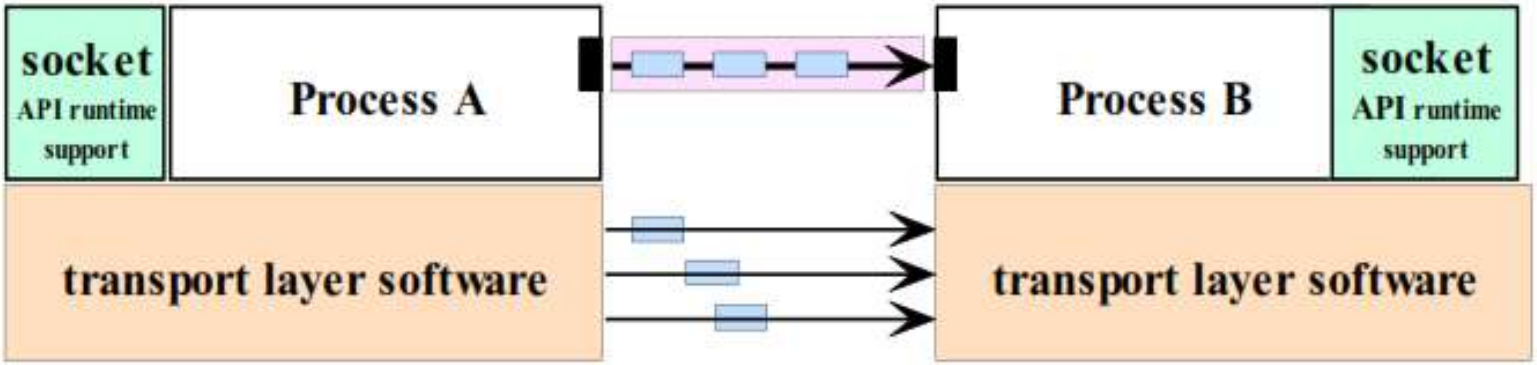
Sockets **stream y datagramas**



connectionless datagram socket

a datagram

a logical connection created and maintained by the runtime support of the datagram socket API



connection-oriented datagram socket

Direcciones de sockets

- Las direcciones se usan para:
  - Asignar una dirección local a un socket (bind)
  - Especificar una dirección remota (connect o sendto)

- Las direcciones son dependientes del dominio
  - Se utiliza la estructura genérica struct sockaddr
  - Cada dominio usa una estructura específica
    - Direcciones en AF_UNIX (struct sockaddr_un)
      - Nombre de fichero
    - Direcciones en AF_INET(struct sockaddr_in)
      - Cada socket debe tener asignada una dirección única
        - Dirección de host (32 bits) + puerto (16 bits) + protocolo

  - Es necesario la conversión de tipos (*casting*) en las llamadas

Puertos

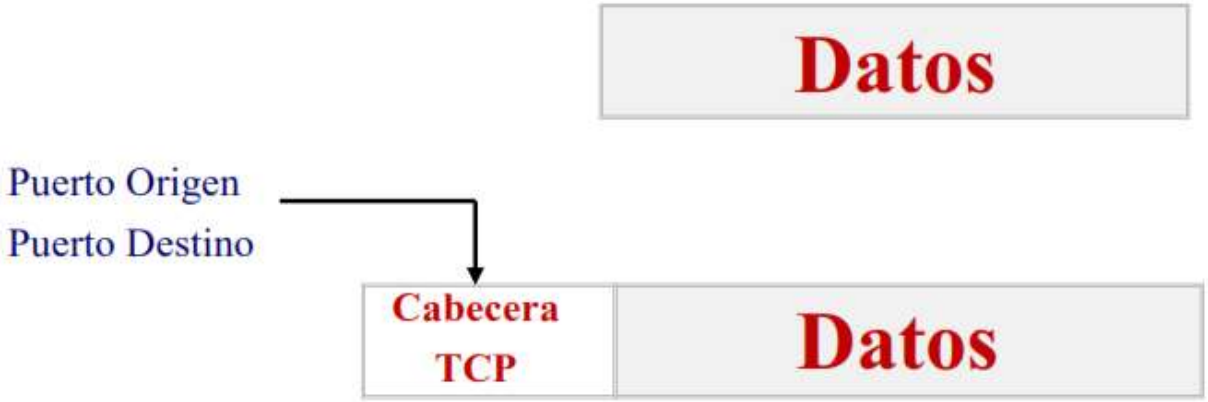
- Un puerto identifica un destino en un computador
  - Los puertos se asocian a procesos,
    - permiten que la transmisión se dirija a un proceso específico en el computador destino
  - Un puerto tiene un único receptor y múltiples emisores (excepto *multicast*)
  - Toda aplicación que desee enviar y recibir datos debe abrir un puerto

  - Número entero de 16 bits
    - 2^{16} puertos en una máquina ~ 65536 puertos posibles
    - Reservados por la IANA para aplicaciones de Internet: 0-1023 (también llamados ***well-known*** puertos)
    - Puertos entre 1024 y 49151 son puertos registrados para ser usados por los servicios
    - Puertos por encima de 65535 para uso privado

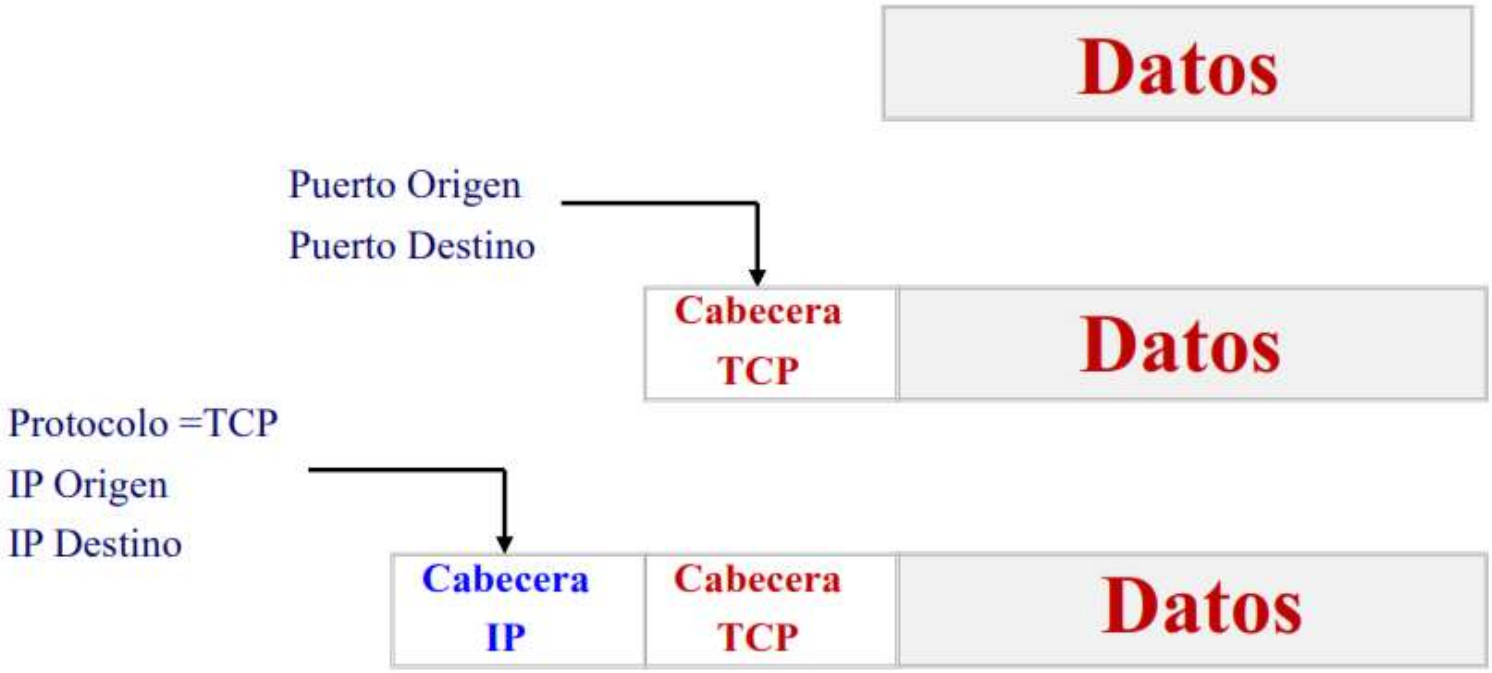
  - El espacio de puertos para streams y datagramas es independiente
    - http://www.iana.org/assignments/port-numbers

Encapsulación de un paquete TCP

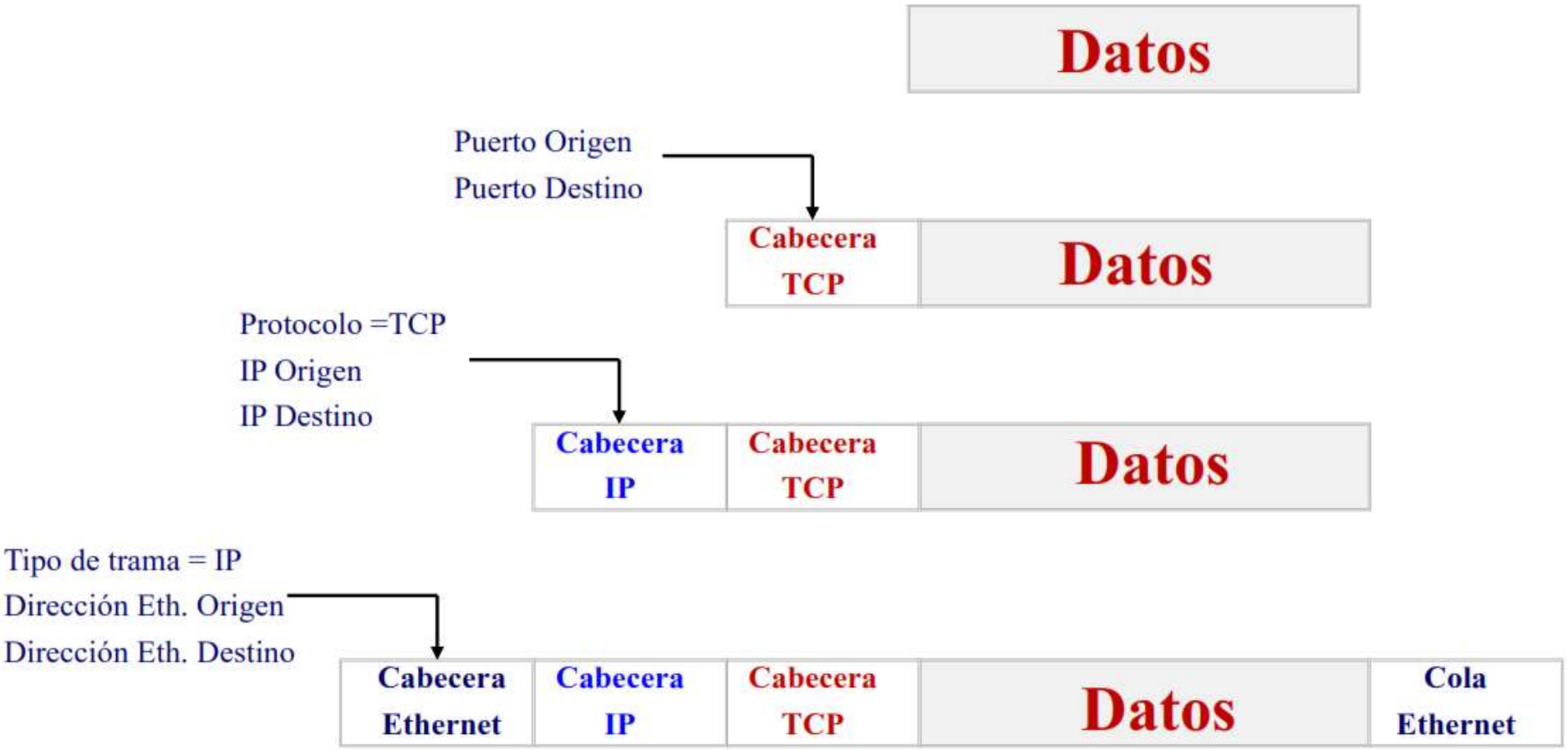
Encapsulación de un paquete TCP



Encapsulación de un paquete TCP



Encapsulación de un paquete TCP



Comparación de Protocolos

	IP	UDP	TCP
¿Orientado a conexión?	No	No	Si
¿Limite entre mensajes?	Si	Si	No
¿Ack?	No	No	Si
¿Timeout y retransmisión?	No	No	Si
¿Detección de duplicación?	No	No	Si
¿Secuenciamiento?	No	No	Si
¿Flujo de control?	No	No	Si

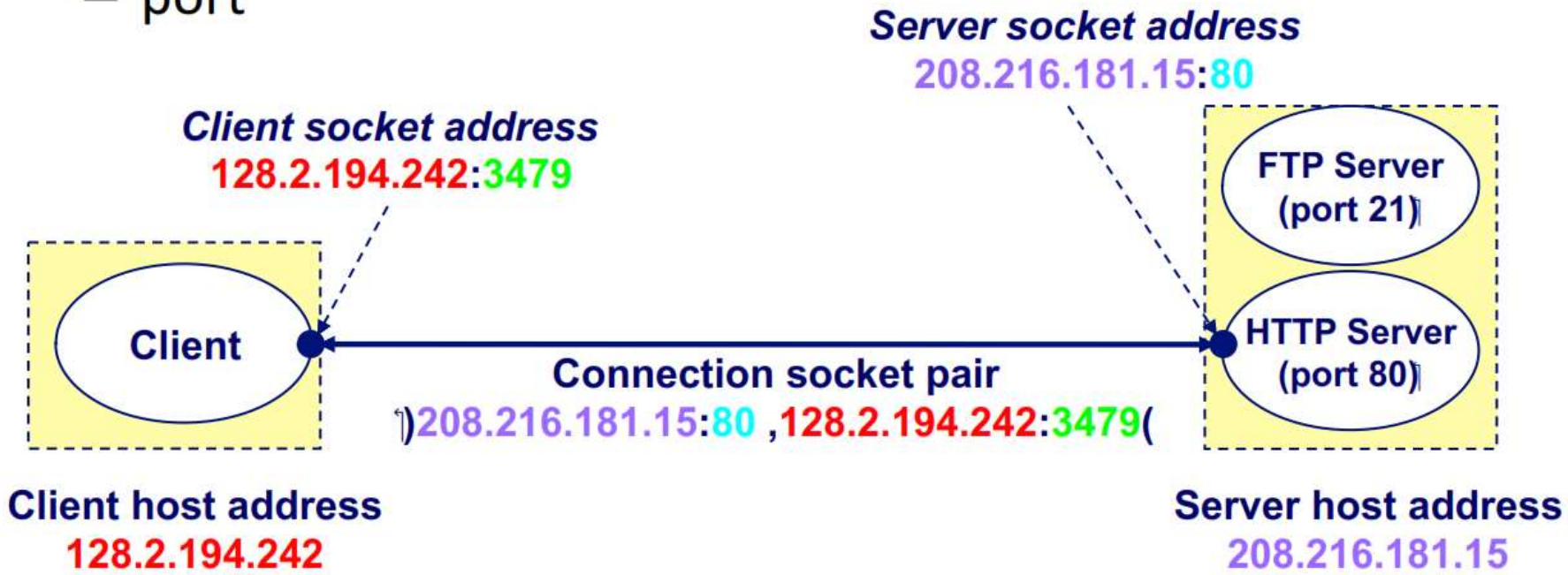
Información asociada a una comunicación

- Protocolo
  - TCP, UDP
- Dirección IP local (origen)
- Puerto local (origen)
- Dirección IP remota (destino)
- Puerto remoto (destino)

(Protocolo, IP-local, P-local, IP-remoto, P-remoto)

Identify the Destination

- Addressing
  - IP address
  - hostname (resolve to IP address via DNS)
- Multiplexing
  - port



How to use Sockets

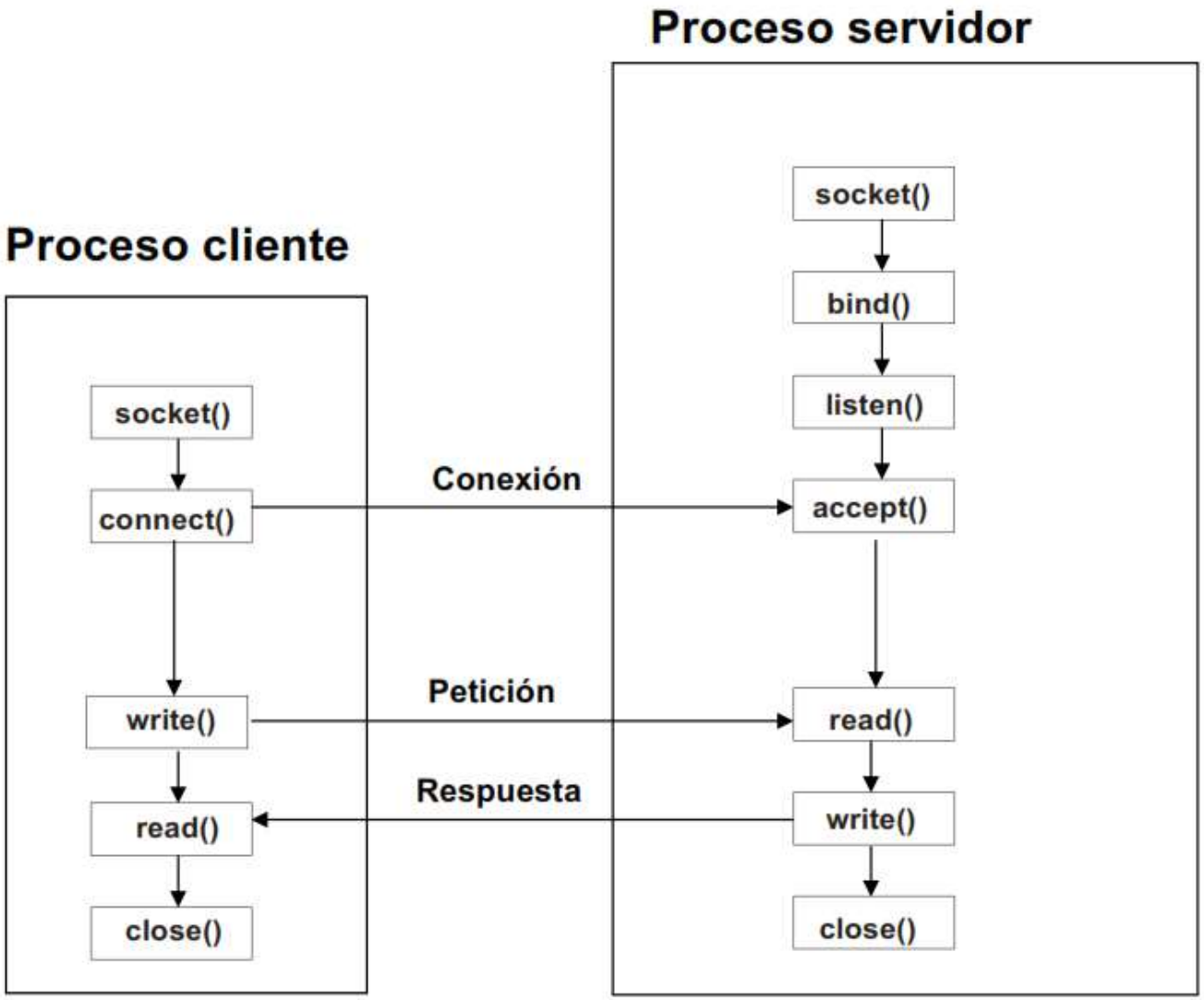
- How to use sockets
  - Setup socket
    - Where is the remote machine (IP address, hostname)
    - What service gets the data (port)
  - Send and Receive
    - Designed just like any other I/O in unix
    - send -- write
    - recv -- read
  - Close the socket

Modelos de comunicación

- Sockets **stream** (SOCK_STREAM)
  - Orientado a conexión
  - **TCP**

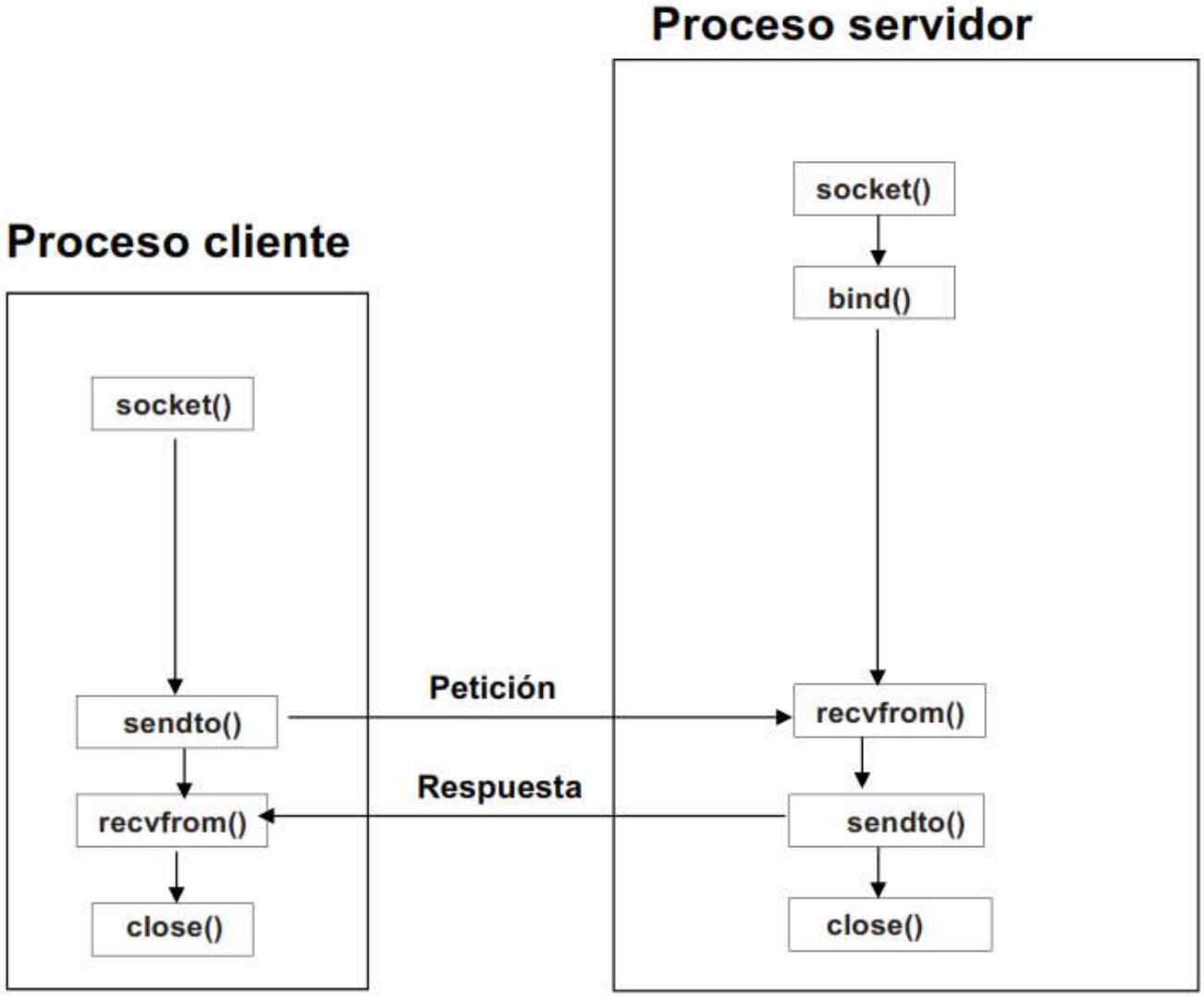
- Sockets **datagrama** (SOCK_DGRAM)
  - No orientado a conexión
  - **UDP**

Modelos de comunicación con sockets *stream*



Modelos de comunicación con sockets

datagrama



Servicios POSIX para utilizar sockets

- Creación de un socket (**socket**)
- Asignación de direcciones (**bind**)
- Preparar para aceptar conexiones (**listen**)
- Aceptar una conexión (**accept**)
- Solicitud de conexión (**connect**)
- Obtener la dirección de un socket
- Transferencia de datos
  - *Streams*
  - Datagramas
- Cerrar un socket (**close**)

Berkeley Socket Services

	Primitive	Meaning
ClientServer	SOCKET	Create a new communication end point
Server	BIND	Attach a local address to a socket
Server	LISTEN	Announce willingness to accept connections; give queue size
Server	ACCEPT	Block the caller until a connection attempt arrives
Client	CONNECT	Actively attempt to establish a connection
Client/Server	SEND	Send some data over the connection
Client/Server	RECEIVE	Receive some data from the connection
Client/Server	CLOSE	Release the connection

- The **SOCKET** primitive creates a new endpoint and allocates table space for it within the transport entity
- The first four primitives are executed in that order by servers
- A successful SOCKET call returns an ordinary **file descriptor** for use in succeeding calls, the same way an OPEN call on a file does

SERVER SOCKET

- Newly created socket has no network address (yet)
  - The machine may have several addresses (thru several interface cards)
  - It must be assigned using the **BIND** primitive method
- Once a socket has **bound** an address, remote clients can connect to it
- The parameters of the **SOCKET** call specify the addressing format to be used, the type of service desired (reliable byte stream , DGRA, etc), and the protocol.

```
import socket
# Creating a server socket on the local machine
sock = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
sock.bind( ('', 2525) )
sock.listen( 5 )
new_sock, (client_host, client_port) = sock.accept()
print "Client:", client_host, client_port
```

CLIENT SOCKET

- A client socket is created exactly as a server socket except that it does **not locally bound** to the machine, and it **does not listen**
- A client socket is **connecting** to an already running server socket, usually on a remote host, but also on the local host (as yet one more method of inter-process communication!)

```
import socket
# Creating a client socket
sock = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
host = socket.gethostname()
# connect to local host at port 2525
server = (host, 2525)
sock.connect(server)
```

CONNECT & ACCEPT primitives

- When a **CONNECT** request arrives from a client to the server, the transport entity creates a **new copy of the server socket** and returns it to the **ACCEPT** method (as a file descriptor)
- The server can then fork off a process or thread to handle the connection on the new socket and go back to waiting for the next connection on the original socket
- ACCEPT returns a file descriptor, which can be used for reading and writing in the standard way, the same as for files.

```
import socket
# Creating a server socket on the local machine
sock = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
sock.bind( ('', 2525) )    # bind to all local interfaces
sock.listen( 5 )           # allow max 5 simultaneous connections
newsock, (client_host, client_port) = sock.accept()
print "Client:", client_host, client_port
```

SEND & RECEIVE primitives

- The **CONNECT** primitive blocks the caller and actively starts the connection process (the transport entity is in charge)
- When it completes (when the appropriate **TCP** segment is received from the server), the client process is awakened by the **OS** and the connection is established
- Both sides can now use **SEND** and **RECEIVE** to transmit and receive data over the full-duplex connection

```
# server to client:
newsock.send("Hello from Server 2525")

# client to server
server = (host, 2525)
sock.connect(server)    # connect to server
sock.recv(100)          # receive max 100 chars
```

CLOSE primitive

- When both sides have executed the CLOSE method, the connection is released
- Berkeley sockets have proved tremendously popular and have became the standard for abstracting transport services to applications
- The socket API is often used with the TCP protocol to provide a connection-oriented service called a **reliable byte stream**
- But sockets can also be used with a connectionless service (UDP)
- In such case, **CONNECT** sets the address of the remote transport peer and **SEND** and **RECEIVE** send and receive UDP datagrams to and from the remote peer

```
# Server:
newsock.close()

# Client
sock.close()
```

The Simplest Client/Server App

SERVER

```
import socket
# Creating a server socket on the local machine
sock = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
sock.bind( ('', 2525) )
sock.listen( 5 )
newsock, (client_host, client_port) = sock.accept()
print "Client:", client_host, client_port
newsock.send("Hi from server 2525")
newsock.close()
```

CLIENT

```
import socket
# creating a client socket
sock = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
host = socket.gethostname()
# connect to local host at port 2525
server = (host, 2525)
sock.connect(server)
print sock.recv(100)
sock.close()
```

Q: How many clients can connect to this server?

Ejercicio

Modificar el servidor para que envíe la fecha y hora actual