

Infraestructura Computacional

Concurrencia en Servidores

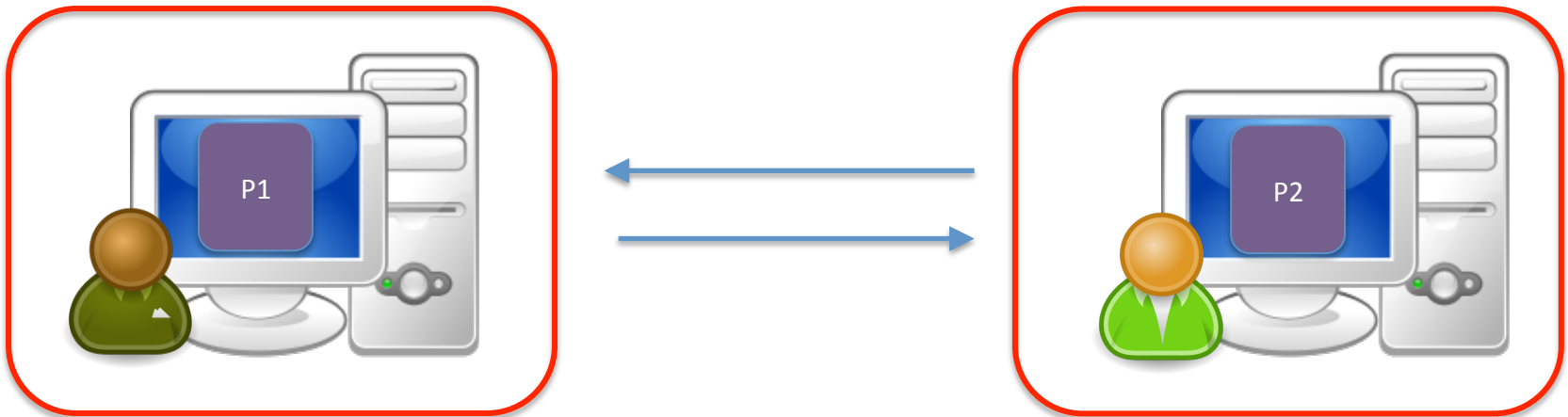
Rafael Gómez

Francisco Rueda

Sandra Rueda

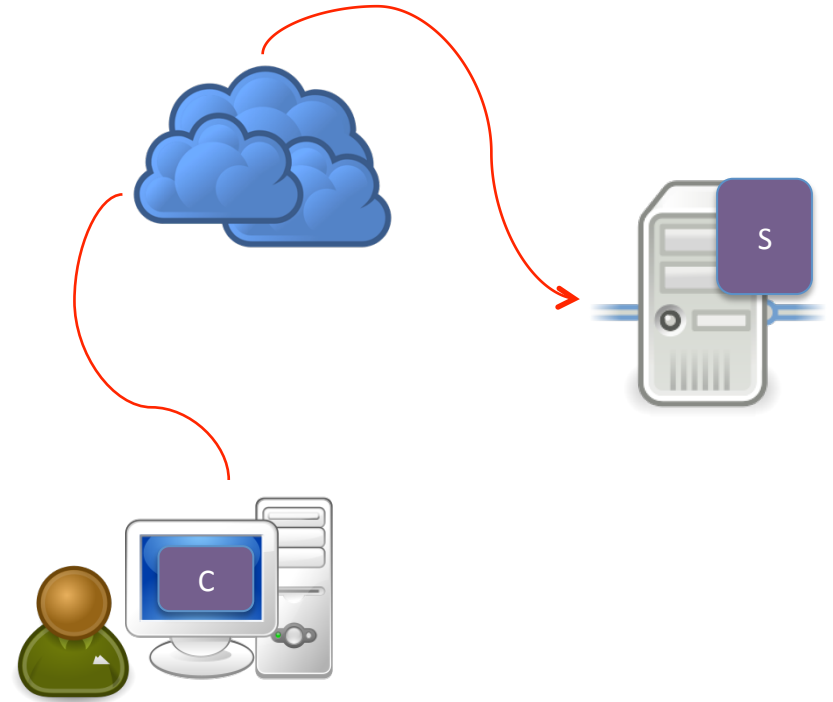
Comunicación

- Procesos en máquinas diferentes



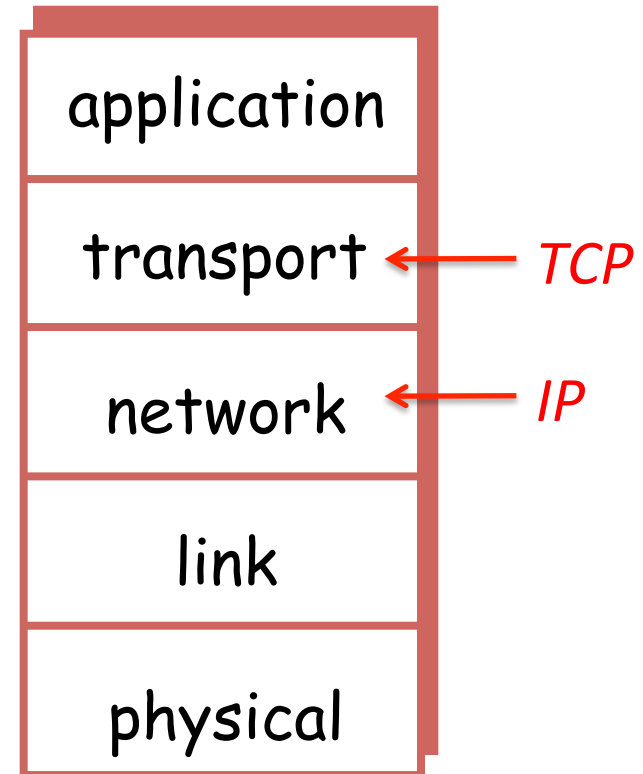
Arquitectura Cliente Servidor

- Servidor
 - Siempre en el mismo equipo
 - Dirección IP fija
- Cliente
 - Inicia la comunicación
 - Dirección IP puede ser dinámica



Comunicaciones en Internet

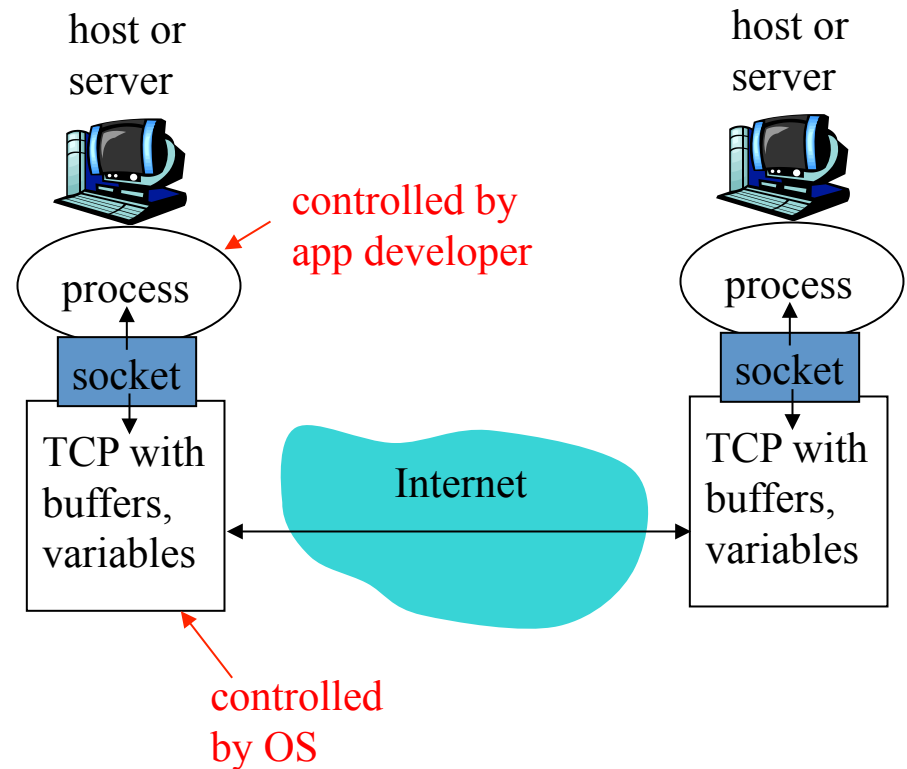
- El protocolo de comunicaciones en Internet está diseñado en capas
 - Cada capa es responsable de un conjunto de tareas
 - TCP es la capa responsable de la transferencia de datos entre procesos



[Kurose y Ross]

Sockets

- Procesos corriendo en máquinas diferentes envían y reciben mensajes por medio de sockets



[Kurose y Ross]

Sockets

- La identificación del socket asociado a un proceso incluye dos elementos:
 - Dirección IP
 - Número de puerto



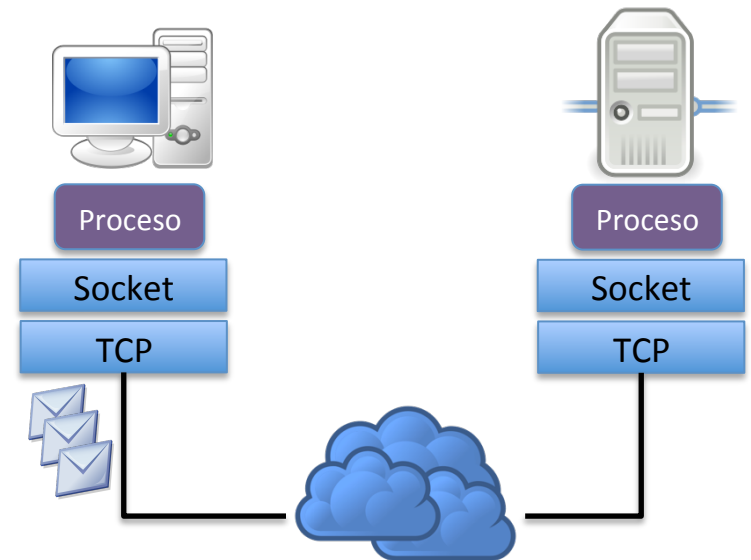
Dirección IP
157.253.2xx.y



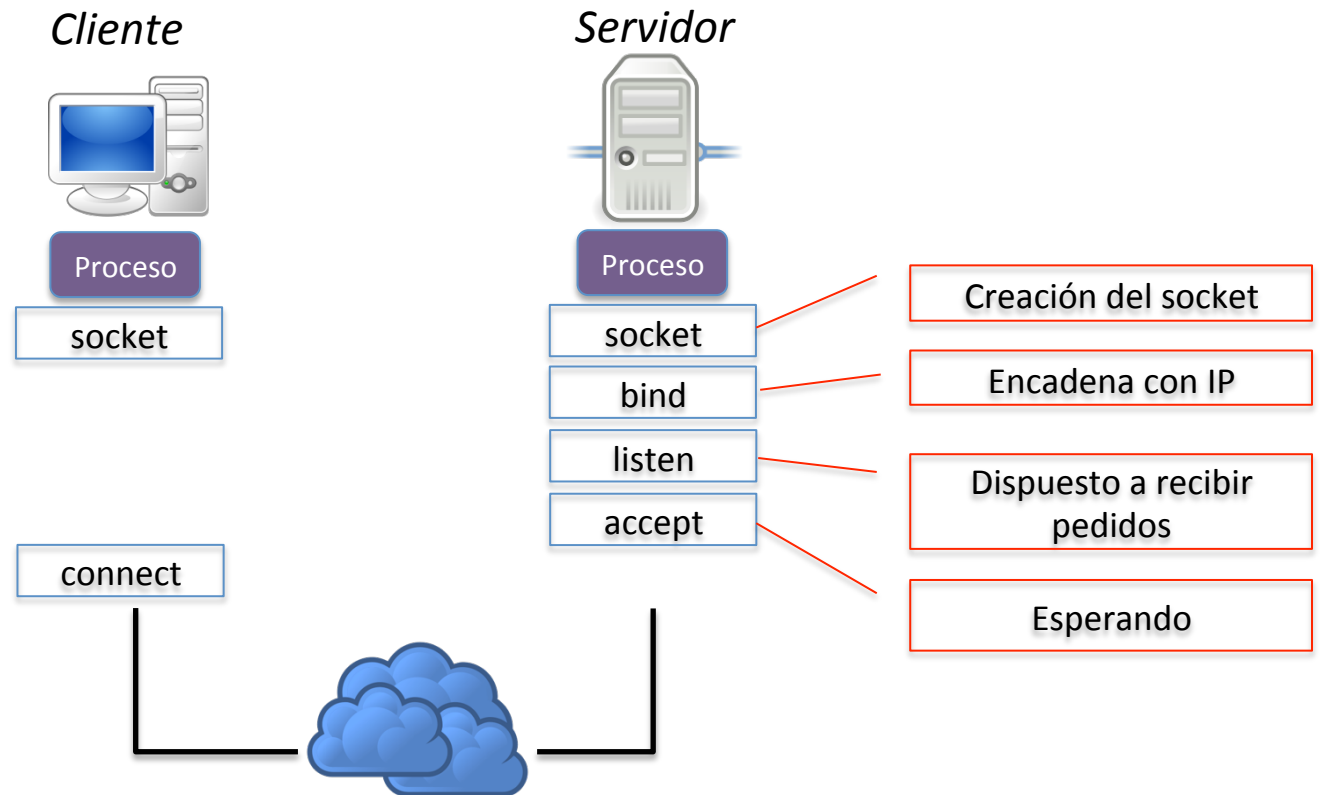
Puerto
80

Sockets

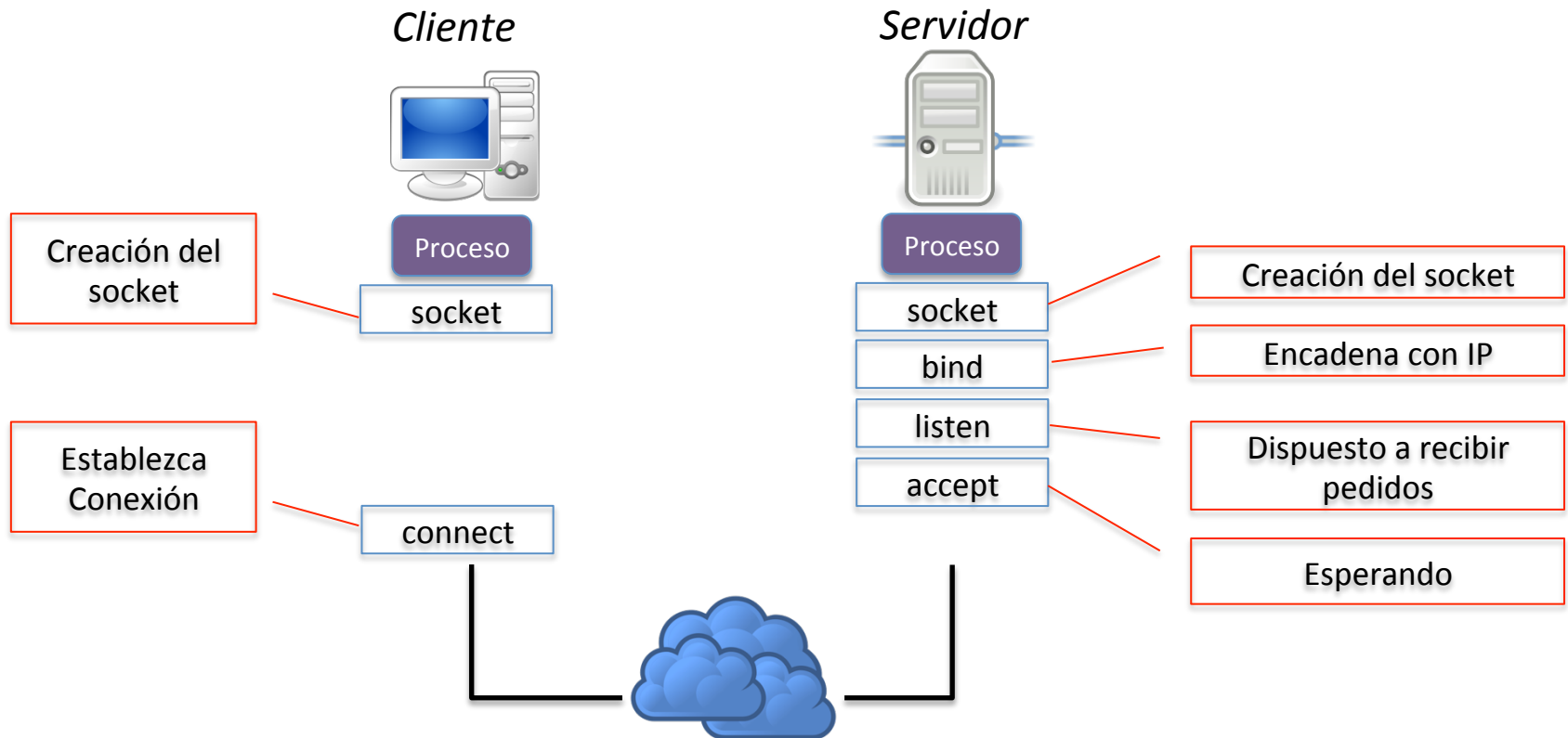
- Transmisión confiable (TCP)
 - Establece una conexión
 - Orden
 - Control de flujo
 - Control de congestión
- Mejor esfuerzo (UDP)



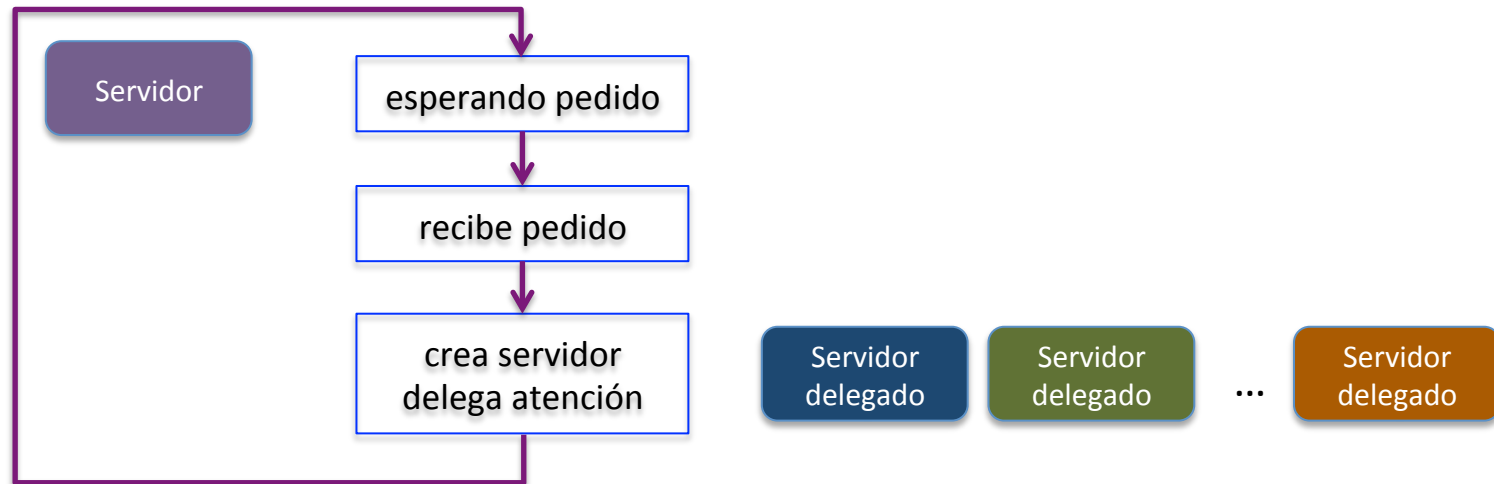
Servidor



Cliente



Servidor Concurrente



Servidor Concurrente

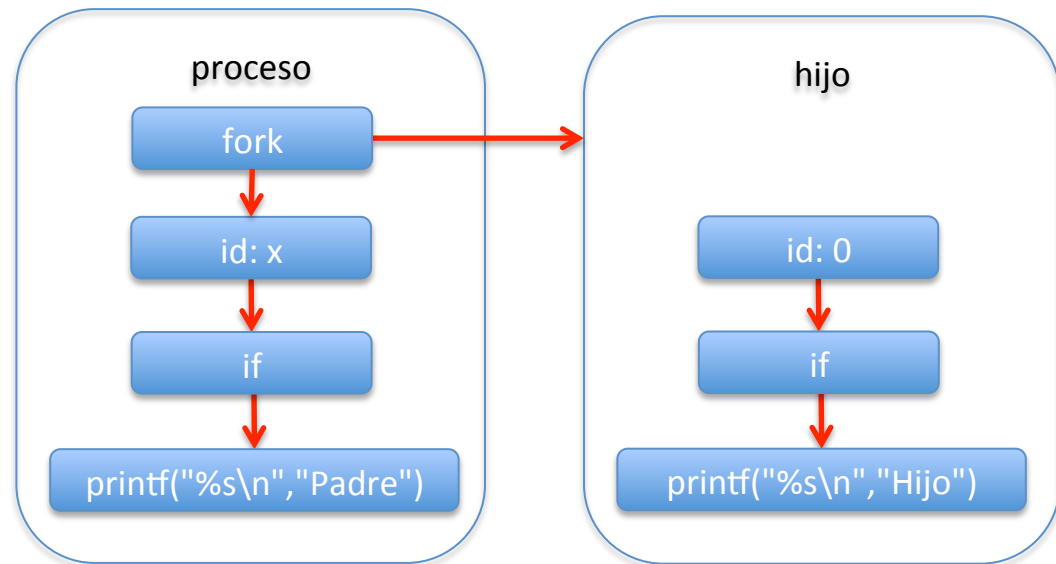
- fork
 - Instrucción en Unix que crea un proceso hijo
 - Crea un nuevo proceso duplicando el proceso padre (el proceso que hace el llamado)
 - Los dos continúan ejecución a partir de la misma instrucción (la instrucción siguiente al fork)

Servidor Concurrente

- fork
 - Proceso:
 - Segmento de datos
 - Segmento de pila
 - Segmento de instrucciones (compartido)
 - El valor de retorno difiere:
 - Si el proceso falla el padre recibe -1
 - Si el proceso es exitoso, el padre recibe el identificador del hijo y el hijo recibe 0

Servidor Concurrente

```
id = fork();  
    if (id == 0){  
        printf("%s\n", "Hijo");  
    } else {  
        printf("%s\n", "Padre");  
    }  
}
```



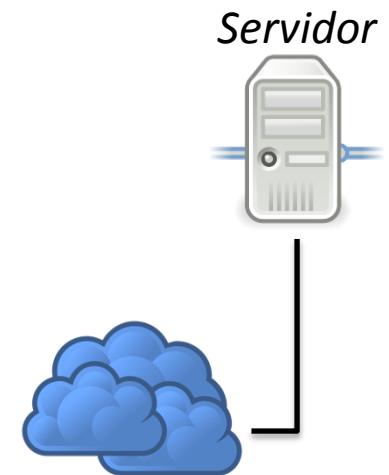
Servidor Concurrente

```
s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);  
bind(s, &serv_addr, sizeof(serv_addr));  
listen(s, n);  
while (true){  
    s2 = accept(s, &cli_addr, &cli_len);  
    id = fork();  
    if (id == 0){  
        // este es el hijo  
        close(s);  
        atender(s2);  
        exit(0);  
    } else {  
        close(s2);  
    }  
}
```

Creación del socket

Encadena con IP y
puerto

Encola hasta n pedidos



Servidor Concurrente

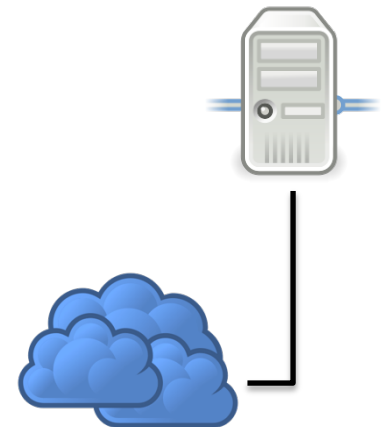
```
s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
bind(s, &serv_addr, sizeof(serv_addr));
listen(s, n);
while (true){
    s2 = accept(s, &cli_addr, &cli_len);
    id = fork();
    if (id == 0){
        // este es el hijo
        close(s);
        atender(s2);
        exit(0);
    } else {
        close(s2);
    }
}
```

Ejecuta
indefinidamente

Espera un cliente y
obtiene sus datos

Crea otro proceso para
atender al cliente

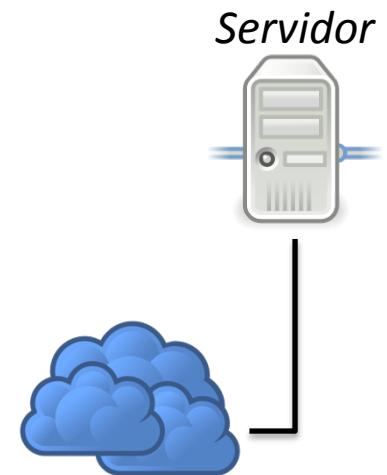
Servidor



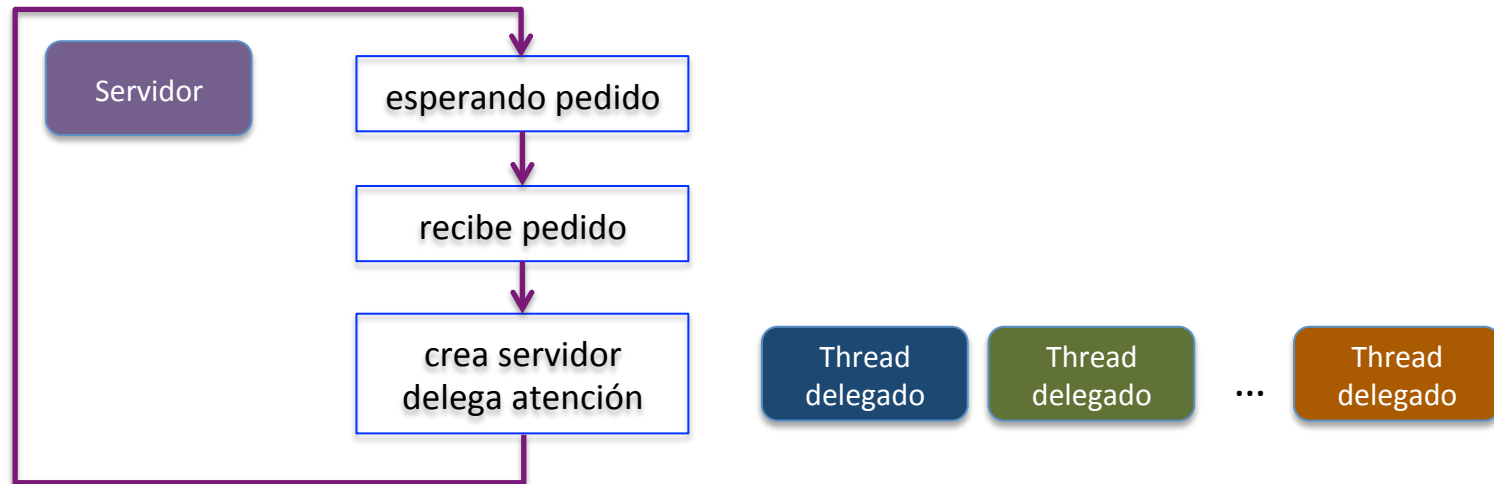
Servidor Concurrente

```
s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);  
bind(s, &serv_addr, sizeof(serv_addr));  
listen(s, n);  
while (true){  
    s2 = accept(s, &cli_addr, &cli_len);  
    id = fork();  
    if (id == 0){  
        // este es el hijo  
        close(s);  
        atender(s2);  
        exit(0);  
    } else {  
        close(s2);  
    }  
}
```

Atiende al cliente



Servidor Concurrente

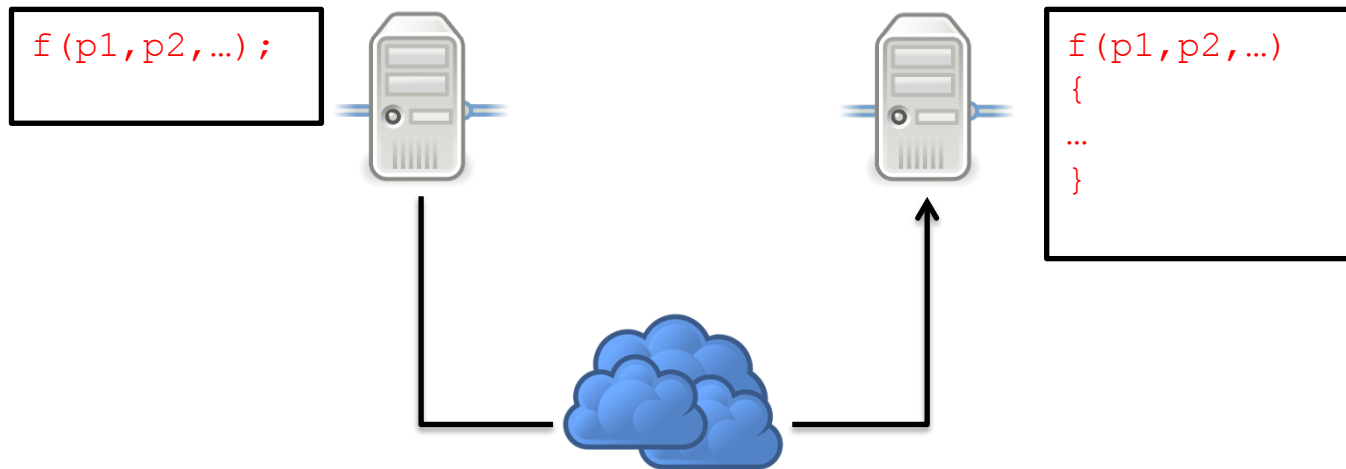


Mecanismos de Comunicación

- Sockets
- RPC/RMI
- Servicios Web

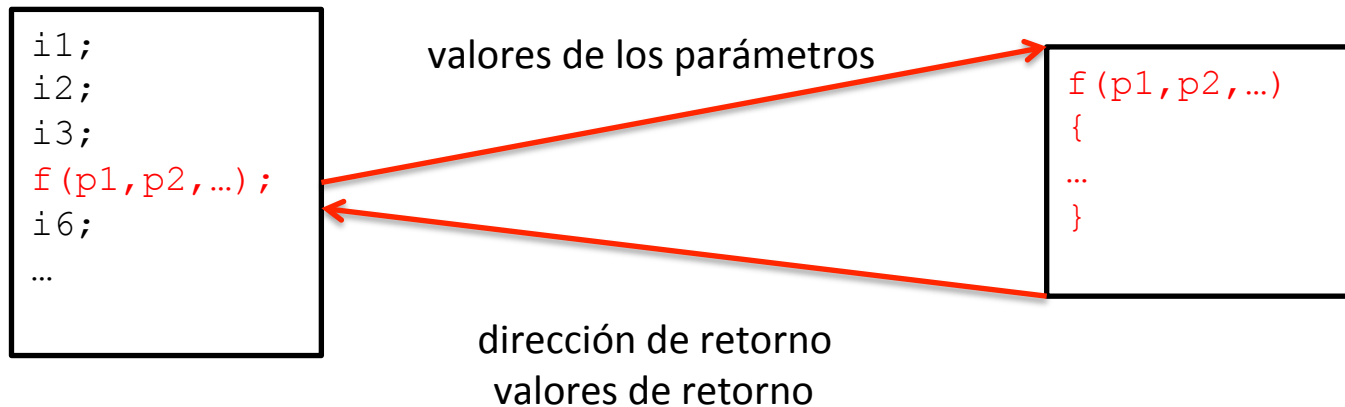
RPC

- Remote Procedure Call
 - O cómo permitir a un programa llamar a un procedimiento localizado en un computador remoto.



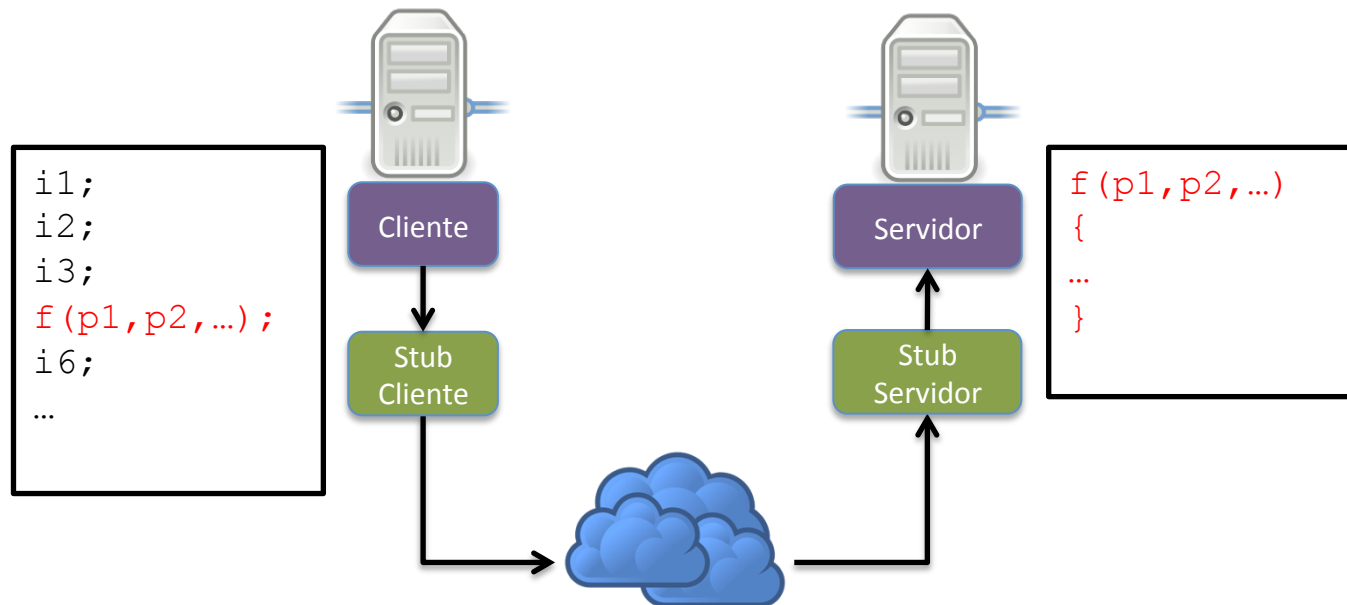
RPC

- El mecanismo es análogo al mecanismo que se usa cuando un procedimiento llama a otro procedimiento localizado en el mismo computador



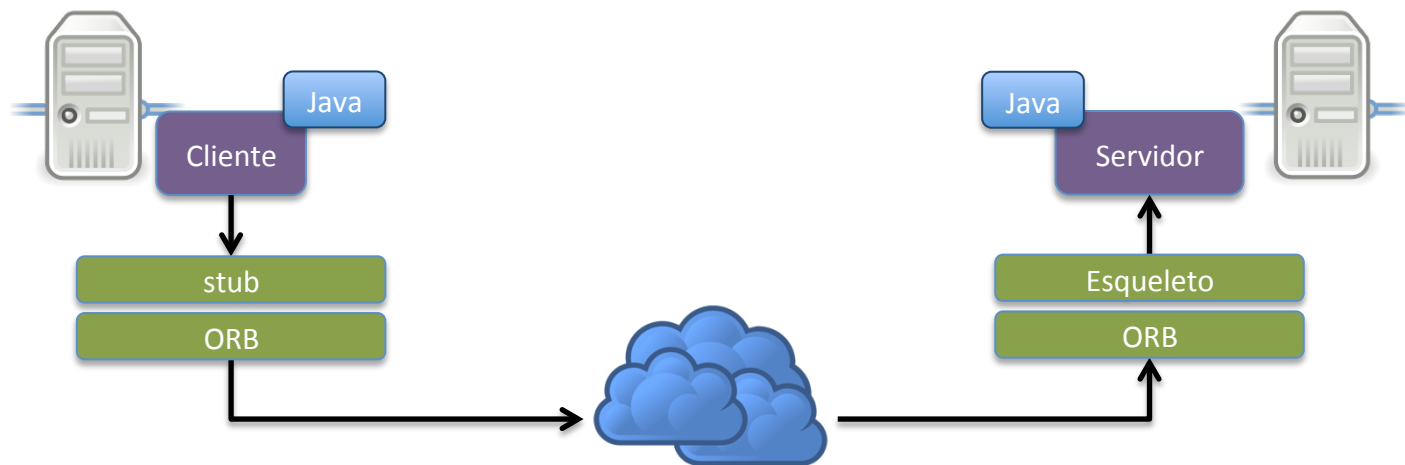
RPC

- Tanto el cliente como el servidor deben encadenarse con una librería que se encarga de la transmisión de los datos y la ejecución del procedimiento solicitado.



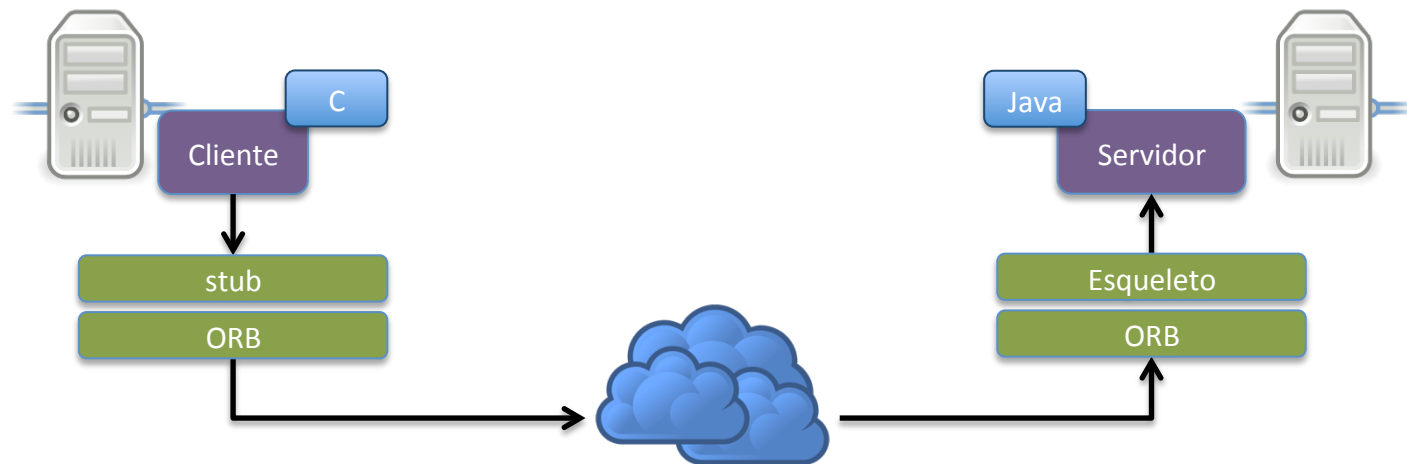
RMI

- Remote Method Invocation
 - API de Java equivalente a RPC
 - RMI implementa su propio ORB, Object Request Broker, que permite el llamado a métodos de objetos remotos



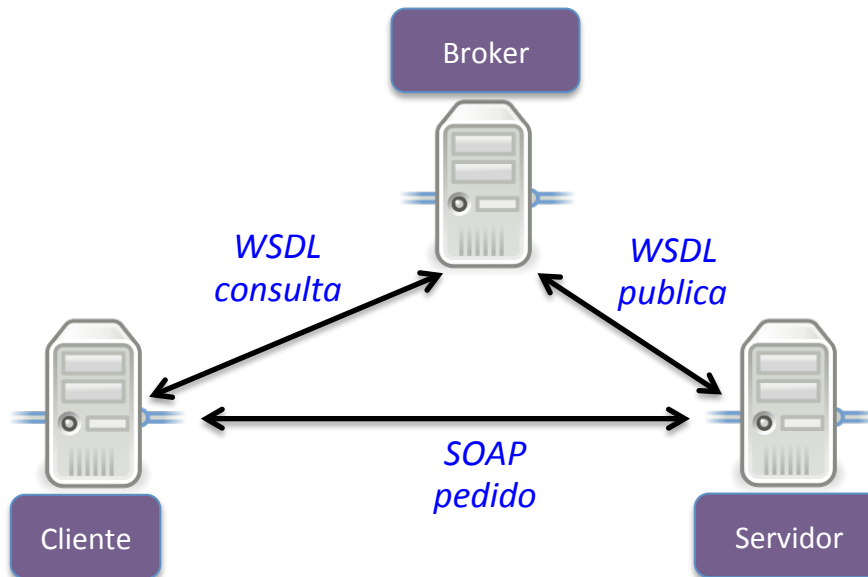
CORBA

- CORBA
 - Common Object Request Broker Architecture
 - Permite el llamado a métodos de objetos remotos y escritos posiblemente en otros lenguajes



Web Services - WS

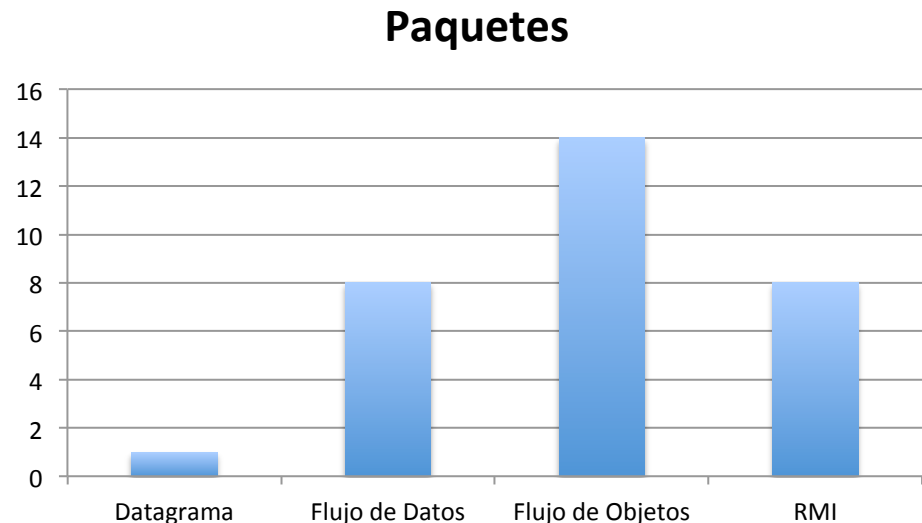
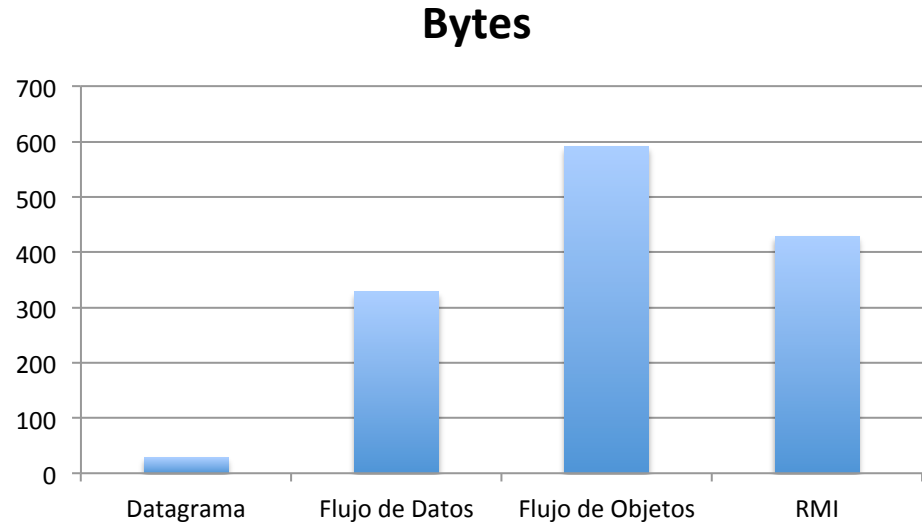
- Análogo a RPC
 - Mecanismo de llamado a procedimientos remotos sobre WWW
 - WSDL - Lenguaje para definir servicios
 - SOAP – Mensajes para solicitar servicios



Mecanismos de Comunicación Remota

- Protocolos Java enviando un mensaje de 5 bytes
 - Datagrama. UDP.
 - Flujo de datos. TCP.
 - Flujo de objetos. TCP, Permite a un programados enviar un mensaje a nivel de aplicación como un solo objeto.
 - RMI. TCP, API de alto nivel.

[Pendergast]



Mecanismos de Comunicación Remota

	uso	eficiencia	abstracción
sockets	estándar de facto en comunicaciones en Internet	eficientes	abstracción de bajo nivel y como consecuencia es más dispendioso construir aplicaciones
WS		sobrecarga en el manejo de mensajes	abstracción de alto nivel y como consecuencias es más fácil construir aplicaciones

Manejo de la Concurrency

Tipo de Servidor	Implementación
Iterativo	un solo thread
Concurrente	un thread por solicitud
Concurrente	un thread por conexión
Concurrente	pool de threads
Combinaciones	

Servidor Iterativo

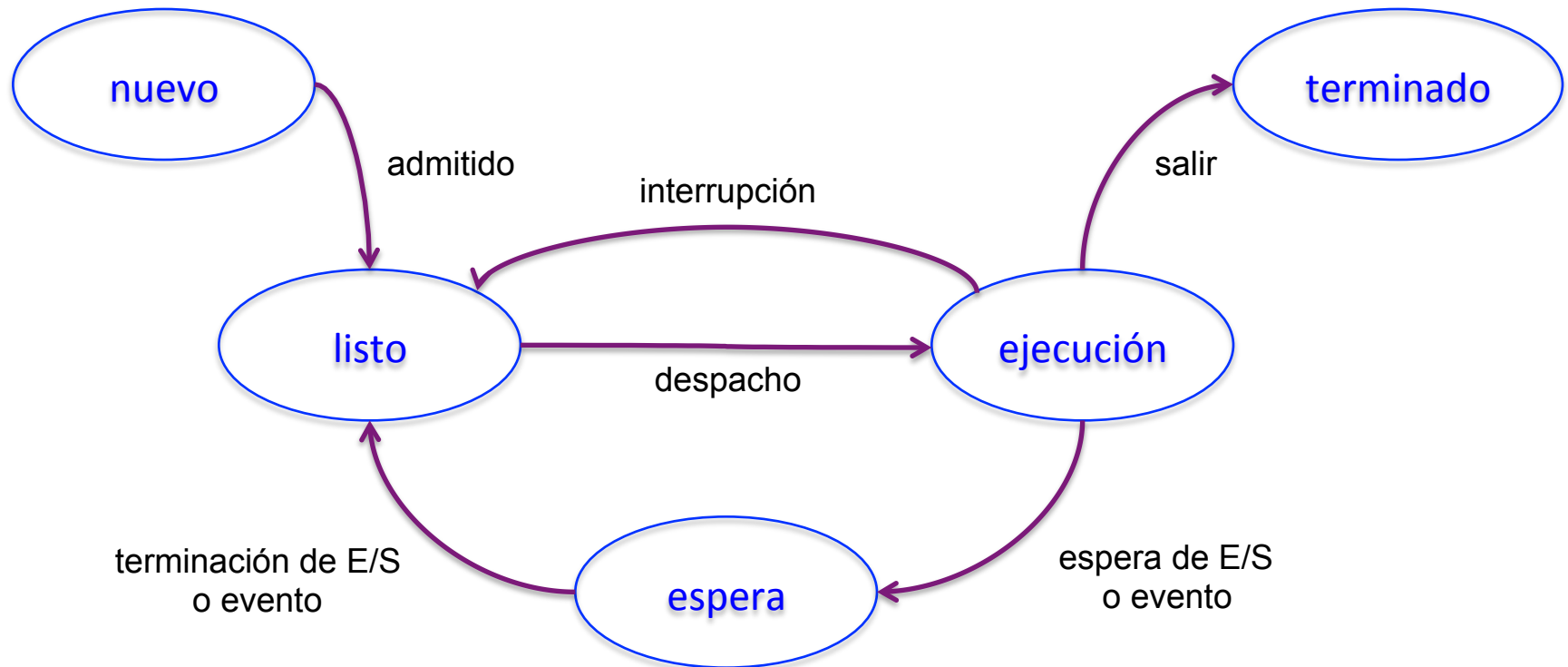
```
while (true) {  
    s2 = accept (s1,.... )  
    while (haya_conexión) {  
        obtener (s2, solicitud)  
        servir (s2, solicitud)  
    }  
    close (s2) ;  
}
```

Termina de atender un
cliente para continuar

Servidor Iterativo

- En el caso de los servidores iterativos no hay concurrencia.
- Además :
 - Código simple
 - No hay cambios de contexto
 - Si el proceso se duerme el procesador queda desocupado
 - No es posible usar varios procesadores

Estados de un Proceso



Servidor Concurrente

```
while (true) {  
    s2 = s1. accept (.....)  
    serv c = new serv(s2 ) ← Servidor por  
    c.start ()           conexión  
}                          Crea un delegado y continua  
.....
```

Atendiendo Pedidos

```
public class TreadServ extends Thread
{
    public void run ( ) {
        while (haya_conexión) {
            obtener(socket, solicitud);
            servir(socket, solicitud);
        }
    }
}
```


Servidor Concurrente

- Por Conexión
 - Se pierde tiempo creando unidades concurrentes cada vez que hay una conexión
 - si el procesamiento es largo, el tiempo de creación se “amortiza”

Manejo de Pools

- Para evitar el tiempo de creación en el momento del servicio es posible crear un pool de threads para atender los pedidos

Servidor Concurrente

```
ServerSocket s1 = new ServerSocket();  
// Crear los threads t1,t2,...,tn  
i = 0  
while (true) {  
    ServerSocket s2 = s1.accept (...);  
    poner ( s2, i )  
    i++ mod n  
}
```

← pool de threads

← poner en la cola del thread i

Servidor Concurrente

```
thread i {  
    while (true) {  
        sacar (sock, i)  
        while (haya_conexión) {  
            obtener (sock, solicitud)  
            servir (sock, solicitud )  
        }  
    }  
}
```

sacar de la
cola



Manejo del Pool

- Uno de los aspectos que se deben tener en cuenta es cuántos threads tener en el pool
- Una alternativa es tener un cierto número, y si se queda corto, agregar más
- ¿Cómo calcular el número ideal de threads que el pool va a manejar?

Ventajas del Pool

- Concurrencia
- Al tener un límite fijo en el número de threads se tiene control sobre el uso de los recursos
- No se gasta tiempo en la creación de los threads
- Se puede balancear carga
 - Es posible definir una política que dependa de la longitud de las colas

Desventajas del Pool

- Se gasta tiempo en cambios de contexto
- El manejo de la cola tiene un costo
- Con el esquema básico no se maneja asignación de prioridades
 - El thread principal (que administra) debería tener la prioridad más alta

Servidor Concurrente

- thread por conexión vs. thread por solicitud



Servidor Concurrente

- Se puede tener una variante en la cual hay un grupo de threads y el primero disponible es el que se encarga de atender el próximo pedido

Servidor Concurrente

```
ServerSocket s1 = new ServerSocket()  
crear los threads t1, t2, .. tn  
dormir los threads t1, t2, ... tn  
turno = 1;  
despertar tturno;  
while (true) {  
}
```

← pool de
threads

← turno indica el
siguiente thread que
atenderá un pedido

*cada thread atiende
por conexión*

thread i:

```
correr() {  
    while (true) {  
        ServerSocket s2 = s1.accept(...);  
        despertar_proximo_thread();  
        while (haya_conexión) {  
            obtener (s2, solicitud)  
            servir (s2, solicitud)  
        }  
        if (turno!=0)  
            dormir();  
        else  
            turno = i;  
    }  
}
```

variable local

variable global

```
despertar_proximo_thread()
```

```
{
```

```
    turno = proximo();
```

```
    if (turno==0)
```


```
        turno = 0;
```

```
    else
```

```
        despertar tturno;
```

```
    }
```

```
}
```



siguiente thread
disponible

Servidor Concurrente

- El método del servidor concurrente con pool de threads que buscan directamente en el socket tiene la ventaja de evitar el manejo de la cola. Aunque incurre en otros costos.
- Existen otros métodos posibles, por ejemplo hay infraestructuras que permiten que el usuario ponga filtros o sistemas que permiten tener distintas políticas (p.e. Java)

Eficiencia

- Algunos factores influyen en la eficiencia que puede tener el manejo de concurrencia :
 - Número de procesadores
 - % de tiempo dedicado al procesamiento
 - Tiempo para administrar unidades concurrentes (procesos o threads)
 - Carga
 - Duración de las tareas

Uso de Servidores Concurrentes

- ¿Cuándo?

	Alto	Bajo
Número de procesadores	X	
% uso del procesador		X
Tiempo de administración		X
Carga	X	
Duración de las tareas	X	

Uso de Servidores Concurrentes

- ¿Cuándo?

	Alto	Bajo
% de uso de procesador por thread vs. eventos, por ejemplo	X	
tiempo invertido en tareas de administración de los threads		X
el número de threads es alto (se requiere manejo de concurrencia para darle un tiempo en el procesador a todos)		X
	X	
tareas de larga duración (se amortiza el tiempo de administración)	X	

Concurrencia

- Convendría usar concurrencia cuando
 - Hay mas de un procesador
 - El servicio del pedido no es intensivo en uso de procesador
 - Se usan threads
 - La carga es alta
 - Las tareas son largas

¿Qué Hacer?

Servicio	Número de pedidos	Duración de los pedidos	Uso del procesador
Servidor web			
Servidor de correo			
Servidor web de carga baja			
Servidor grid			



Tipos de Servicios

Servicio	Número de pedidos	Duración de los pedidos	Uso del procesador
Servidor web	A	Depende del servicio	Depende del servicio
Servidor de correo	A	B (suponiendo tareas básicas)	B (suponiendo tareas básicas)
Servidor web de carga baja	B	Depende del servicio	Depende del servicio
Servidor grid	Varía	A	A

Tipos de Concurrency

Servicio	Tipo de Concurrency	Razón
Servidor web	Pool de threads por conexión con un tope máximo	Debe atender múltiples clientes, pero sin exceder el uso de los recursos disponibles.
Servidor de correo	Pool de threads por conexión con un tope máximo	Debe atender múltiples clientes pero sin exceder el uso de recursos disponibles.
Servidor web de carga baja	Thread por conexión	Debe atender múltiples clientes. Como la carga es baja suponemos que nunca va a exceder el uso de los recursos disponibles.
Servidor grid	Pool de threads (con el límite establecido por defecto)	Puede atender un número bajo de clientes (suponemos que cada cliente tiene un porcentaje de uso de procesador alto, como consecuencia no hay recursos disponibles para atender clientes adicionales).

Arquitectura

- Factores a tener en cuenta
 - Número de procesadores
 - Memoria RAM disponible

Número de Procesadores

- Determinan el número máximo de procesos o threads que se pueden ejecutar de forma simultánea
 - Cada proceso o thread en ejecución corre en un procesador diferente
- ¿Qué pasa cuando se crean varios procesos o threads en una arquitectura con un solo procesador?

Memoria RAM

- Los procesos en ejecución deben estar en memoria principal
 - Un proceso puede removerse de memoria principal y enviarse temporalmente a memoria secundaria
 - Por ejemplo, mientras espera la terminación de un evento
 - Más tarde se vuelve a copiar a memoria para continuar con su ejecución

Memoria RAM

- La información de los procesos en ejecución debe estar en memoria
 - En algunos casos NO es posible cargar toda la información del proceso en memoria
 - Una falla de página o defecto de página es una referencia a una parte de un proceso que no está en memoria principal

Memoria Lógica vs. Física

página 0
página 1
página 2
página 3

*Memoria Lógica
(para proceso i)*

0	1
1	4
2	3
3	

*Tabla de Páginas
(para proceso i)*

0	
1	página 0
2	
3	página 2
4	página 1
5	
6	
7	página 3

Memoria Real

Memoria

- Si hay una falla de página, la información debe ser recuperada (de disco) y almacenada en memoria principal (volveremos a este tema cuando hablemos de memoria virtual).
- Si la memoria es pequeña se generarán muchas fallas de página, reduciendo el desempeño.

Concurrencia

- ¿Cuándo es conveniente usar concurrencia?
- ¿Cómo se relaciona la concurrencia con el hardware subyacente?
- ¿Cómo controlar el uso de recursos?

Referencias

- [Pendergast] Performance, Overhead, and Packetization Characteristics of Java Application Level Protocols. Mark Pendergast. 2011
- Threads en Java: Java Programming Language, Sun Educational Services, capítulo 13. 2001
- Organización de Threads: Douglas C. Schmidt, Evaluating architectures for multithreaded Object Request Brokers, Communications ACM, Vol. 41, No 10. Octubre de 1998
- [Kurose y Ross] Computer Networking. A Top Down Approach. J. Kurose y K. Ross.