Tema 3
Paso de mensajes

Sistemas Distribuidos
Grado en Ingeniería Informática
Universidad Carlos III de Madrid

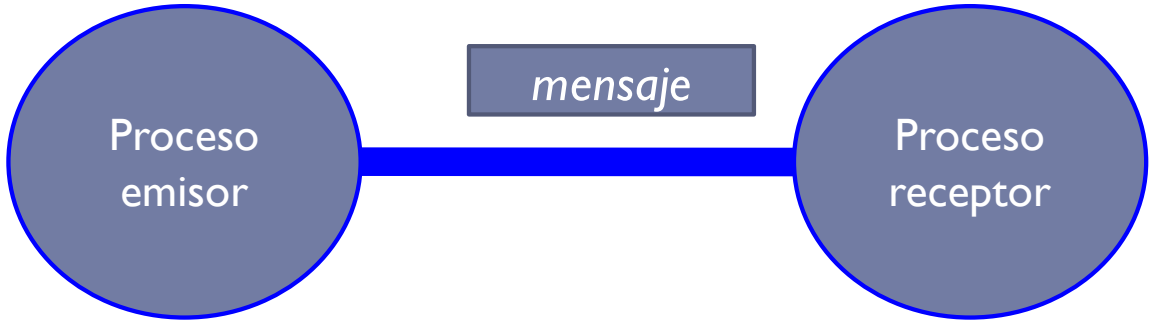
Contenido

- Conceptos básicos
- Formas de representación de los datos
- MPI (Message Passing Interface)
- Colas de mensajes POSIX
- Modelo cliente Servidor
- Arquitectura SW de aplicaciones distribuidas
- Servidores concurrentes

Paso de mensajes

- Mecanismo de comunicación y sincronización de procesos que ejecutan en distintas máquinas
- En este tipo de comunicación, los procesos intercambian mensajes
- Entre los procesos que comunican debe existir un enlace de comunicación
- Dos primitivas básicas:
  - send (destino, mensaje)
  - receive(origen, mensaje)

Modelo de comunicación



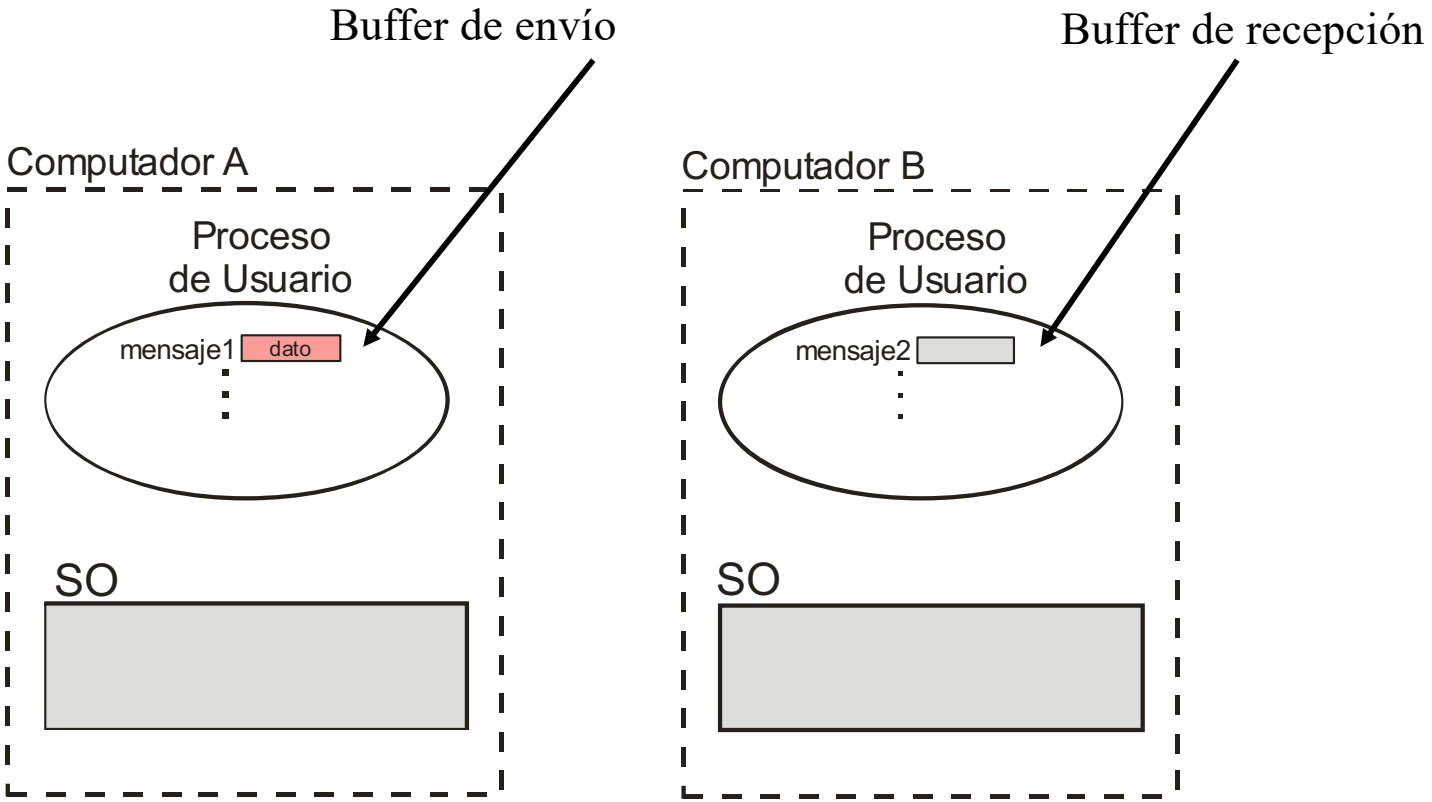
Emisor:

send(destino, mensaje)

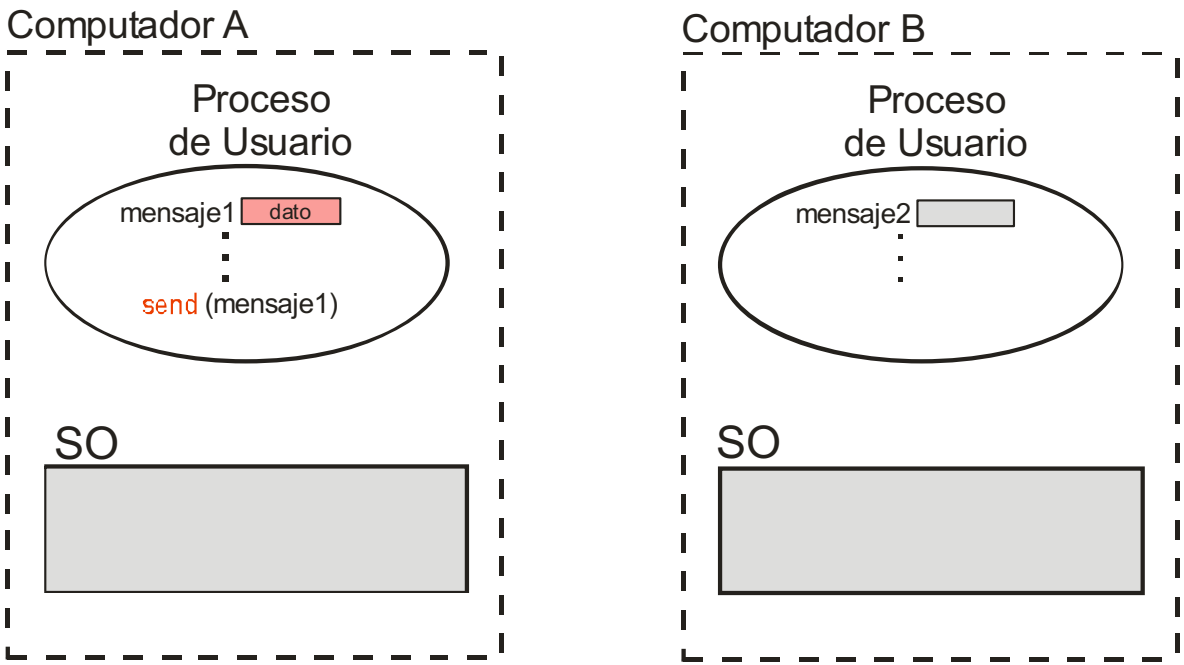
Receptor:

receive(origen, mensaje)

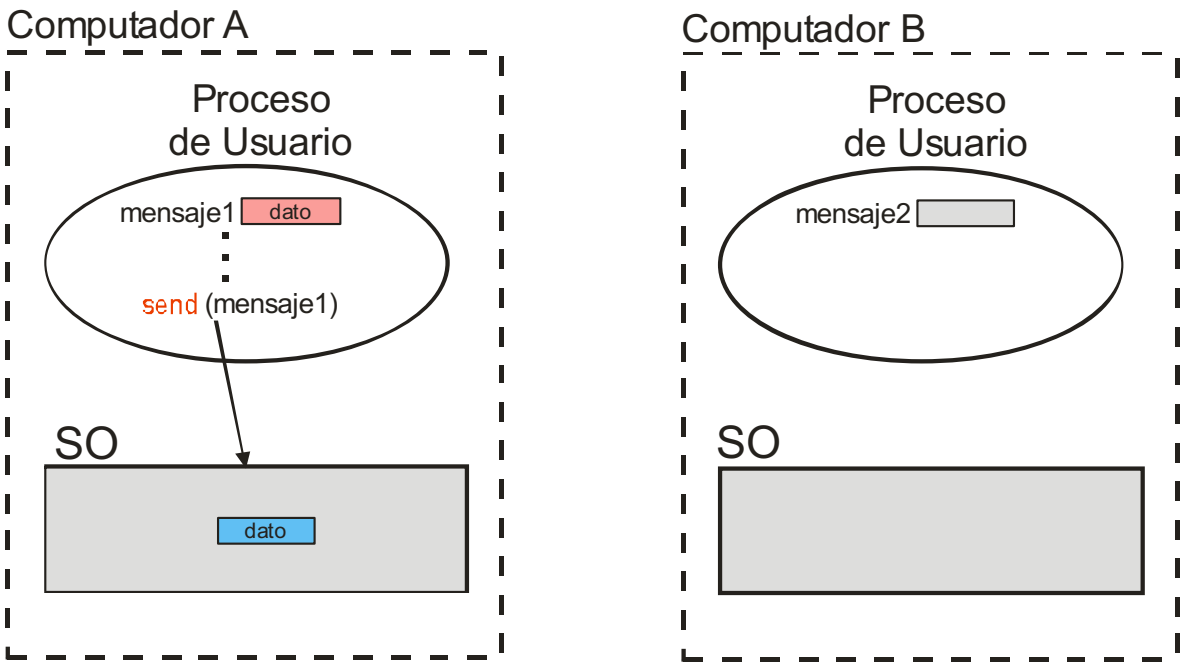
Proceso de comunicación



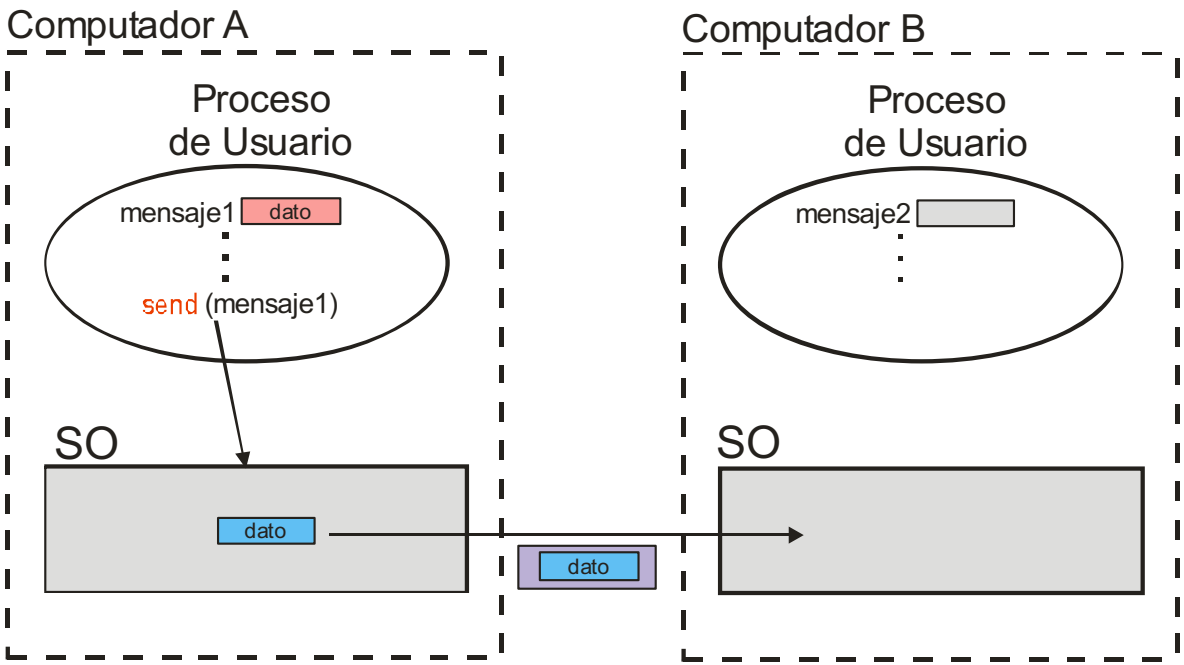
Proceso de comunicación



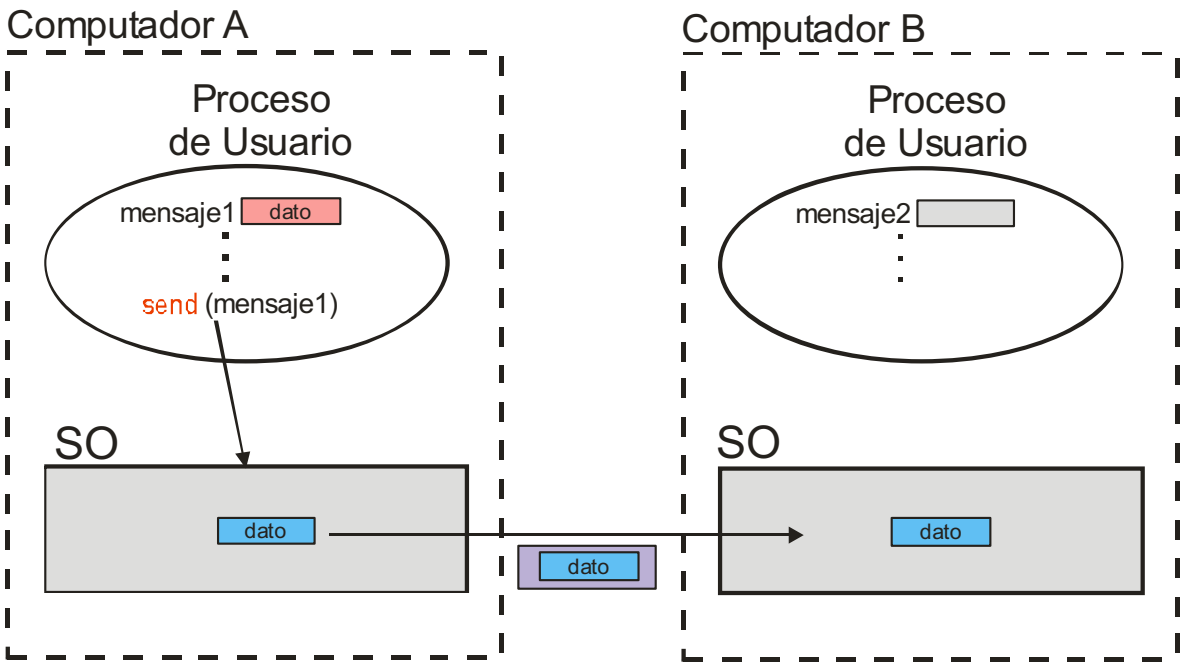
Proceso de comunicación



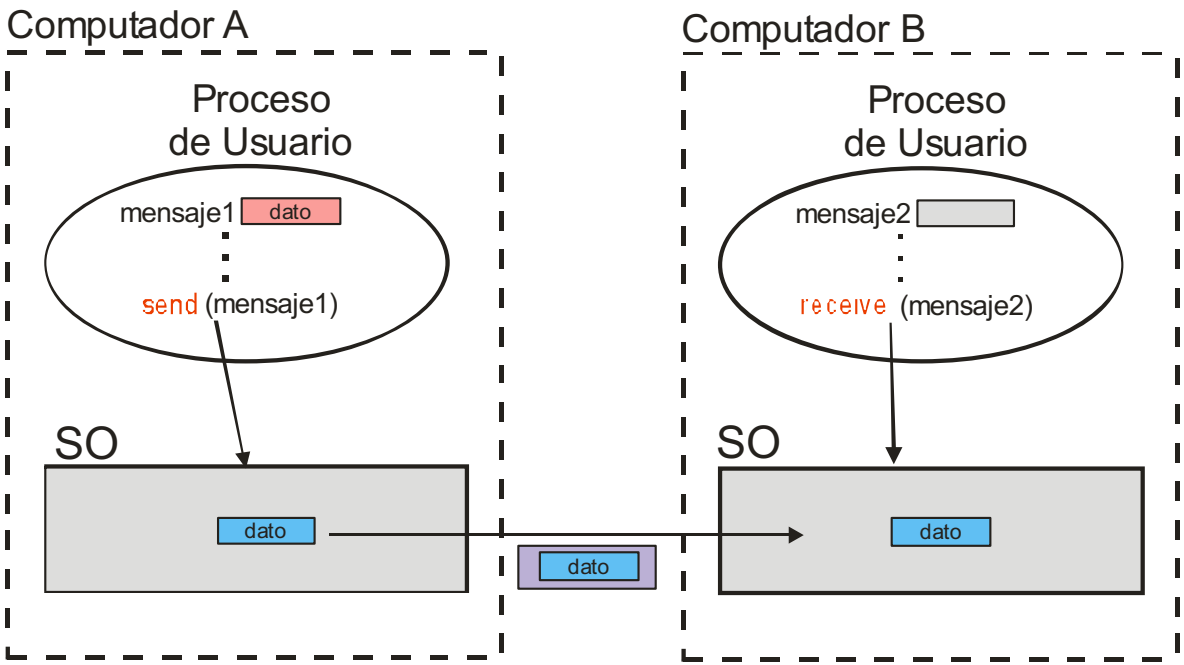
Proceso de comunicación



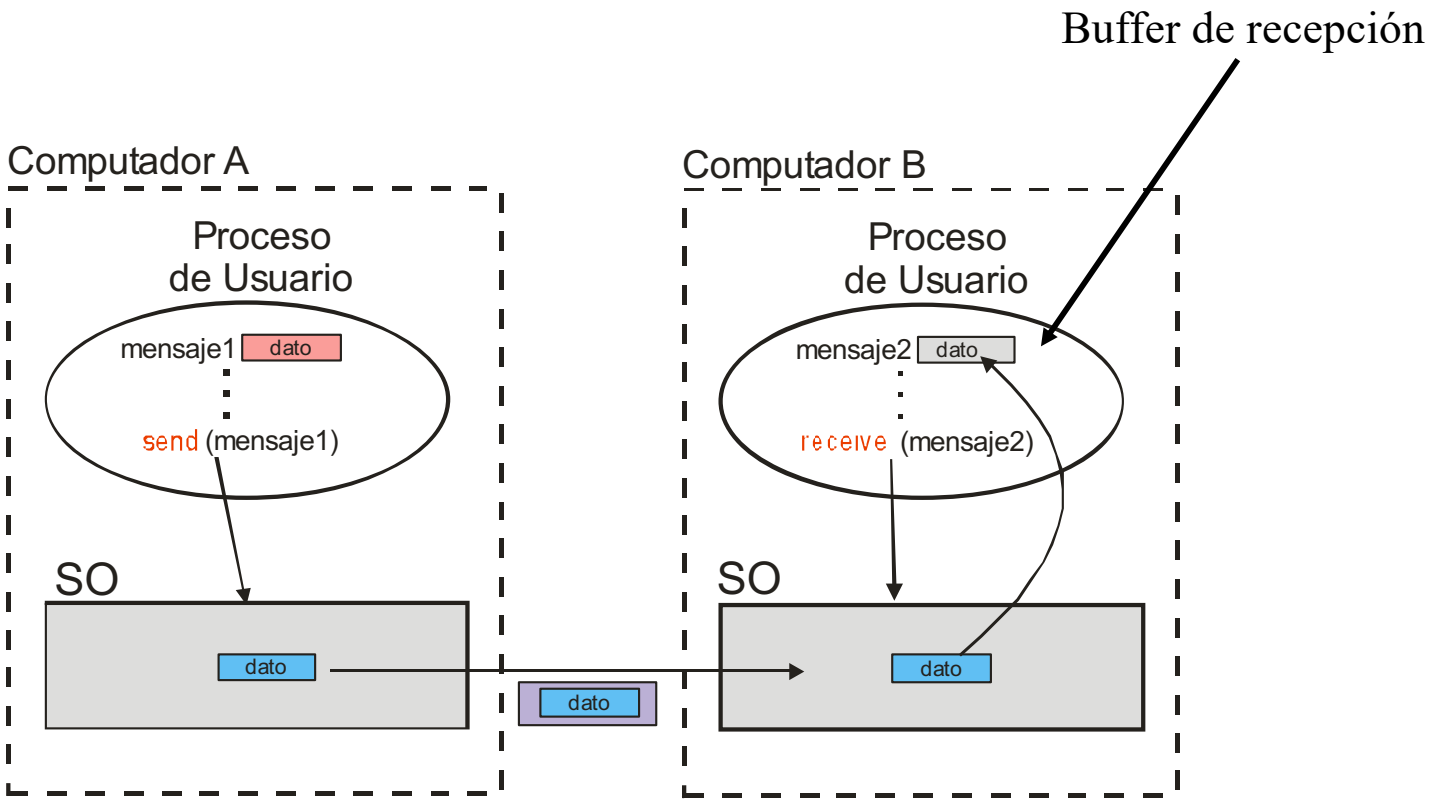
Proceso de comunicación



Proceso de comunicación



Proceso de comunicación



Paso de mensajes: conceptos básicos

- Tamaño del mensaje
  - Longitud fija o variable
- Flujo de datos
  - Bidireccional, unidireccional
- Nombrado
  - Comunicación directa
  - Comunicación indirecta
- Sincronización
- Almacenamiento (*buffering*)
- Fiabilidad
- Representación de datos, serialización

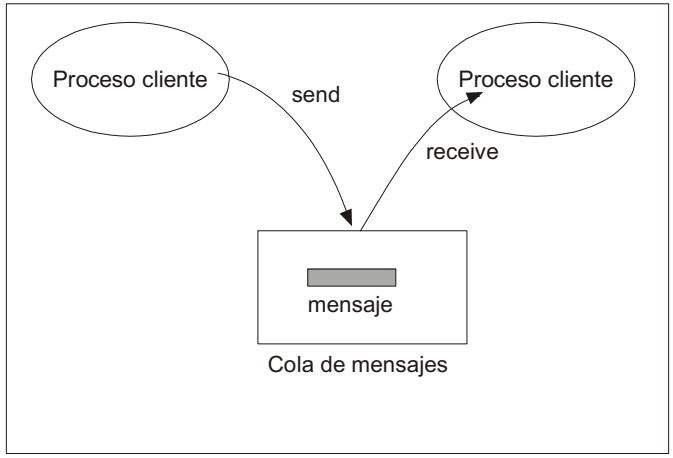
Nombrado

- Comunicación directa
  - send(P, m): envía un mensaje m al proceso P
  - receive(Q, m): recibe un mensaje del proceso Q
  - receive (ANY, m): recibe un mensaje de cualquiera

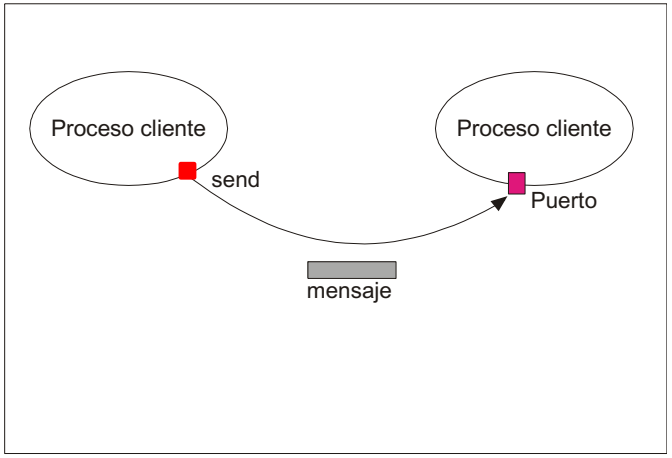
- Ejemplo: MPI (*Message Passing Interface*)
  - Interfaz de paso de mensajes para aplicaciones paralelas
  - Útil en clusters y arquitecturas paralelas heterogéneas

Nombrado

- Comunicación indirecta: los datos se envían a estructuras intermedias
  - Puertos: se asocian a un proceso (un único receptor)
    - Ejemplo: sockets
  - Colas de mensajes: múltiples emisores y receptores
    - Ejemplo: Colas de mensajes POSIX



Comunicación con colas de mensajes



Comunicación con puertos

Sincronización

- Operaciones síncronas (**bloqueante**)
  - La operación suspende al proceso hasta que la operación haya finalizado
- Asíncrona **(no bloqueante)**
  - La operación no bloquea al proceso que realiza la operación
- Envío asíncrono:
  - Dependiendo de la implementación del mecanismo de comunicación, no se garantiza que los datos sean entregados
- Recepción asíncrona:
  - Si los datos han llegado antes de invocar la operación, se entregan inmediatamente
  - Si los datos no han llegado:
    - La operación no recoge ningún dato e informa. Necesarios reintentos
    - Se notifica al receptor cuando lleguen los datos. Necesario proporcionar un manejador del evento

Almacenamiento

- El almacenamiento hace referencia a la capacidad del mecanismo de comunicación empleado:
  - Sin almacenamiento:
    - Comunicación entre los procesos emisor y receptor debe ser síncrona
  - Con almacenamiento: las estructuras intermedias (colas o puertos) tienen un cierto tamaño para almacenar mensajes
    - Si la cola no está llena, el emisor al enviar un mensaje lo inserta en ella y continúa su ejecución
    - Si la cola está llena el emisor se bloquea

Fiabilidad

- Operaciones que aseguran la entrega de mensajes de forma ordenada:
  - Ejemplo: TCP

- Operaciones que no aseguran la entrega de mensajes
  - Ejemplo: UDP

Formas de representación de los datos

- En memoria: objetos, estructuras, arrays, listas, árboles, …
- Secuencias de bytes almacenadas en ficheros o enviadas por la red
- Es necesario establecer un mecanismo para convertir datos en memoria en secuencias de bytes que puedan ser transferidas entre procesos o almacenadas en ficheros.
- Estos mecanismos de conversión tienen que tener en cuenta:
  - Heterogeneidad
    - Arquitectura (little-endian, big-endian)
    - Sistema operativo
    - Lenguaje de programación

Representación de datos

- Codificación, serialización o marshalling:
  - Proceso que convierte una estructura de datos u objeto en memoria en una secuencia de bytes que puede ser transmitida por la red o almacenada en ficheros
- Decodificación, deserialización o unmarchalling:
  - Proceso que convierte una secuencia de bytes en su correspondiente estructura de datos u objeto en memoria
- Tipos:
  - Formatos específicos de lenguajes de programación
  - Formatos específicos de middleware
  - Esquemas basados en texto que representan los datos utilizando cadenas de caracteres o texto codificado en una representación de tipo ASCII
    - Formatos estándar: JSON, XML

Formatos de codificación específicos de los lenguajes de programación

- Soporte en el propio lenguaje para codificar objetos de memoria en secuencias de bytes
- Java: java.io.Serializable
- Python: pickle

Serialización en Java

- Serializar un objecto:

```
OutputStream out = new OutputStream();
MyObject anObject = new MyObject();
ObjectOutputStream oOut = new ObjectOutputStream(out);
oOut.writeObject(anObject);
```

- Deserializar un objeto:

```
InputStream in = new InputStream();
ObjectInputStream oIn = new ObjectInputStream(in);
MyObject anObject = (MyObject)(oIn.readObject());
```

Serialización en Python

- Ejemplo: serializar una lista

```
import pickle

lista = [1,2,3,4,5]
fichero = open('lista.pckl','wb')   # formato binario
pickle.dump(lista, fichero)
fichero.close()
```

- Deserializar una lista

```
fichero = open('lista.pckl','rb')

lista_fichero = pickle.load(fichero)

print(lista_fichero)
fichero.close()
```

Desventajas de los formatos específicos de los lenguajes de programación

- Difícil intercambiar datos entre aplicaciones escritas en lenguajes de programación diferentes
- Problemas de compatibilidad entre versiones
- Problemas de rendimiento

Esquemas de codificación propios de middleware

- Esquemas de representación utilizados en mecanismos de de comunicación basados en llamadas a procedimientos remotos o invocación de métodos remotos
  - XDR (*External Data Representation*): formato empleado en ONC-RPC
  - CDR (*Common Data Representation*): formato empleado en CORBA
  - Protocol Buffers: esquema de codificación binario desarrollado por Google
- Son esquemas de codificación binaria que se basan en un lenguaje de definición de interfaces (IDL) que veremos en el tema 5

Ejemplo de codificación de datos en XDR, CDR y Protocol Buffers

- Mensaje genérico: Mensaje: 'Smith', 'London', 1934
- XDR:

```
struct Mensaje {
  string name<>;
  string place<>;
  int    year
};
```

CDR:

```
 struct Mensaje {
      string name;
      string place;
      long   year;
};
```

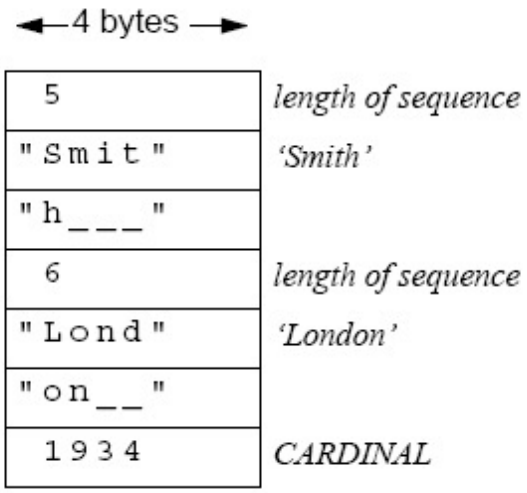
- Protocolo Buffers:

```
message Mensaje {
    required    string name = 1;
    required    string place = 2;
    int32       year= 3;
}
```

Ejemplo de codificación de datos en XDR y CDR

- Mensaje: 'Smith', 'London', 1934

XDR



CDR

index in sequence of bytes	4 bytes	*notes*
0–3	5	*length of string*
4–7	"Smit"	*'Smith'*
8–11	"h___"	
12–15	6	*length of string*
16–19	"Lond"	*'London'*
20-23	"on__"	
24–27	1934	*unsigned long*

- Desventajas: solo son válidos para aplicaciones desarrolladas en un middleware concreto (XDR, CORBA o Protocol Buffers)

Esquemas de codificación basados en texto

- XML (Extensible Markup Language)
- JSON(JavaScript Object Notation)
- CSV (Comma-Separated Values)

Ejemplo de codificación de datos

- Mensaje: 'Smith', 'London', 1934

XML

```
<person>
        <name>Smith</name>
        <place>London</place>
        <year>1934</year>
</person >
```

JSON

```
{ "person": {
      "name": "Smith",
      "place" : "London",
      "year": "1934"
  }
}
```

CSV

```
Smith, London, 1934
```

Esquemas de codificación basados en texto

- Ventajas:
  - Mayor interoperabilidad: independientes de los lenguajes de programación, arquitecturas, sistemas operativos y mecanismos de comunicación
- Desventajas:
  - Mayor sobrecarga de procesamiento y envío de datos

XML

```
<person>
        <name>Smith</name>
        <place>London</place>
        <year>1934</year>
</person >
```

MPI

- Message Passing Interface
- Interfaz estándar de paso de mensajes para el desarrollo de aplicaciones paralelas que ejecutan en computadores que no comparten memoria

Características de MPI

- Portabilidad:
  - Definido independiente de plataforma paralela.
  - Útil en arquitecturas paralelas heterogéneas.

- Eficiencia:
  - Definido para aplicaciones multihilo (*multithread*)
  - Sobre una comunicación fiable y eficiente.
  - Busca el máximo de cada plataforma.

- Funcionalidad:
  - Fácil de usar por cualquier programador que ya haya usado cualquier biblioteca de paso de mensajes.

Características de MPI

- Estructuras de datos
  - Tipos de datos (básicos, vectores, compuestos, …)
  - Grupo de procesos (grupos, comunicadores, …)
- Paso de mensajes
  - Llamadas punto a punto (bloqueantes, asíncronas)
  - Llamadas colectivas (*bcast*, *scatter*, *gather*, …)
- Entrada y salida
  - Gestión de ficheros (apertura, cierre, …)
  - Gestión de contenidos (vistas, punteros, …)
- Procesos
  - Gestión de procesos (creación, …)
  - *Profiling*

¿Cómo es MPI?

```
#include <stdio.h>
#include "mpi.h"

main(int argc, char **argv)
{
        int node,size;
        int tam = 255;
        char name[255];

        MPI_Init(&argc,&argv);

        MPI_Comm_size(MPI_COMM_WORLD, &size );
        MPI_Comm_rank(MPI_COMM_WORLD, &node);
        MPI_Get_processor_name(name, &tam);
        printf("El proceso %d de %d procesos (%s)\n",node, size, name);

        MPI_Finalize();
}
```

Todos los procesos ejecutan el mismo programa

Send-receive en MPI

send-recv.c

```
#include <stdio.h>
#include "mpi.h"

main(int argc, char **argv)  {
        int node,size;
        int tam = 255;
        char name[255];
        int num = 10;

        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD, &size );
        MPI_Comm_rank(MPI_COMM_WORLD, &node);

        if (node == 0)
             MPI_Send(&num, 1 MPI_INT, 1, 0, MPI_COMM_WORLD);
        else
             MPI_Recv(&num, 1 MPI_INT, 0, 0, MPI_COMM_WORLD);

        MPI_Finalize();
}
```

Send-receive en MPI

```
#include <stdio.h>
#include "mpi.h"

main(int argc, char **argv)  {
        int node,size;
        int tam = 255;
        char name[255];
        int num = 10;

        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD, &size );
        MPI_Comm_rank(MPI_COMM_WORLD, &node);

        if (node == 0)
             MPI_Send(&num, 1 MPI_INT, 1, 0, MPI_COMM_WORLD);
        else
             MPI_Recv(&num, 1 MPI_INT, 0, 0, MPI_COMM_WORLD);

        MPI_Finalize();
}
```

Tipo de datos

- MPI_CHAR
- MPY_BYTE
- MPI_INT
- MPI_FLOAT
- ….
- Tipos de datos derivados

Dato a enviar

Proceso destinatario

Etiqueta asociada al mensaje

comunicador

Número de elementos

Send-receive en MPI

```
#include <stdio.h>
#include "mpi.h"

main(int argc, char **argv)  {
        int node,size;
        int tam = 255;
        char name[255];
        int num = 10;

        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD, &size );
        MPI_Comm_rank(MPI_COMM_WORLD, &node);

        if (node == 0)
             MPI_Send(&num, 1 MPI_INT, 1, 0, MPI_COMM_WORLD);
        else
             MPI_Recv(&num, 1 MPI_INT, 0, 0, MPI_COMM_WORLD);

        MPI_Finalize();
}
```

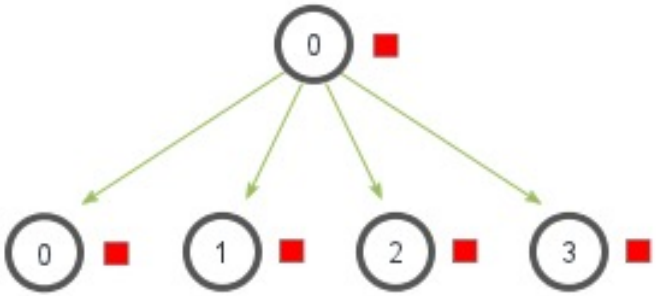
Tipo de datos

- MPI_CHAR
- MPY_BYTE
- MPI_INT
- MPI_FLOAT
- ….
- Tipos de datos derivados

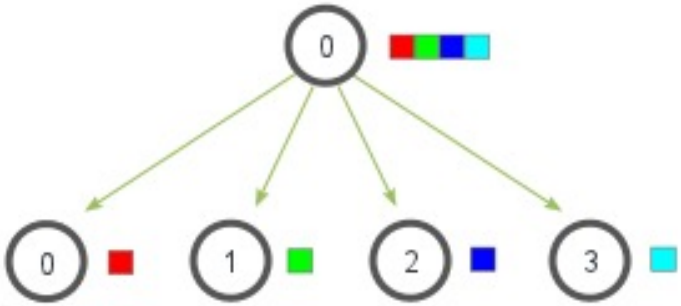
Proceso emisor del mensaje

Operaciones colectivas

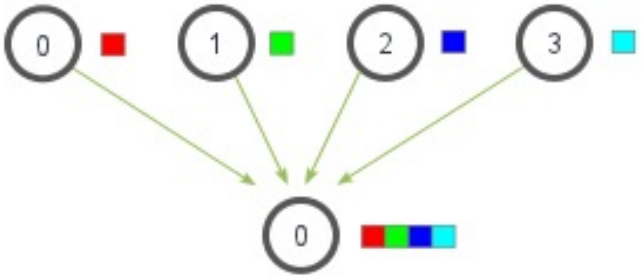
MPI_Bcast



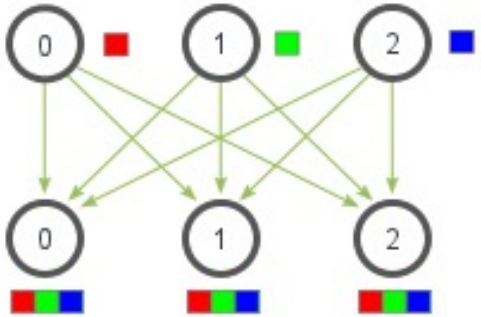
MPI_Scatter



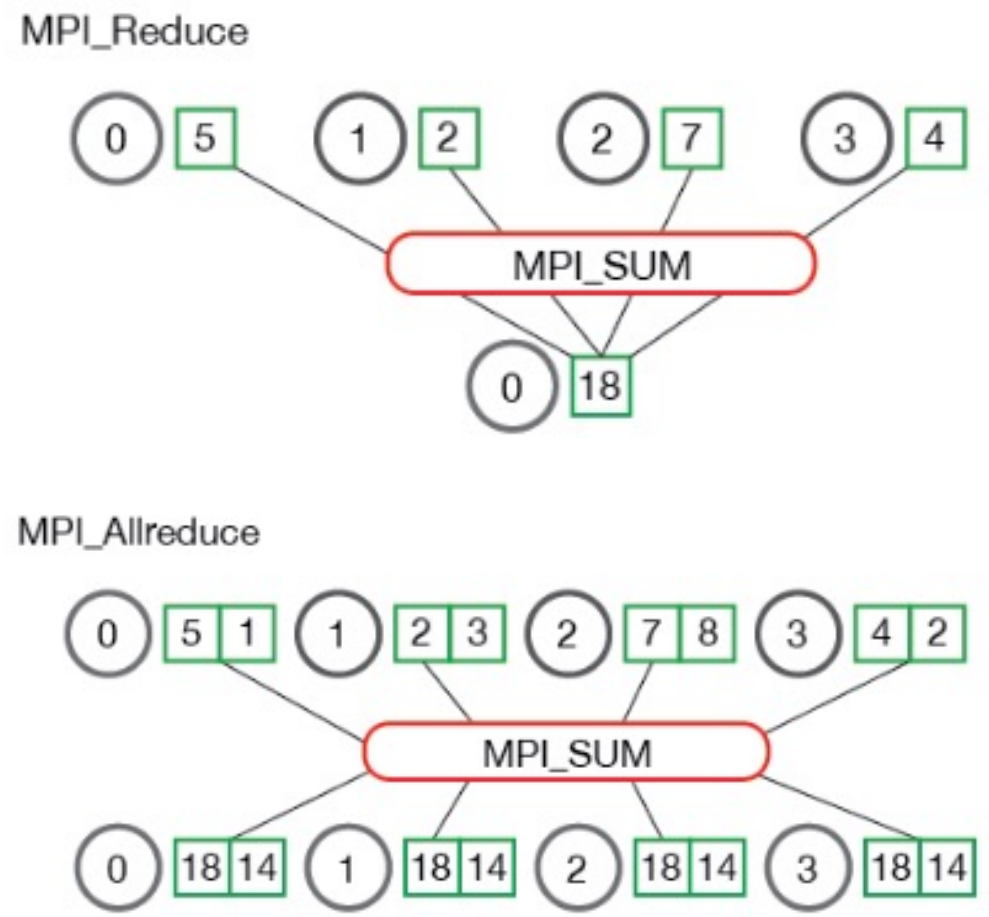
MPI_Gather



MPI_Allgather



Operaciones de reducción (MPI_Reduce)



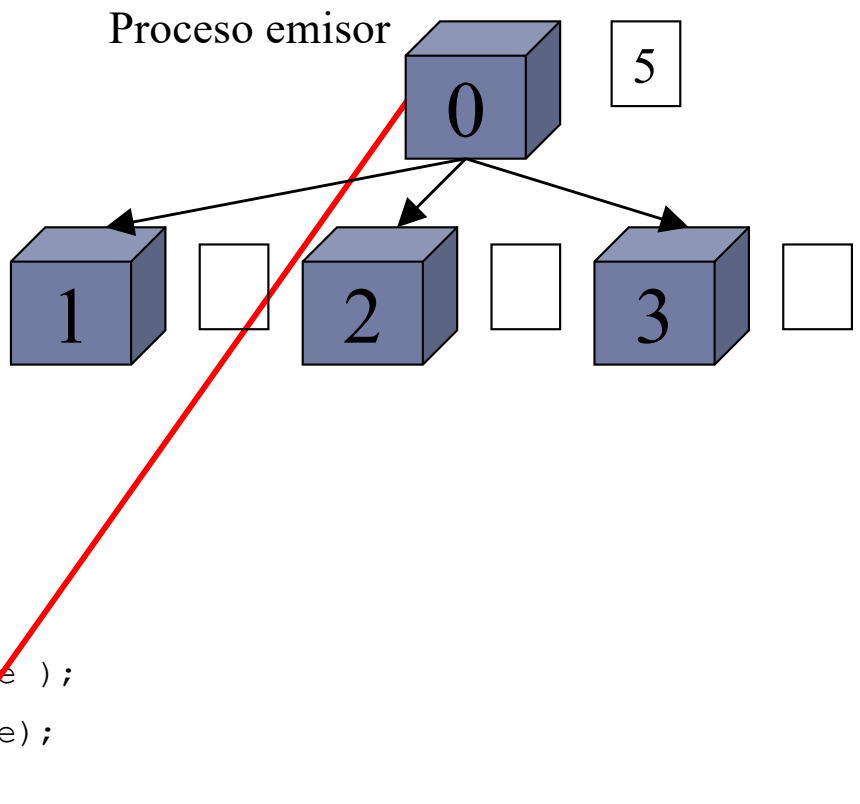
Operaciones colectivas en MPI

bcast.c

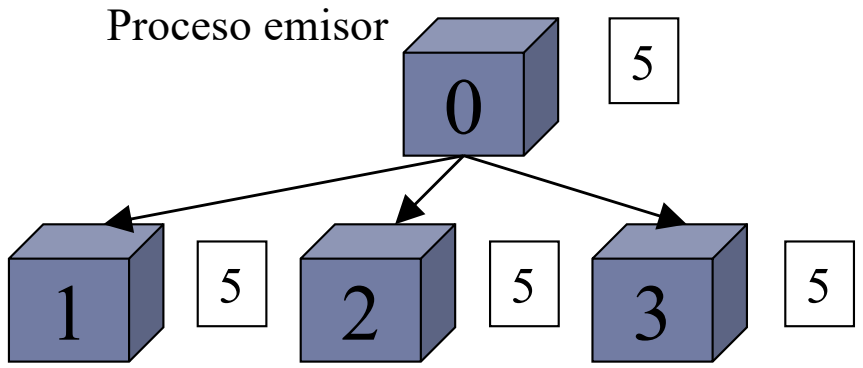
```
#include <stdio.h>
#include "mpi.h"

main(int argc, char **argv)  {
        int node,size;
        int tam = 255;
        char name[255];
        int num;

        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD, &size );
        MPI_Comm_rank(MPI_COMM_WORLD, &node);
        if (node == 0)
              num = 5;
        MPI_Bcast(&num, 1, MPI_INT, 0, MPI_COMM_WORLD);

        MPI_Barrier(MPI_COMM_WORD);
        printf("El proceso %d recibe %d\n",node, num);
        MPI_Finalize();
}
```

Operaciones colectivas en MPI

```
#include <stdio.h>
#include "mpi.h"

main(int argc, char **argv)  {
        int node,size;
        int tam = 255;
        char name[255];
        int num;

        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD, &size );
        MPI_Comm_rank(MPI_COMM_WORLD, &node);
        if (node == 0)
              num = 5;
        MPI_Bcast(&num, 1, MPI_INT, 0, MPI_COMM_WORLD);

        MPI_Barrier(MPI_COMM_WORD);
        printf("El proceso %d recibe %d\n",node, num);
        MPI_Finalize();
}
```



Operaciones colectivas en MPI

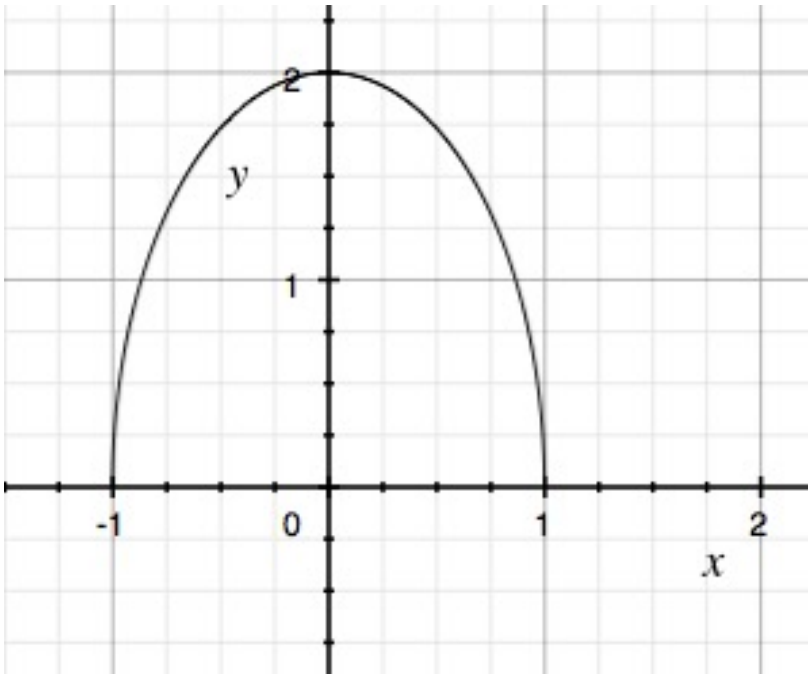


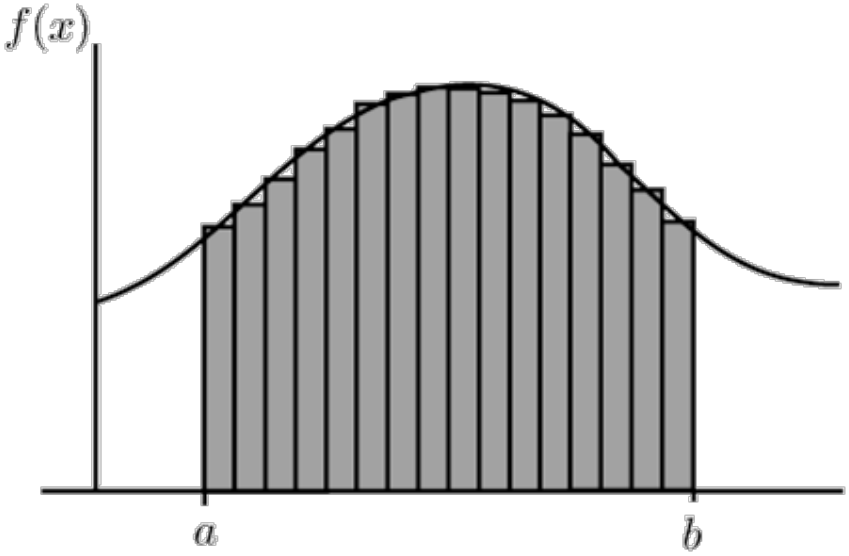

```
#include <stdio.h>
#include "mpi.h"

main(int argc, char **argv)  {
        int node,size;
        int tam = 255;
        char name[255];
        int num;

        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD, &size );
        MPI_Comm_rank(MPI_COMM_WORLD, &node);
        if (node == 0)
              num = 5;
        MPI_Bcast(&num, 1, MPI_INT, 0, MPI_COMM_WORLD);

        MPI_Barrier(MPI_COMM_WORD);
        printf("El proceso %d recibe %d\n",node, num);
        MPI_Finalize();
}
```

El resto de procesos recibe en num el mensaje enviado por el proceso 0

Ejemplo: Cálculo de π

$$\int_0^1 \sqrt{4\left(1-x^2\right)}\,dx = \frac{\pi}{2}$$





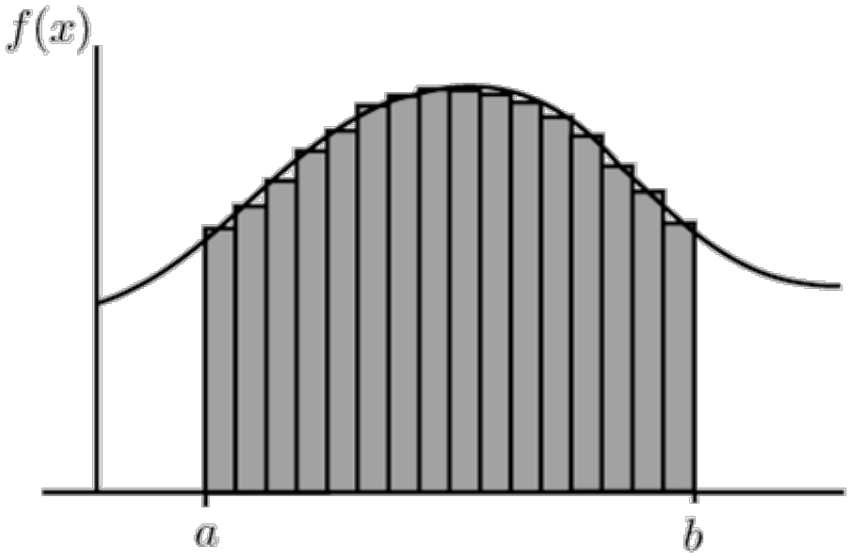
Fuente: Riemann Integration 5.png, Wikimedia Commons, CC BY-NC-SA 2.0

Cálculo secuencial

pi-secuencial.c

```
#include <stdio.h>
#include <math.h>
#include <stdio.h>

#define N 1E7
#define d 1E-7

int main (int argc, char* argv[]) {
    int i;
    double pi=0.0, result=0.0, sum=0.0, x2, x =0, s;

    result = 0;
    for (i=0; i<N; i++) {
        x2 = x * x;
        s = sqrt(1-x2) * d;
        result = result + s;
        x = x + d;
    }

    pi=4*result;
    printf("PI=%lf\n", pi);

    return 0;
}
```

$$\int_0^1 \sqrt{4\left(1-x^2\right)}\,dx = \frac{\pi}{2}$$



Fuente: Riemann Integration 5.png, Wikimedia Commons, CC BY-NC-SA 2.0

Cálculo paralelo con MPI

pi-mpi.c

```
#include <math.h>
#include <stdio.h>
#include <mpi.h>

#define N 1E7
#define d 1E-7

int main (int argc, char* argv[]) {
    int rank, size, i;
    double pi=0.0, result=0.0, sum=0.0, x2, s = 0, x = 0;


    MPI_Init (&argc, &argv);

    MPI_Comm_rank (MPI_COMM_WORLD, &rank);  // id del proceso
    MPI_Comm_size (MPI_COMM_WORLD, &size);  // num. procesos
```

Cálculo paralelo con MPI

pi-mpi.c

```
    //Cada proceso calcula una parte en paralelo
    for (i=rank; i<N; i+=size){
            x2 = x * x;
            s = sqrt(1-x2) * d;
            result = result + s;
            x = x + d * size;
    }

    // se suman todos los resultados
    MPI_Reduce(&result, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

     // calcula pi e imprime
    if (rank==0) {
        pi=4*sum;
        printf("np=%2d;  PI=%lf\n", size, pi);
    }
    MPI_Finalize();
    return 0;
}
```

Colas de mensajes POSIX

- Implementación de paso de mensajes en POSIX
- Mecanismo de comunicación **y** sincronización
- Nombrado indirecto
- Asignan a las colas nombres de ficheros
  - Sólo pueden utilizar colas de mensajes procesos que comparten un mismo sistema de ficheros
- Tamaño del mensaje variable
- Flujo de datos bidireccional
- Sincronización:
  - Envío asíncrono
  - Recepción síncrona o asíncrona
- Los mensajes se pueden etiquetar con prioridades

Llamadas al sistema de POSIX para gestión de colas de mensajes

- Crear y abrir una cola de mensajes
- Cerrar una cola de mensajes
- Borrar una cola de mensajes
- Modificar los atributos
- Enviar
- Recibir

Crear y abrir una cola de mensajes (I)

- Crear una cola de mensajes con nombre y atributos

```
mqd_t mq_open( char *name,
               int flag,
               mode_t mode,
               struct mq_attr *attr)
```

  - El primer argumento especifica el nombre de la cola
    - Sigue la convención de nombrado de archivos
    - **man 7 mq_overview**

  - El segundo argumento especifica los flags de la cola (definidos en fnctl.h):

O_CREAT	Crea una cola si no existe
O_RDONLY	Crea una cola para recibir
O_WRONLY	Crea una cola para enviar
O_RDWR	Crea una cola para recibir y enviar
O_NONBLOCK	Envío y recepción no bloqueante

Crear y abrir una cola de mensajes (II)

- Crear una cola de mensajes con nombre y atributos

```
mqd_t mq_open( char *name,
               int flag,
               mode_t mode,
               struct mq_attr *attr)
```

  - El tercer argumento son los permisos de acceso a la cola (definido en sys/stat.h):
    - O_RDONLY, O_WRONLY, O_RDWR
  - El cuarto argumento especifica los atributos de la cola. Se usa la estructura `attr`, que entre otros campos define:

    `mq_maxmsg` Número máximo de mensajes

    `mq_msgsize` Tamaño máximo del mensaje
  - Si no se especifica se toman los atributos por defecto
  - La llamada `mq_open` devuelve un descriptor de cola o -1 si error

Cerrar y borrar una cola de mensajes

- Cerrar una cola de mensajes

```
int mq_close(mqd_t mqdes)
```

- Borrar una cola de mensajes
  - Después de cerrar la cola de mensajes

```
int mq_unlink(char *name)
```

Ejemplo: Crear una cola de mensajes

```
#include <stdio.h>
#include <mqueue.h>
#include <stdlib.h>

#define NUM_MENSAJES    10
int main () {
        mqd_t mqd;                      /* Descriptor de la cola */
        struct mq_attr atributos;       /* Atributos */

        atributos.mq_maxmsg = NUM_MENSAJES;
        atributos.mq_msgsize = sizeof(int);

        if ((mqd=mq_open("/almacen.txt", O_CREAT|O_WRONLY, 0777, &atributos))==-1){
                printf("Error mq_open\n");
                exit(-1);
        }
        // Escribir en la cola
        mq_close(mqd);
        mq_unlink("/almacen.txt");
}
```

Ejemplo: Crear una cola de mensajes

```
#include <stdio.h>
#include <mqueue.h>
#include <stdlib.h>

#define NUM_MENSAJES     10
int main () {
        mqd_t mqd;                      /* Descriptor de la cola */
        struct mq_attr atributos;       /* Atributos */

        atributos.mq_maxmsg = NUM_MENSAJES;
        atributos.mq_msgsize = sizeof(int);

        if ((mqd=mq_open("/almacen.txt", O_CREAT|O_WRONLY, 0777, &atributos))==-1){
                printf("Error mq_open\n");
                exit(-1);
        }
        // Escribir en la cola
        mq_close(mqd);
        mq_unlink("/almacen.txt");
}
```

En la mayoría de sistemas Linux el número máximo de mensajes está limitado a 10 para procesos sin privilegios

Ejemplo: Compilación

- Es necesario enlazar con la biblioteca **librt**

gcc filename.c **–lrt** –o executable

Gestión de los atributos

- Modificar los atributos de una cola

```
int mq_setattr(mqd_t mqdes,
               struct mq_attr *qstat,
               struct mq_attr *oldmqstat)
```

  - Permite cambiar los atributos de una cola de mensajes
  - Los nuevos atributos se pasan en qstat
  - Si oldmqstat es distinto de NULL se almacenarán en él los antiguos atributos
- Obtener los atributos de una cola

```
int mq_getattr (mqd_t mqdes,  struct mq_attr *qstat)
```

  - Devuelve los atributos de una cola de mensajes
    - `mq_maxmsg`: Número máximo de mensajes
    - `mq_msgsize`: Tamaño máximo del mensaje
    - `mq_curmsgs`: Número actual de mensajes de la cola
    - `mq_flags`: flags asociados a la cola

Ejemplo: Recuperar los atributos

```
#include <stdio.h>
#include <mqueue.h>
#include <stdlib.h>

int main () {
        mqd_t mqd;                      /* Descriptor de la cola */
        struct mq_attr atributos;       /* Atributos */
        if ((mqd=mq_open("/almacen.txt", O_CREAT|O_WRONLY, 0777, &atributos))==-1){
                printf("Error mq_open\n");
                exit(-1);
        }
        if (mq_getattr(mqd, &atributos) == -1){
                printf("Error mq_getattr\n");
                exit(-1);
        }
        printf("Numero maximo de mensajes en la cola: %ld\n", atributos.mq_maxmsg);
        printf("Longitud máxima del mensaje en la cola: %ld\n", atributos.mq_msgsize);
        printf("Numero de mensajes actualmente en la cola: %ld\n", atributos.mq_curmsgs);
        exit(0);
}
```

Comunicación y sincronización

- Las colas de mensajes ofrecen un mecanismo de comunicación **y** sincronización:
  - **Comunicación**:
    - El nombre de la cola permite compartir ésta para que múltiples procesos puedan enviar y recibir datos
  - **Sincronización**
    - Semántica bloqueante
    - No bloqueante **(O_NONBLOCK)**

Servicios POSIX para enviar mensajes

- Enviar un mensaje a una cola

```
int mq_send(mqd_t mqdes,   const char *msg,   size_t len,
                unsigned int prio)
```

  - El mensaje msg de longitud len se envía a la cola de mensajes mqdes con prioridad prio
  - Argumentos de entrada

`mqdes`	Descriptor de la cola
`msg`	Buffer que contiene el mensaje a enviar
`len`	Longitud del mensaje a enviar
`prio`	Prioridad del mensaje. Los mensajes se insertan según su prioridad (0...MQ_PRIOMAX)

  - Si la cola está llena el envío puede ser bloqueante o no (depende de **O_NONBLOCK**)
  - Si es O_NONBLOCK y la cola está llena el proceso no se bloquea pero devuelve -1 (EAGAIN)
  - Devuelve 0 en caso de éxito o -1 en caso de error

Servicios POSIX para recibir mensajes

- Recibir un mensaje desde una cola

```
int mq_receive(mqd_t mqdes,    char *msg,
               size_t len,    unsigned int *prio)
```

  - Recibe el mensaje msg con mayor prioridad en la cola (prio) de longitud len de la cola de mensajes mqdes
  - Argumentos de entrada

`mqdes`	Descriptor de la cola
`msg`	Buffer que contiene el mensaje a recibir
`len`	Longitud del mensaje a recibir
`prio`	Prioridad del mensaje recibido.

  - Si la cola está vacía la recepción puede ser bloqueante o no (depende de **O_NONBLOCK**)
  - Si es O_NONBLOCK y la cola está vacía el proceso no se bloquea pero devuelve -1 (EAGAIN)
  - Devuelve el número de bytes del mensaje recibido o -1 en caso de error.

Productor-consumidor con paso de mensajes

```
Productor () {
  for(;;) {
       <Producir dato>
        send(Consumidor, dato);
  } /* end for */
}
```

```
Consumidor () {
  for(;;) {
     receive(Productor, dato);
     <Consumir dato>
  } /* end for */
}
```

Ejemplo:
Productor-consumidor con colas de mensajes

productor.c

```
#include <mqueue.h>
#include <stdio.h>
#include <stdlib.h>
#define MAX_BUFFER              10    /* tamaño del buffer */
#define DATOS_A_PRODUCIR        100000 /* datos a producir */
int  main(void){
        mqd_t almacen;      /* cola de mensajes donde dejar los
                            datos producidos y extraer los datos a consumir */
        struct mq_attr attr;
        attr.mq_maxmsg = MAX_BUFFER;
        attr.mq_msgsize = sizeof (int);
        almacen = mq_open("/ALMACEN", O_CREAT|O_WRONLY, 0700, &attr);
        if (almacen == -1){
                perror ("mq_open");
                exit(-1);
        }
        Productor(almacen);
        mq_close(almacen);
        return(0);
}
```

Ejemplo:
Productor-consumidor con colas de mensajes

productor.c

```
void Productor(mqd_t cola){
        int dato;
        int i;

        for(i=0;i<DATOS_A_PRODUCIR;i++){
                /* producir dato */
                dato = i;
                if (mq_send(cola, (const char *)&dato, sizeof(int), 0)== -1){
                        perror("mq_send");
                        mq_close(cola);
                        exit(1);
                }
        } /* end for */
        return;
} /* end productor */
```

Ejemplo:
Productor-**consumidor** con colas de mensajes

consumidor.c

```
#include <mqueue.h>
#include <stdio.h>
#include <stdlib.h>

#define MAX_BUFFER                10    /* tamaño del buffer */
#define DATOS_A_PRODUCIR          100000 /* datos a producir */
int main(void){
        mqd_t almacen;        /* cola de mensajes donde dejar los    datos
                               producidos y extraer los datos a consumir */

        almacen = mq_open("/ALMACEN", O_RDONLY);
        if (almacen == -1){
                perror ("mq_open");
                exit(-1);
        }
        Consumidor(almacen );
        mq_close(almacen);
        return(0);
}
```

Ejemplo:
Productor-**consumidor** con colas de mensajes

consumidor.c

```
void Consumidor(mqd_t cola){
        int dato;
        int i;

        for(i=0;i<DATOS_A_PRODUCIR;i++){
                /* recibir dato */
                dato = i;
                if (mq_receive(cola, (char *)&dato, sizeof(int),  0)== -1){
                        perror("mq_receive");
                        mq_close(cola);
                        exit(1);
                }
                /* Consumir el dato */
                printf("El dato consumido es: %d\n", dato);
        } /* end for */
        return;
} /* end consumidor */
```

Otros modelos de colas de mensajes

- Advanced Message Queueing Protocol (AMQP)
- Apache ActiveM!
- IBM MQ
- Java Message Service
- Microsoft Message Queing

Aspectos básicos de la comunicación entre procesos

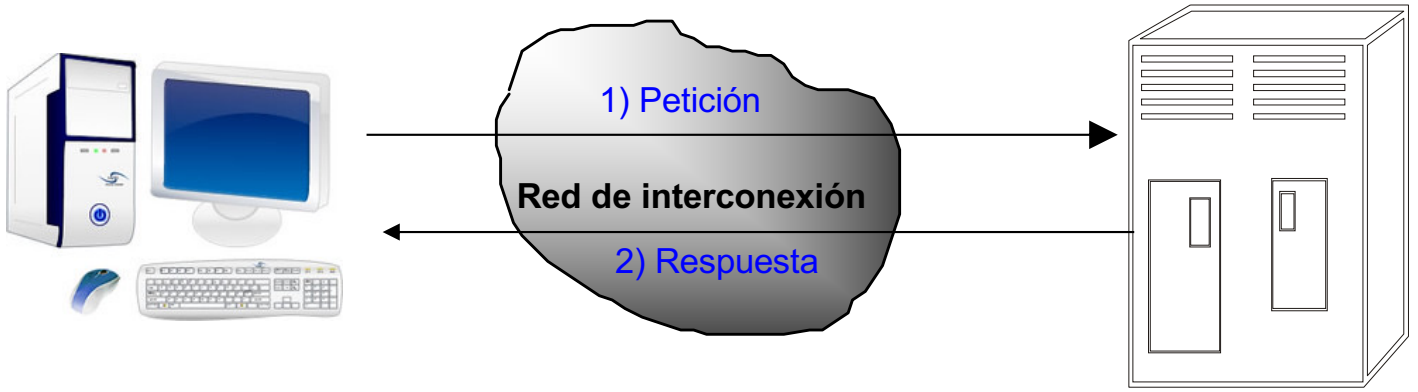
- Receptor capaz de recibir un mensaje
- Emisor capaz de enviar un mensaje
  - mensaje inicialmente almacenado en la memoria del proceso emisor
- El emisor debe conocer (o descubrir) la dirección del receptor
- Sistema de transmisión entre los dos procesos
- Acuerdo sobre el formato y codificación de los mensajes a intercambiar

Formas de comunicación

- Comunicación unidireccional
- Comunicación bidireccional
- Comunicación petición-respuesta (cliente-servidor), caso particular del anterior
- Comunicación multicast

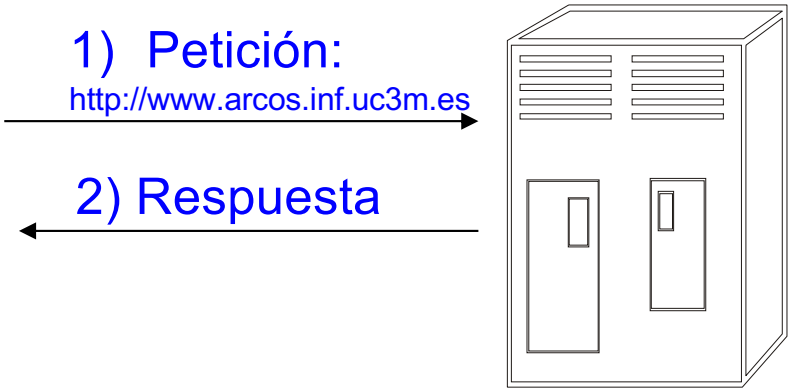
Modelo cliente-servidor

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



Fuente: pc, by maido155, CC BY-NC-SA 2.0

Ejemplo: HTTP



Cliente **Servidor**

1) Petición:

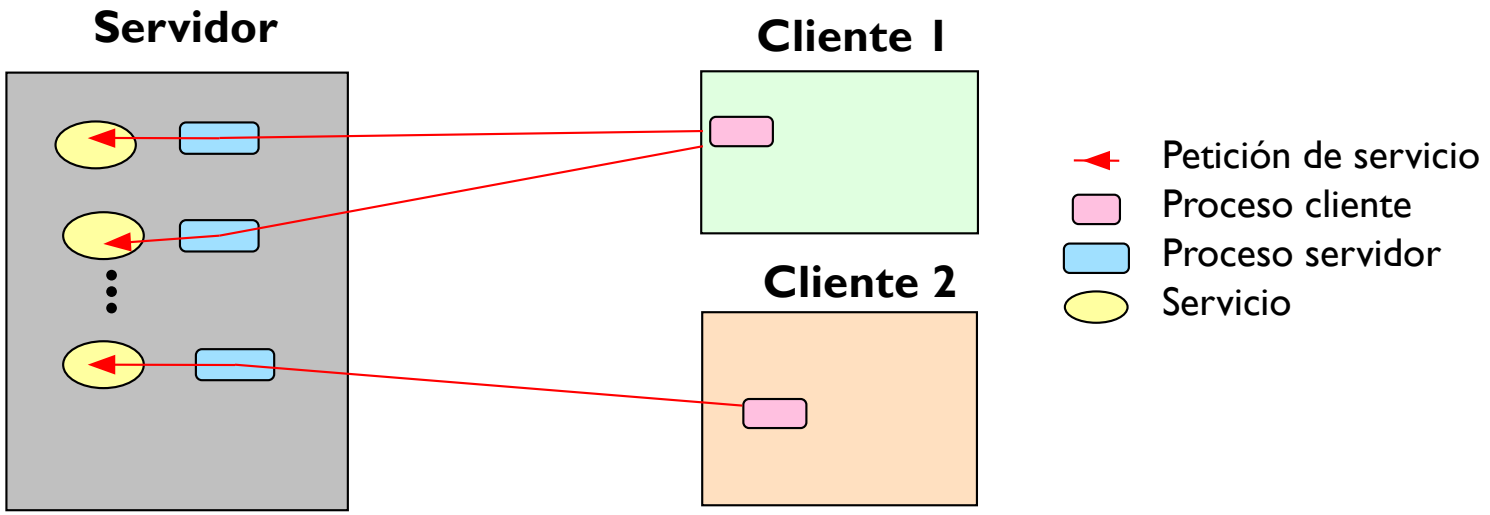
```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: nombre-cliente
[Línea en blanco]
```

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>
```

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
  - Colas de mensajes
  - Llamada a procedimientos remotos (**RPC**)
  - Invocación de **métodos remotos** (RMI, CORBA, …).
  - Servicios Web
- Paradigma principalmente adecuado para servicios centralizados
- Ejemplos: servicios de Internet (HTTP, FTP, DNS, …)

Tipos de servidores de aplicaciones

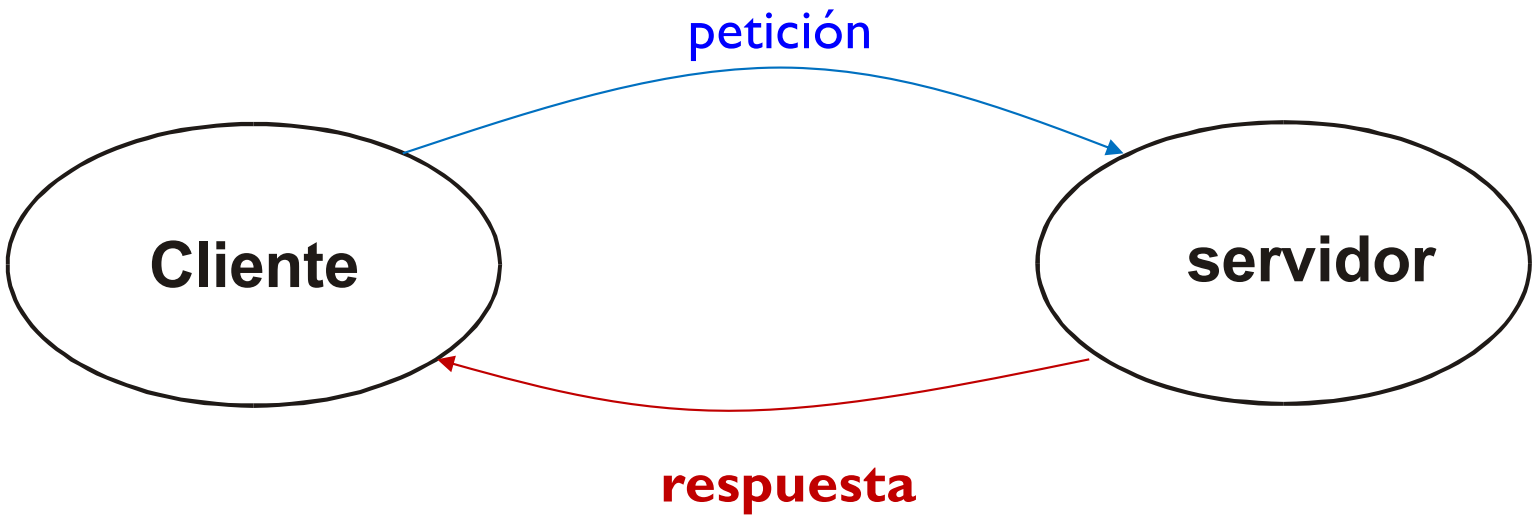
- En función del número de peticiones que es capaz de atender:
  - Secuencial: una petición
  - Concurrente: múltiples peticiones atendidas al mismo tiempo

- En función de si existe una conexión preestablecida con el cliente
  - Servidores orientados a conexión
  - Servidores NO orientados a conexión

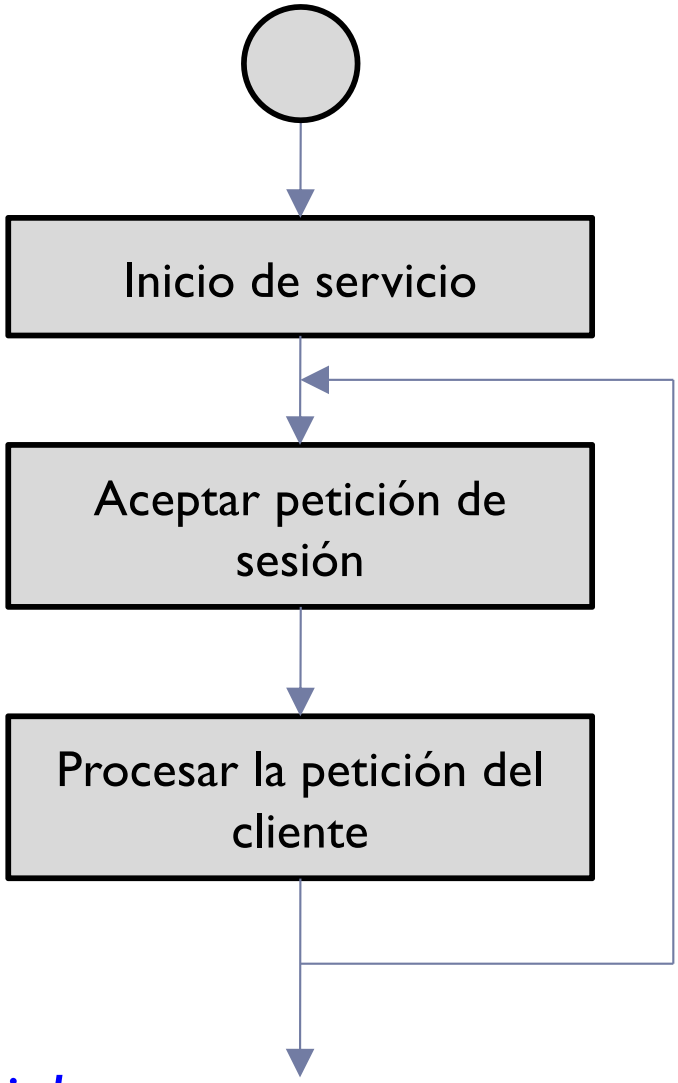
- En función de si almacena o no el estado de la comunicación
  - Servidores con estado
  - Servidores sin estado

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

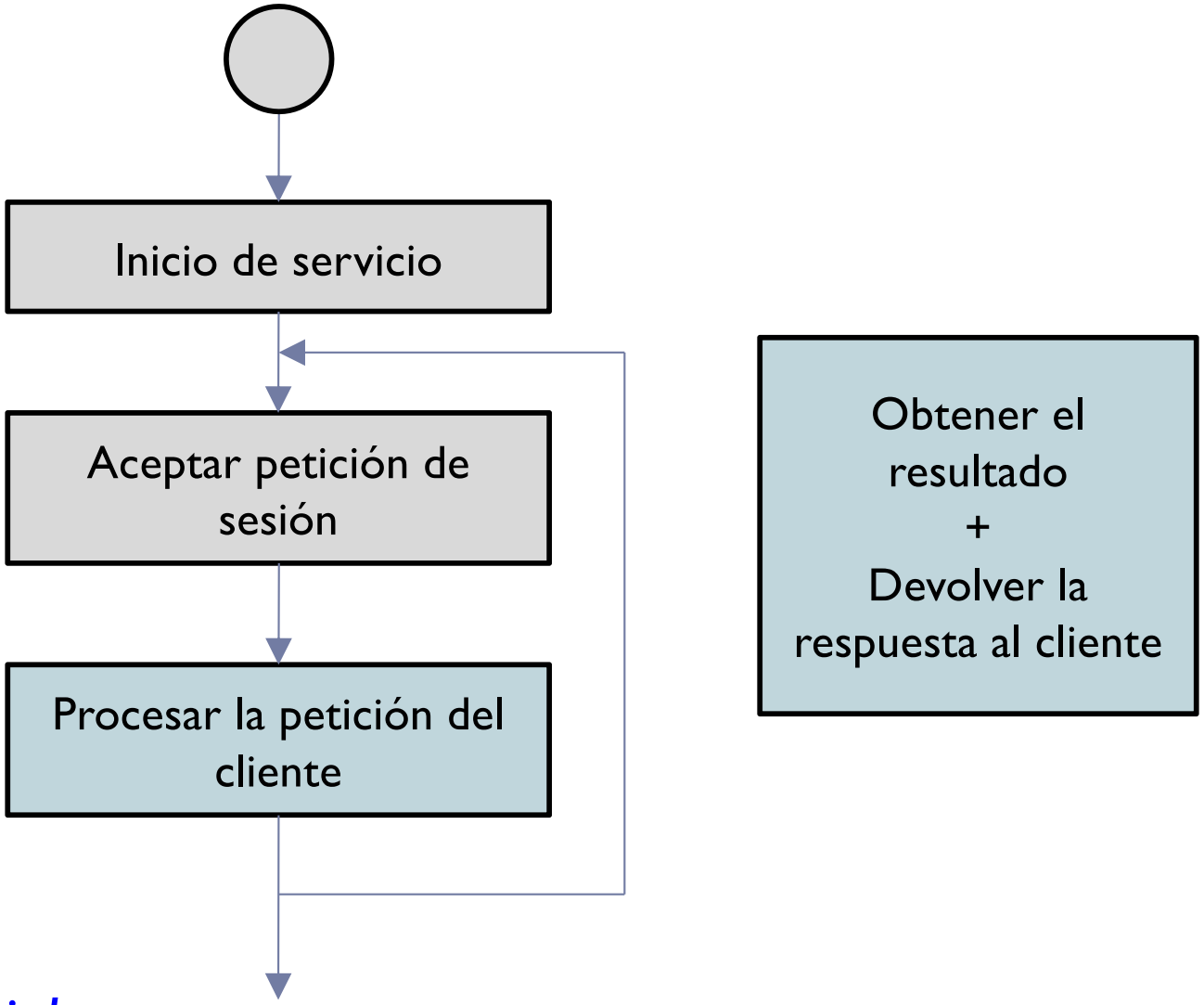


Flujo de ejecución de un servidor secuencial



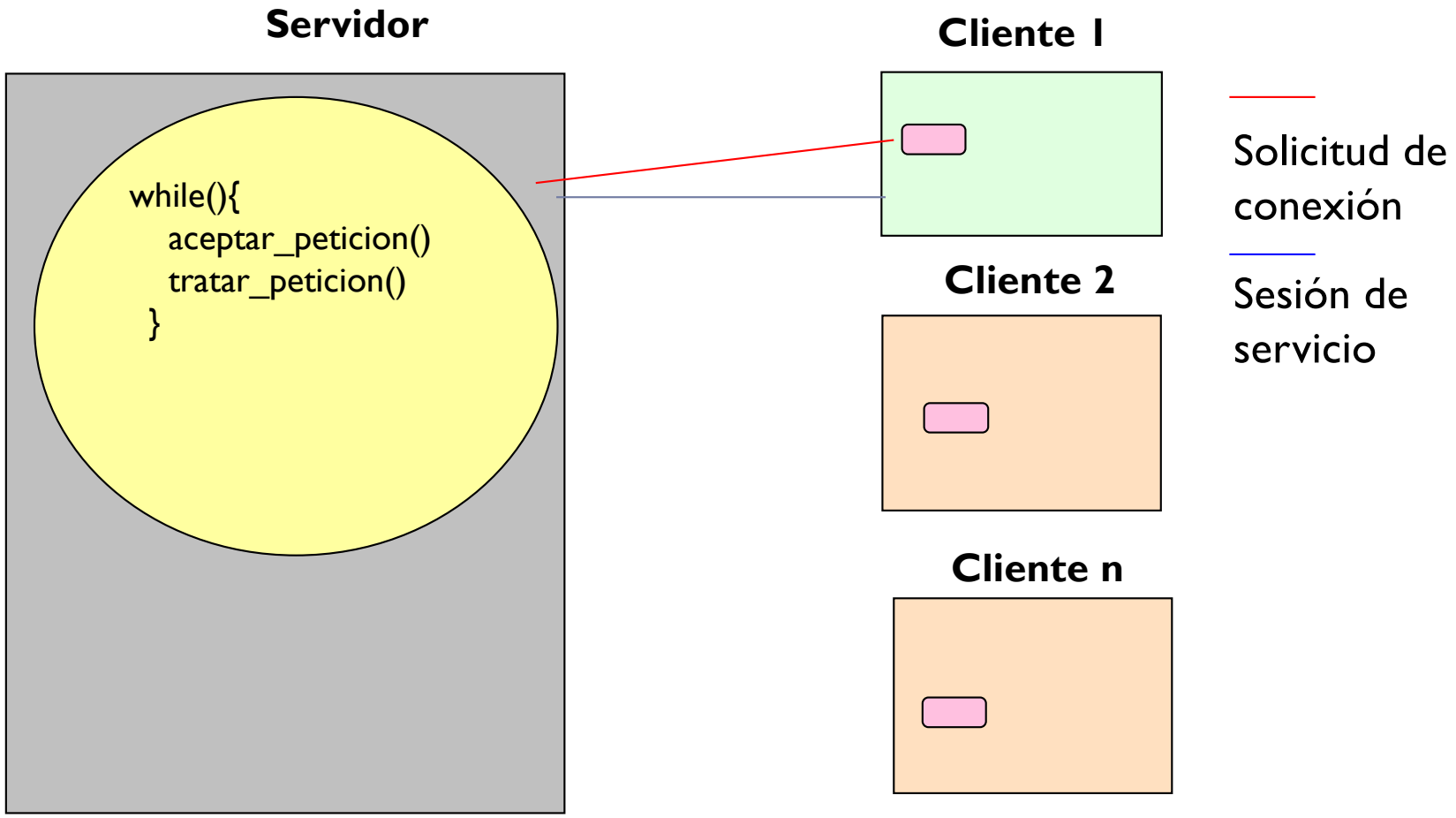
Servidor secuencial

Flujo de ejecución de un servidor secuencial

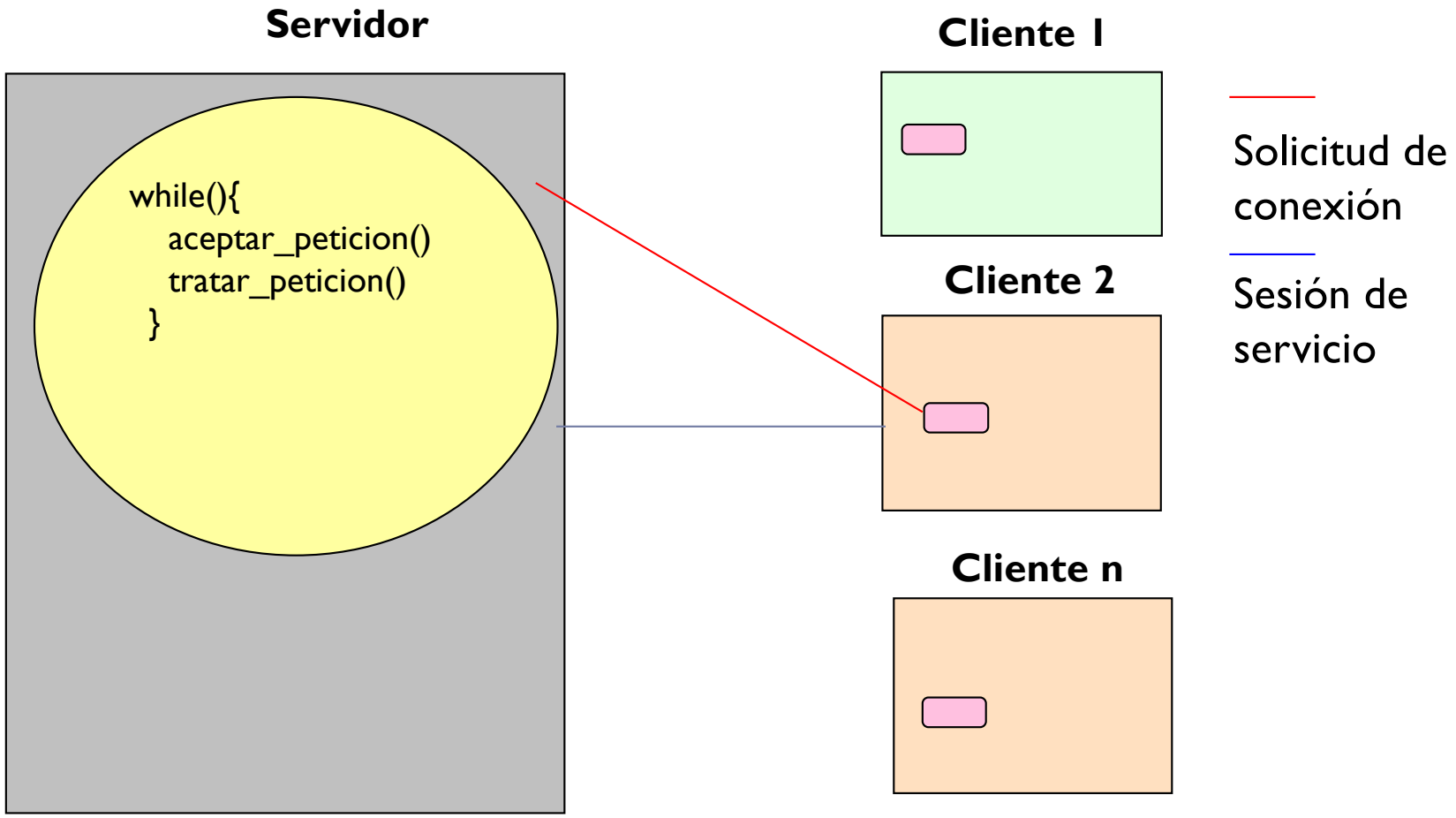


Servidor secuencial

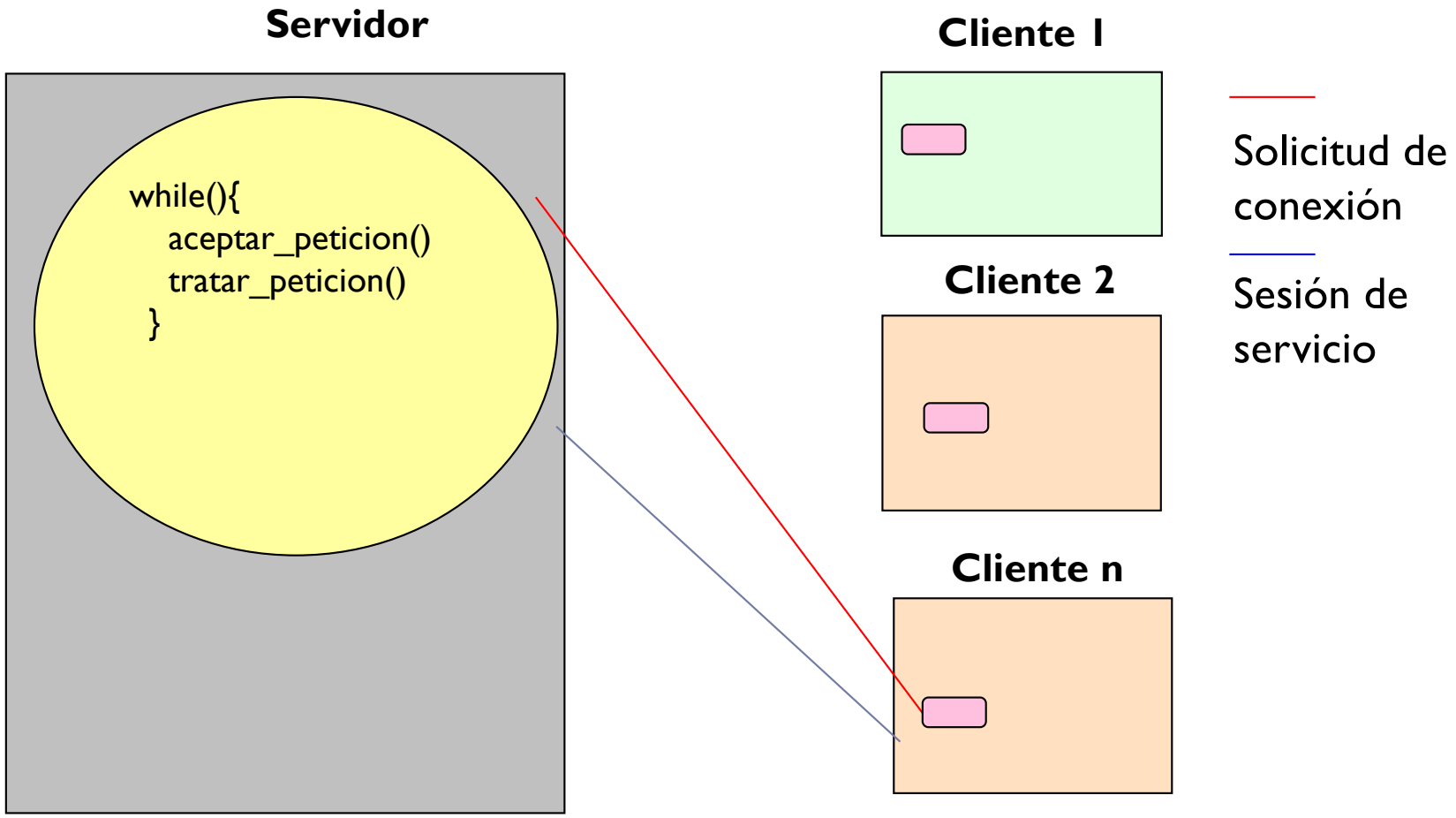
Cliente-Servidor secuencial



Cliente-Servidor secuencial

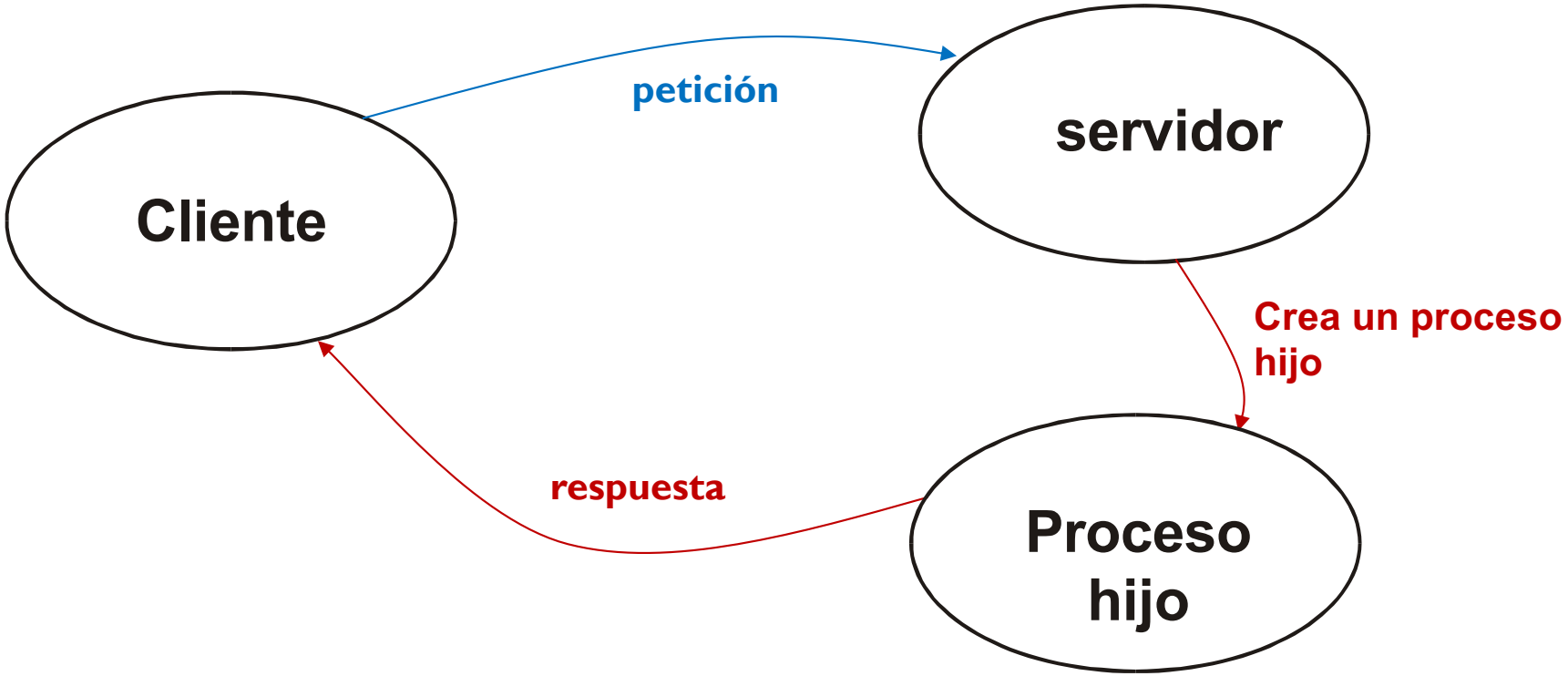


Cliente-Servidor secuencial

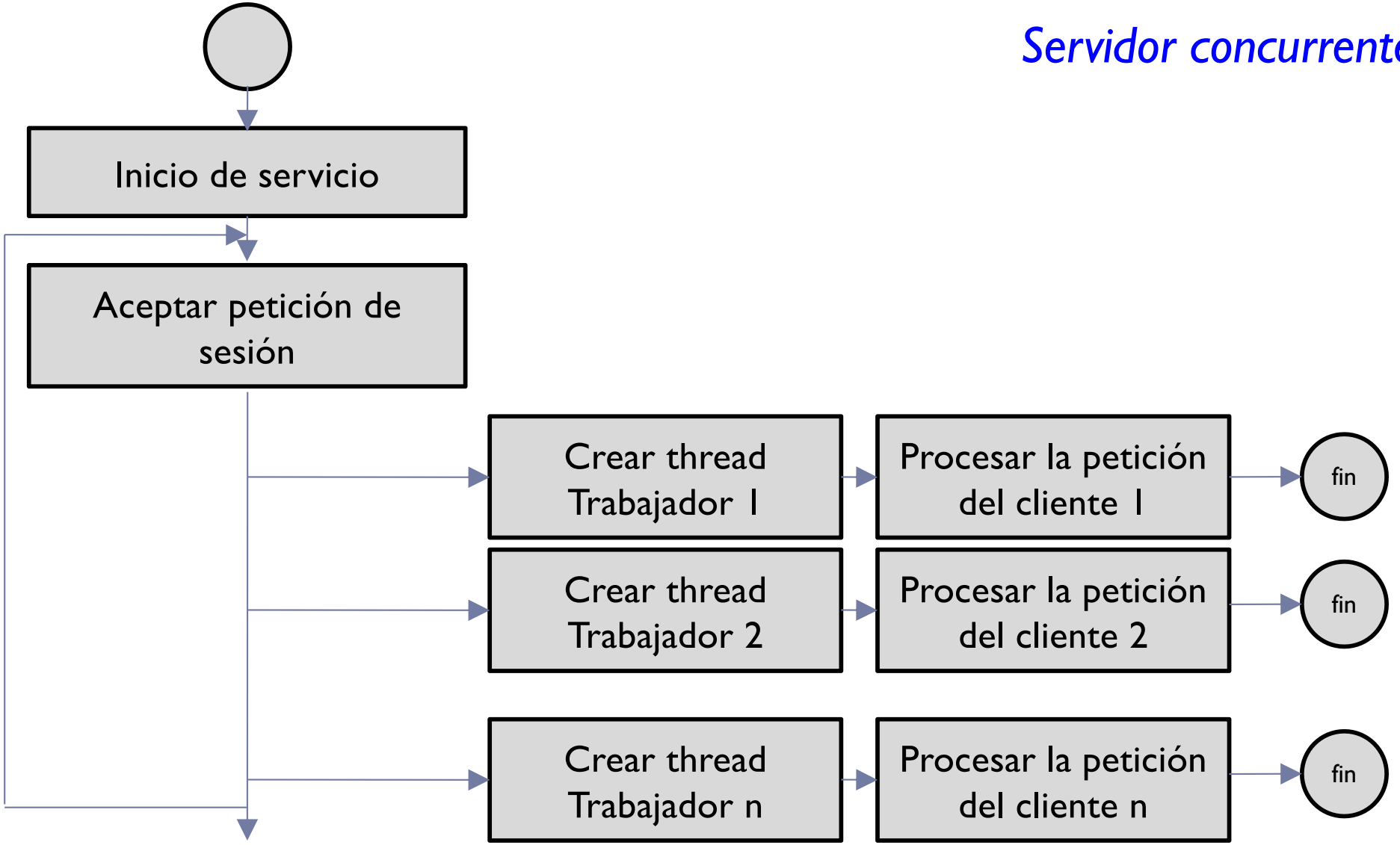


Modelo de servidor concurrente

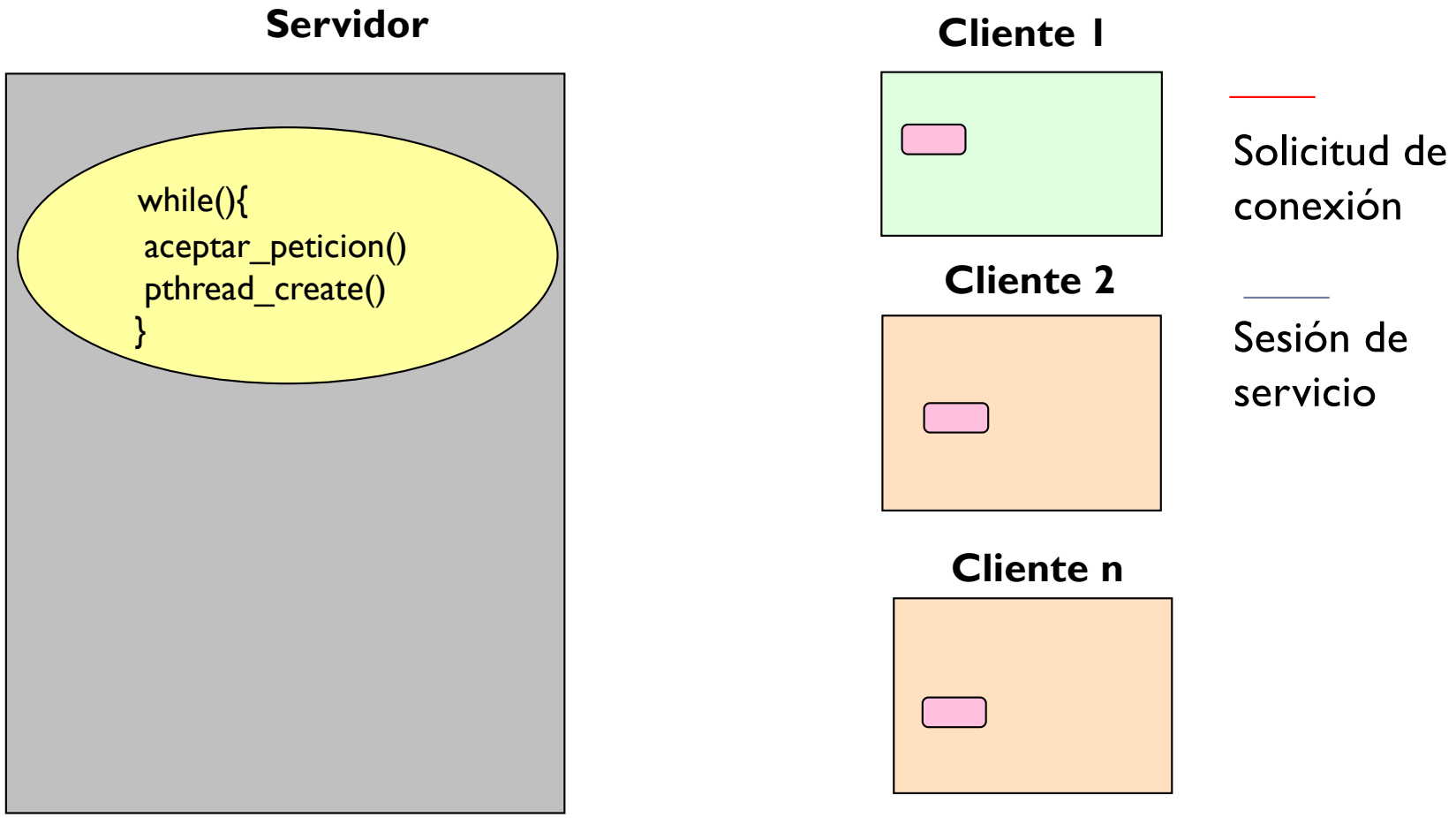
- El servidor crea un hijo que atiende la petición y envía la respuesta al cliente
- Se pueden atender múltiples peticiones de forma concurrente



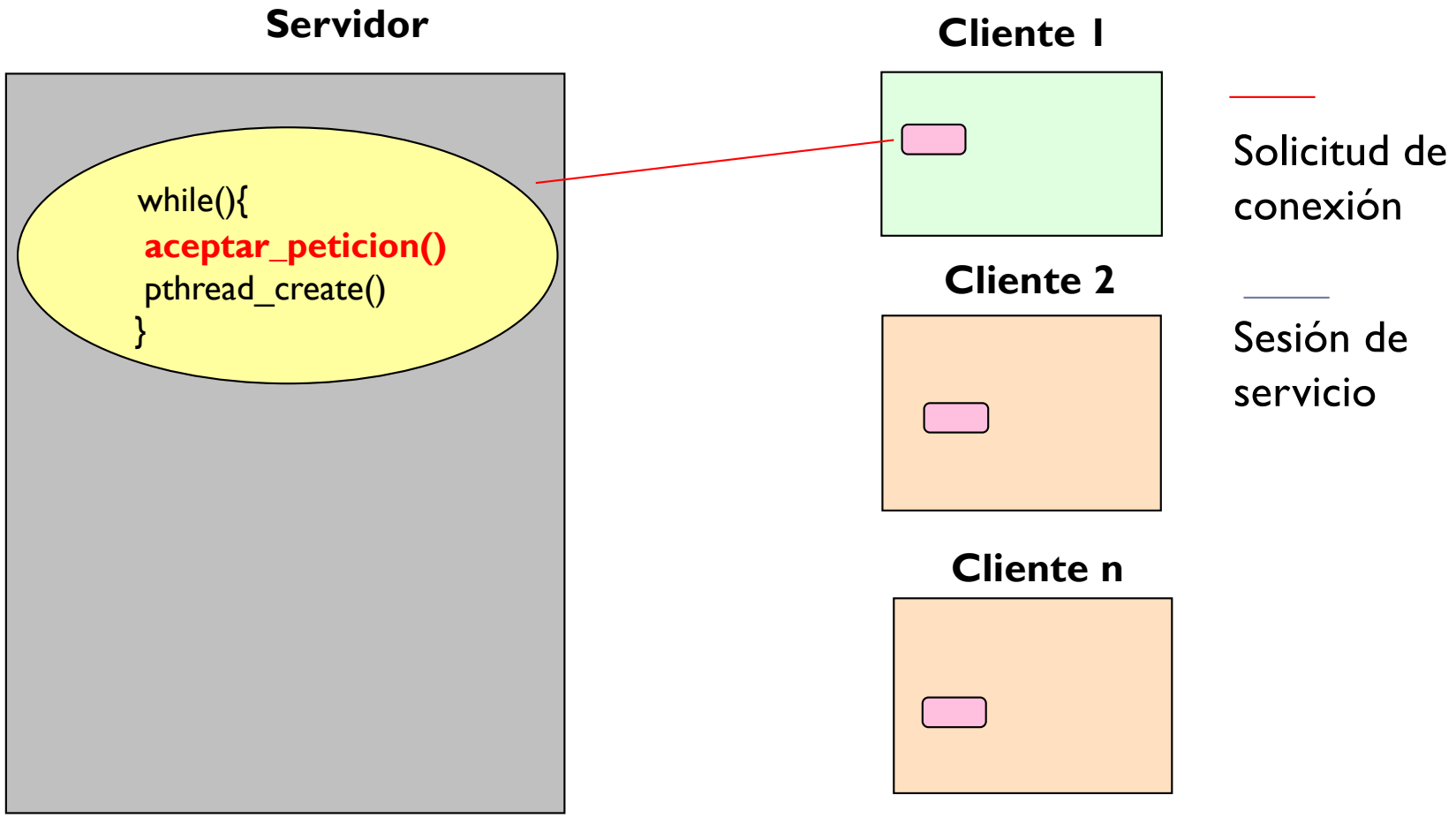
Flujo de ejecución de un servidor concurrente



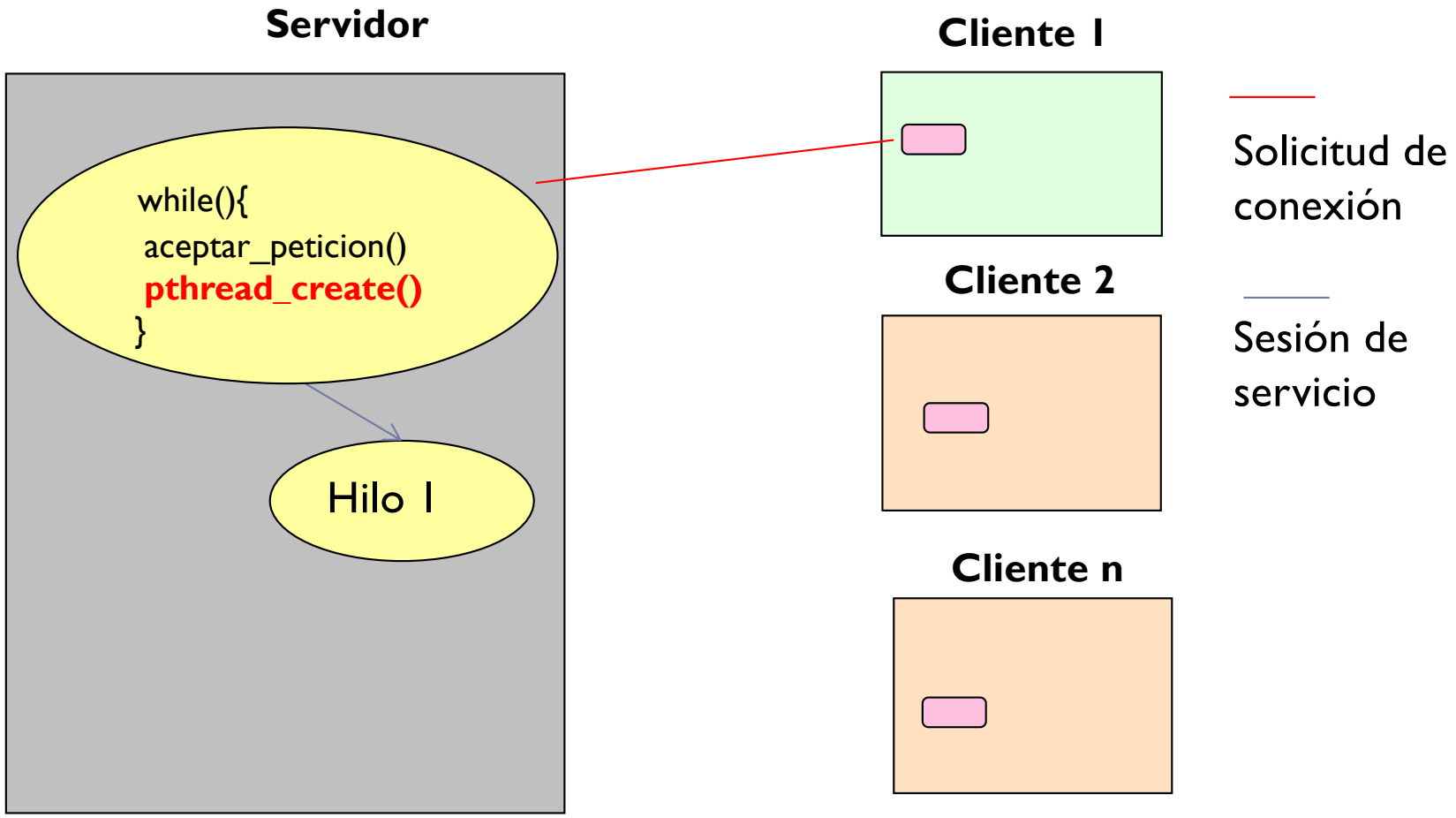
Cliente-Servidor concurrente



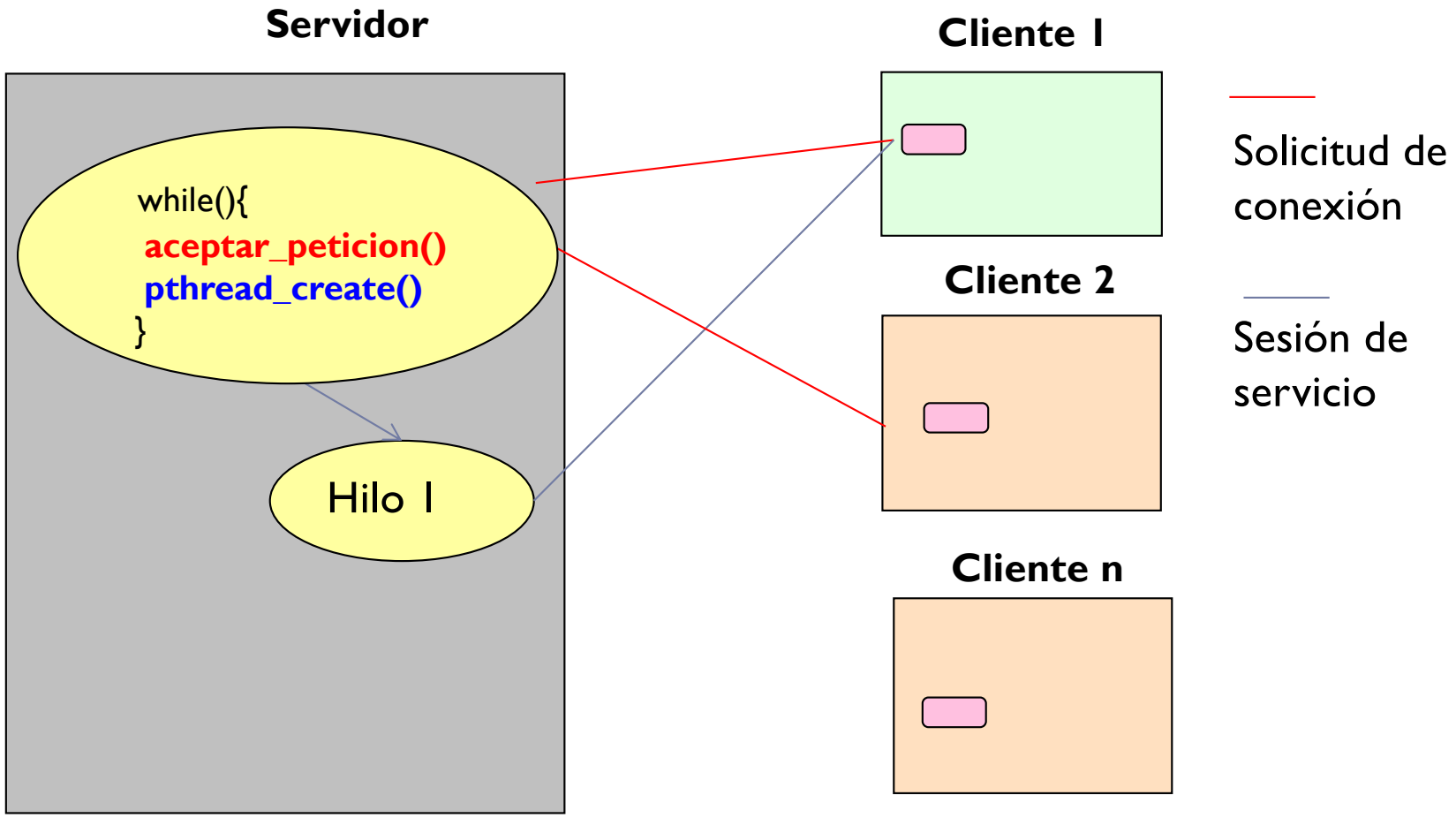
Cliente-Servidor concurrente



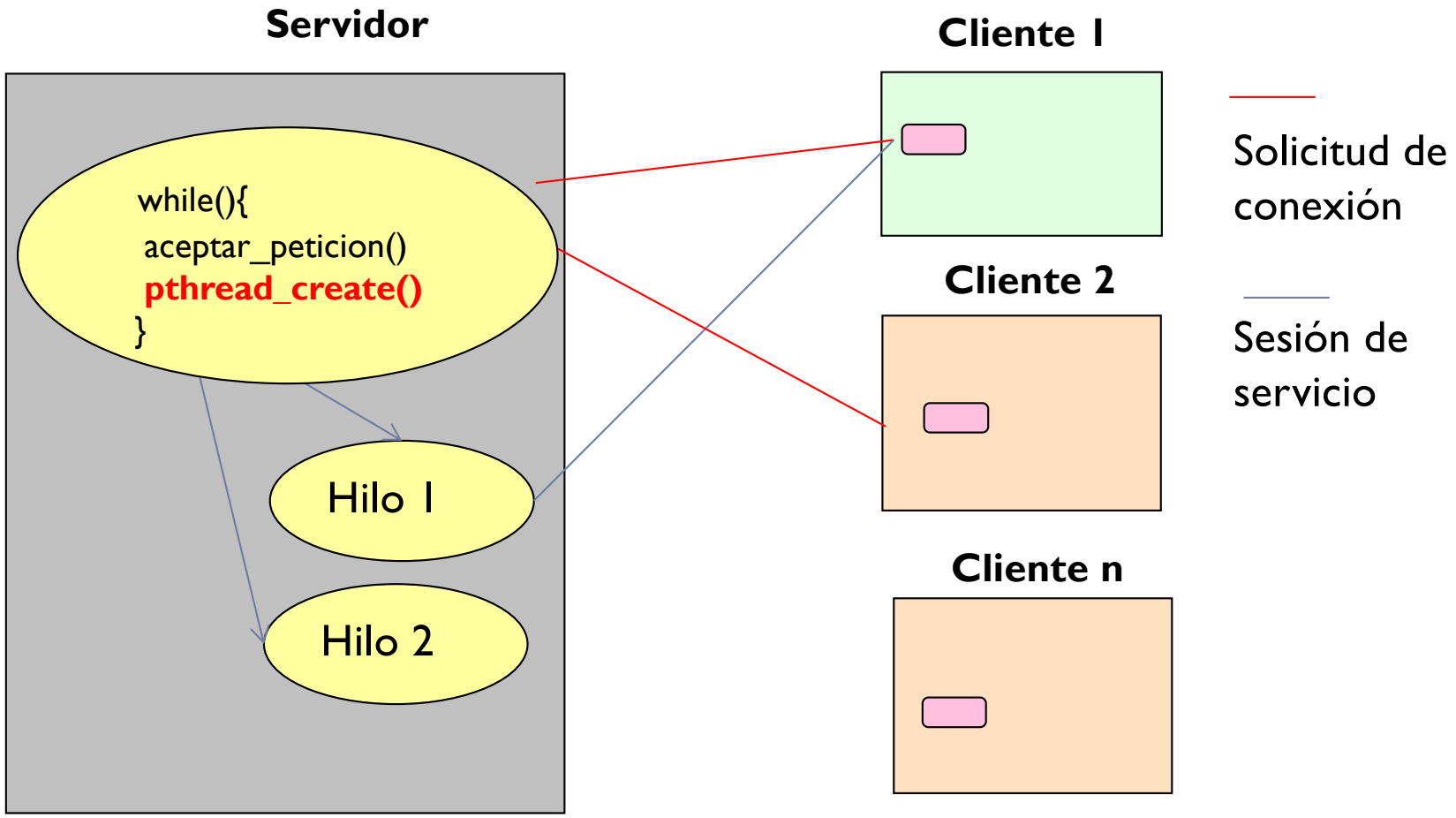
Cliente-Servidor concurrente



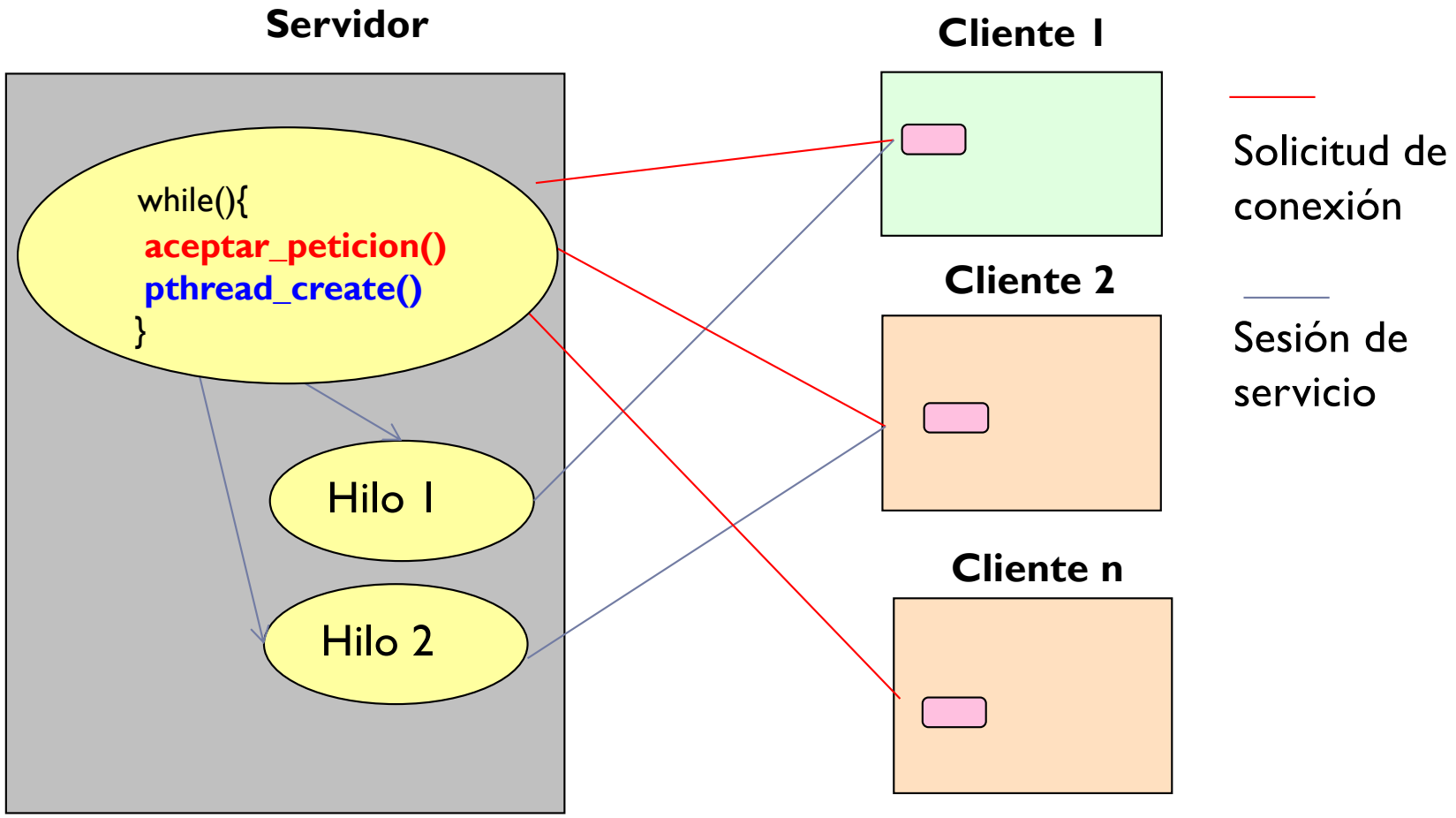
Cliente-Servidor concurrente



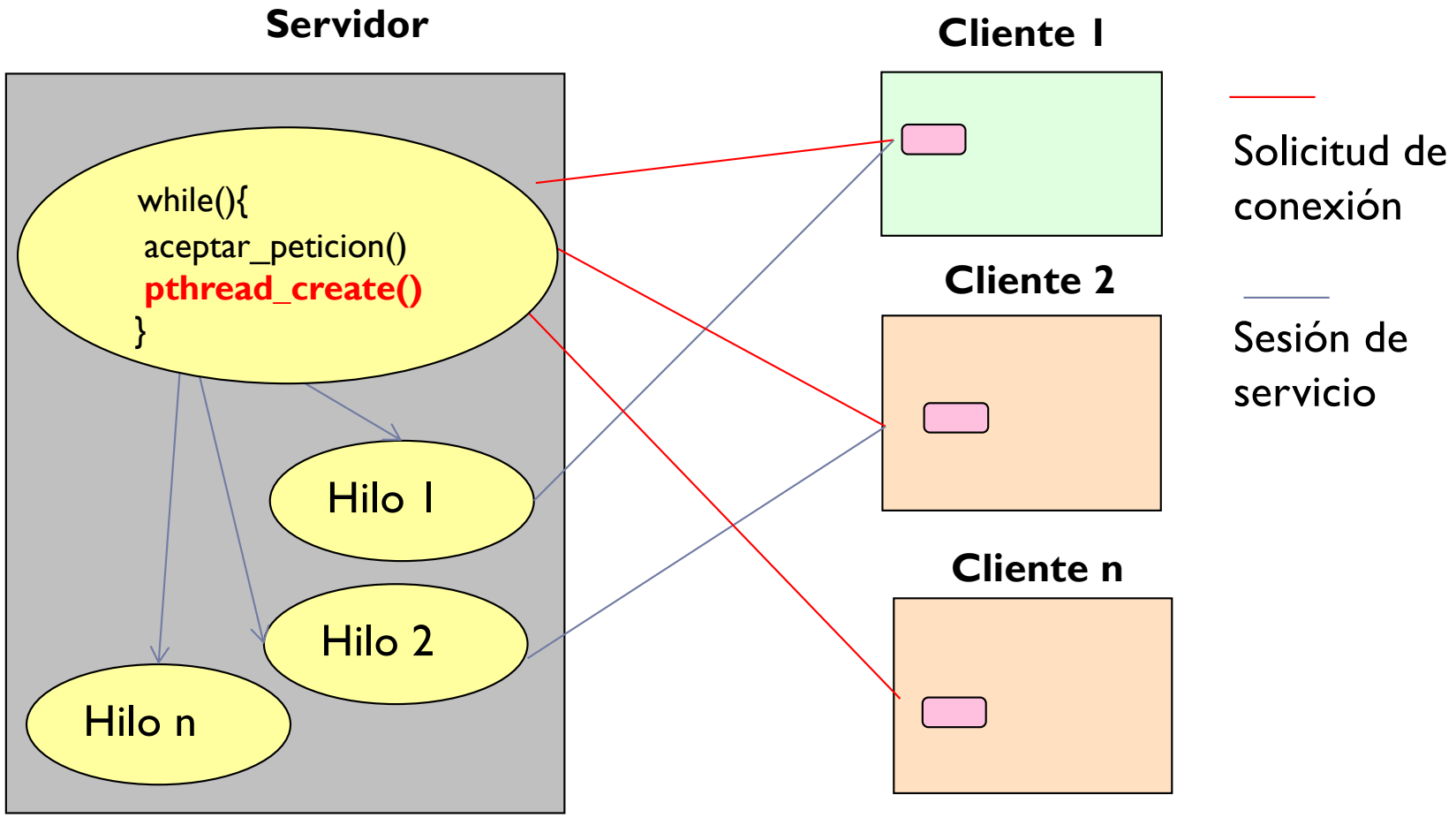
Cliente-Servidor concurrente



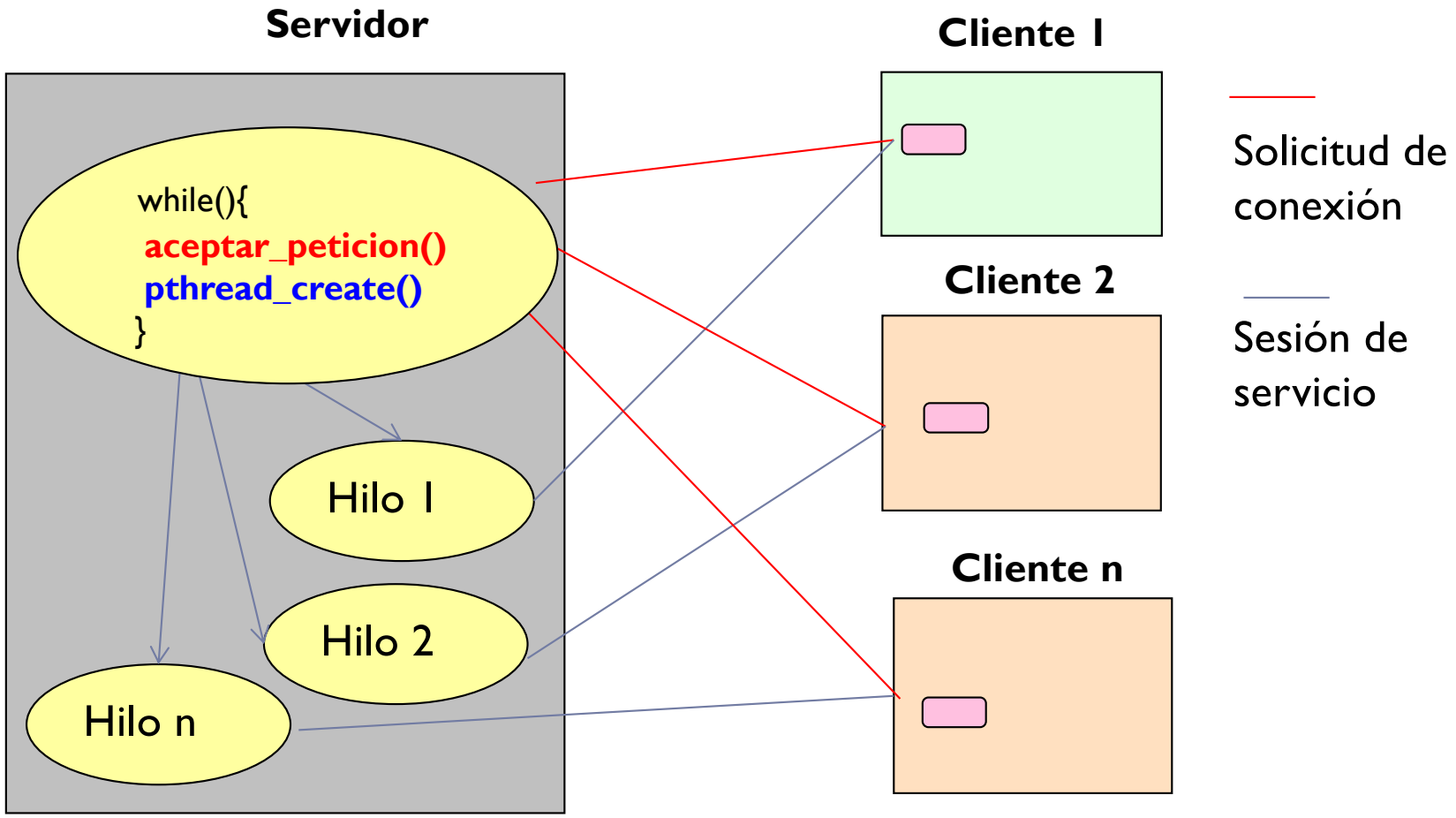
Cliente-Servidor concurrente



Cliente-Servidor concurrente



Cliente-Servidor concurrente



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**: Cada interacción entre cliente y servidor
- Cada cliente entabla una sesión separada e independiente con el servidor
  - El cliente establece 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 establece 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

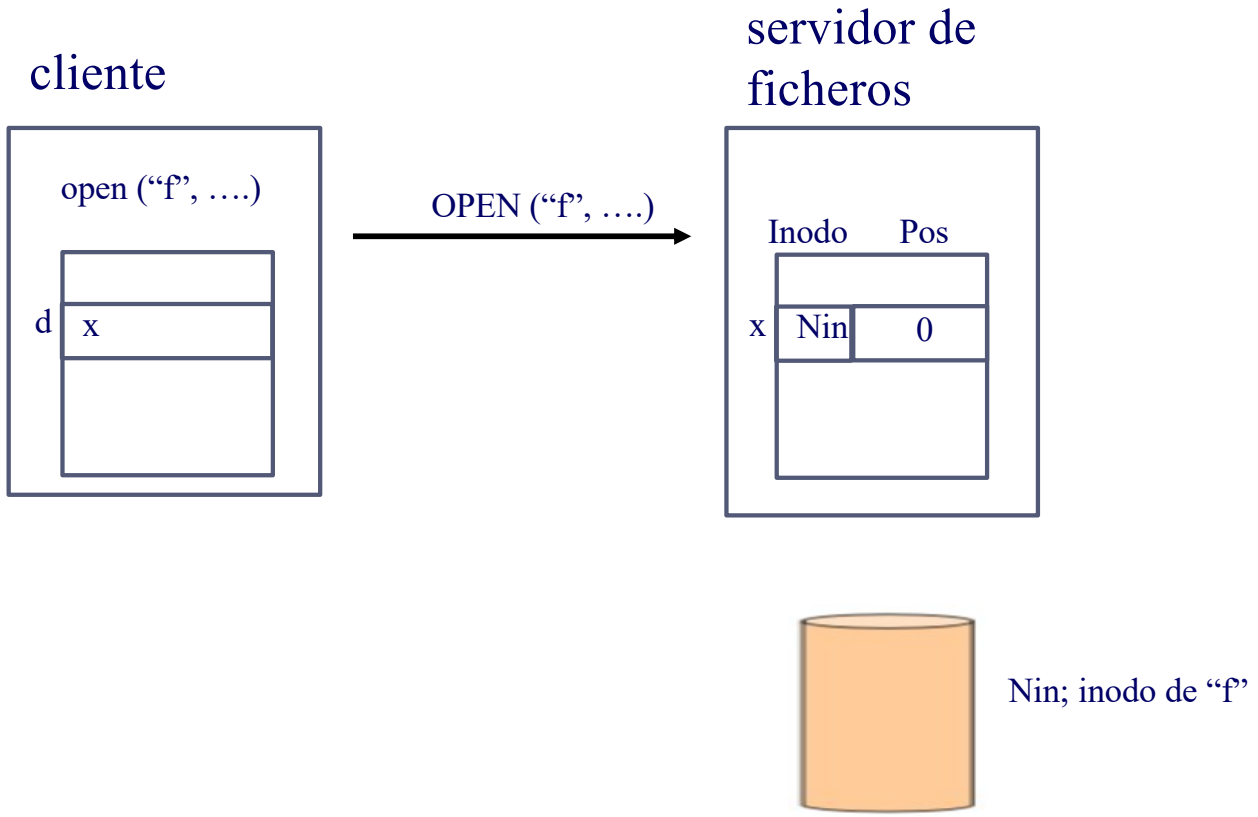
- Definición del protocolo de servicio:
  - Localización del servicio
  - Secuencia de comunicación e intercambio de mensajes entre procesos
  - Representación e interpretación de los datos

Tipos de servidores

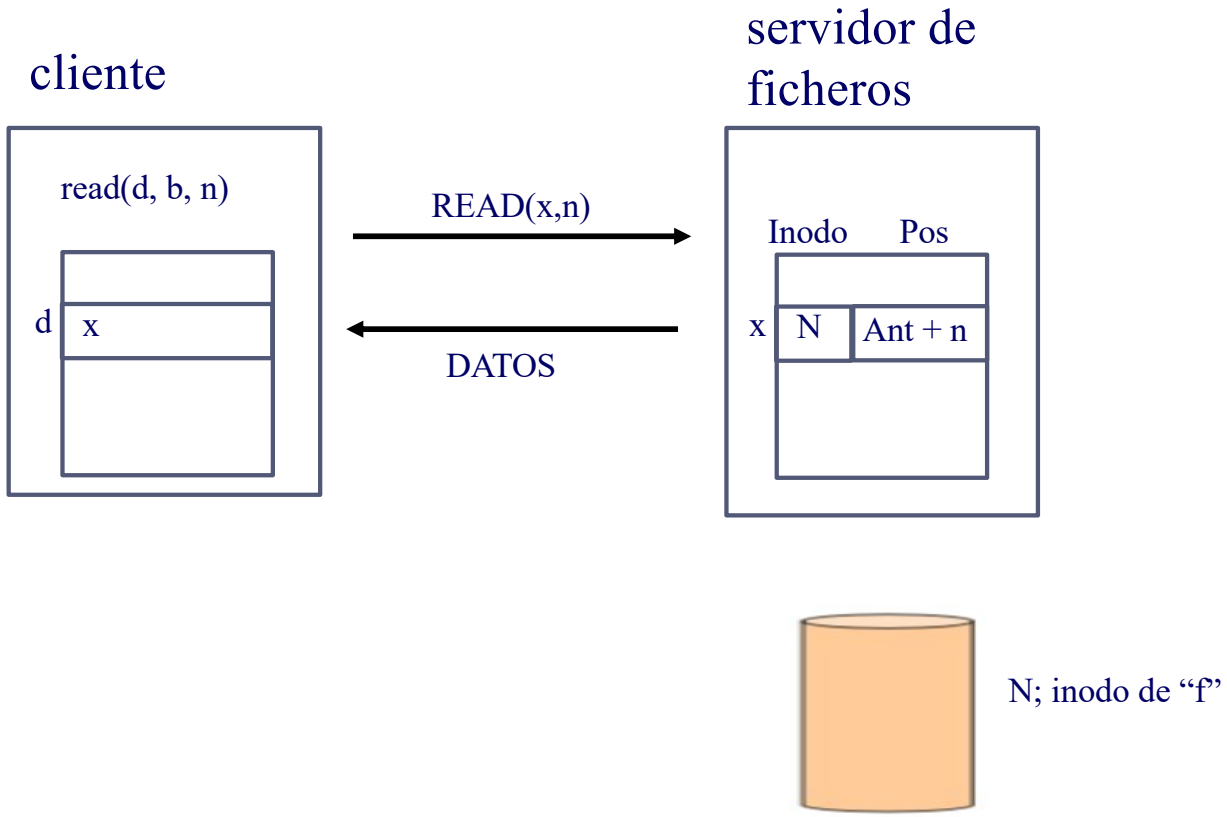
- Servidores **sin estado**:
  - Cada mensaje de petición y respuesta es independiente de las demás
  - Ejemplo: HTTP

- 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

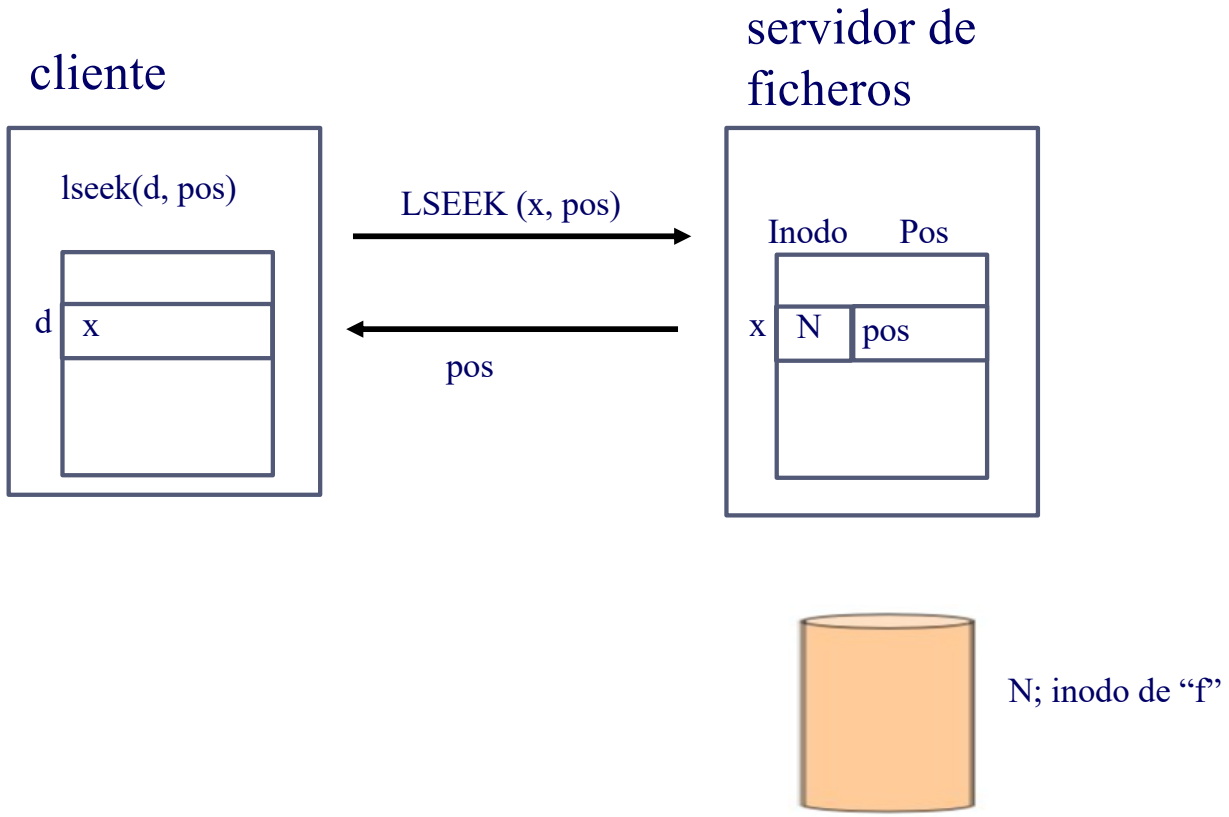
Servicio con estado
Ejemplo: servicio de ficheros



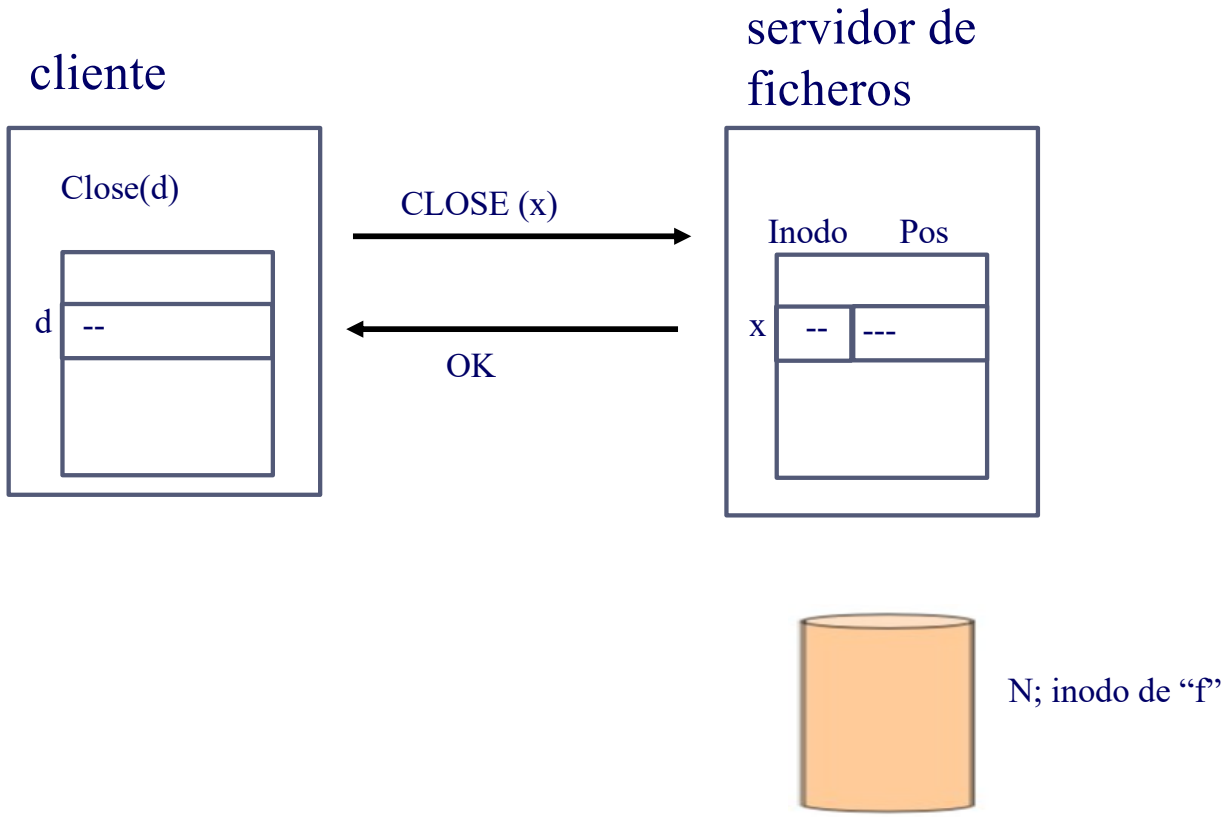
Servicio con estado
Ejemplo: servicio de ficheros



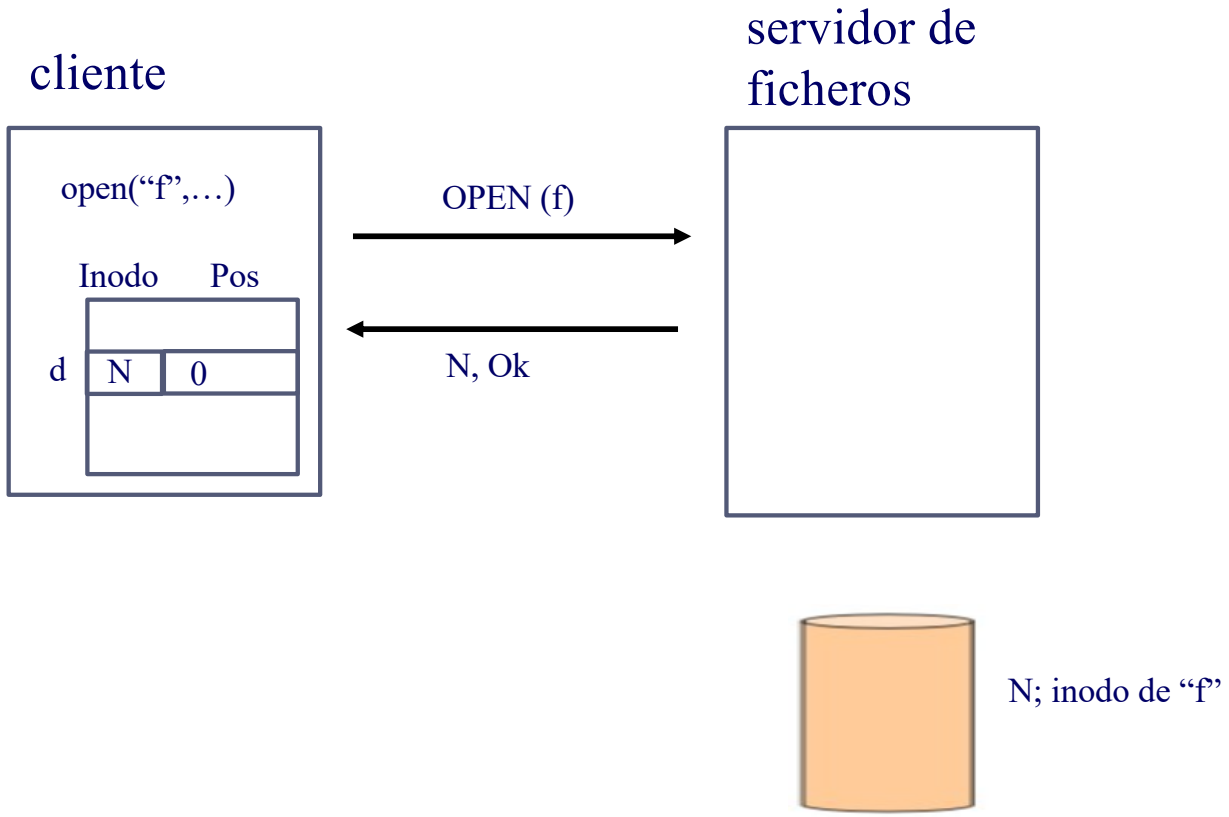
Servicio con estado
Ejemplo: servicio de ficheros



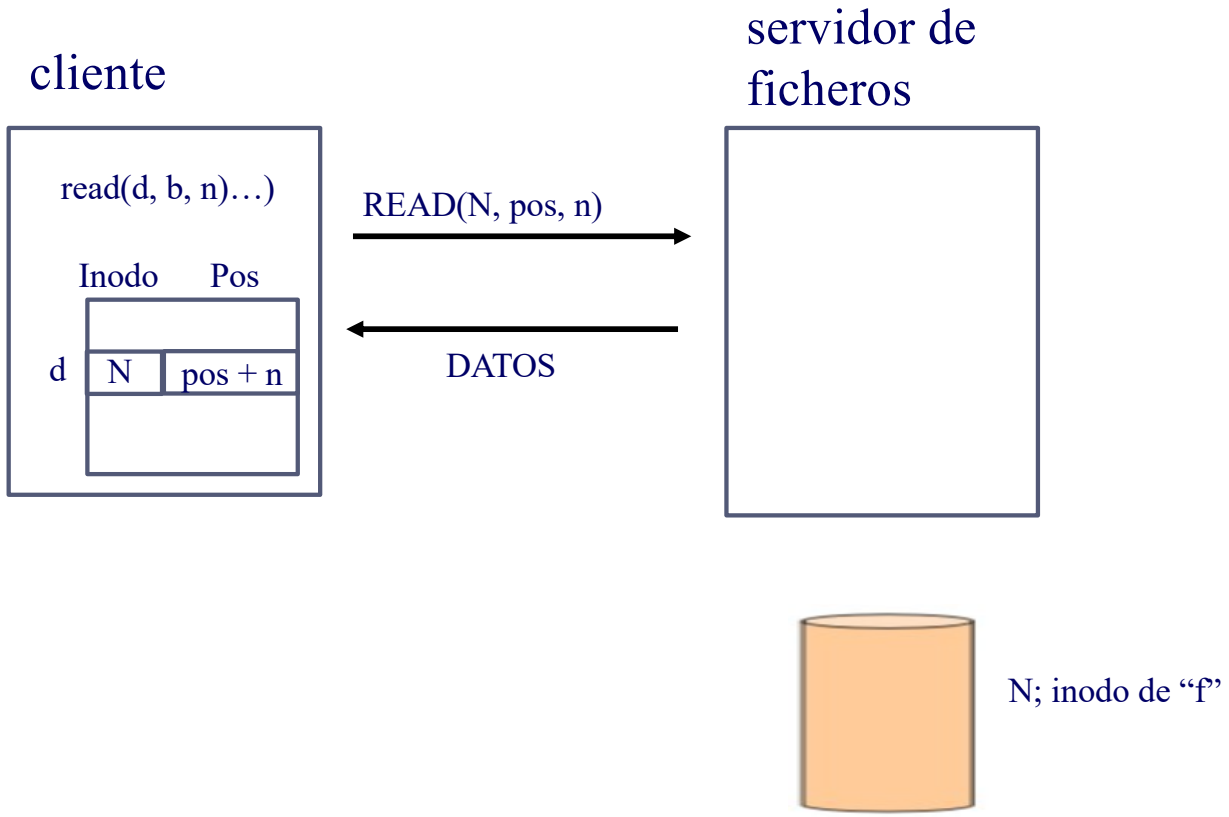
Servicio con estado
Ejemplo: servicio de ficheros



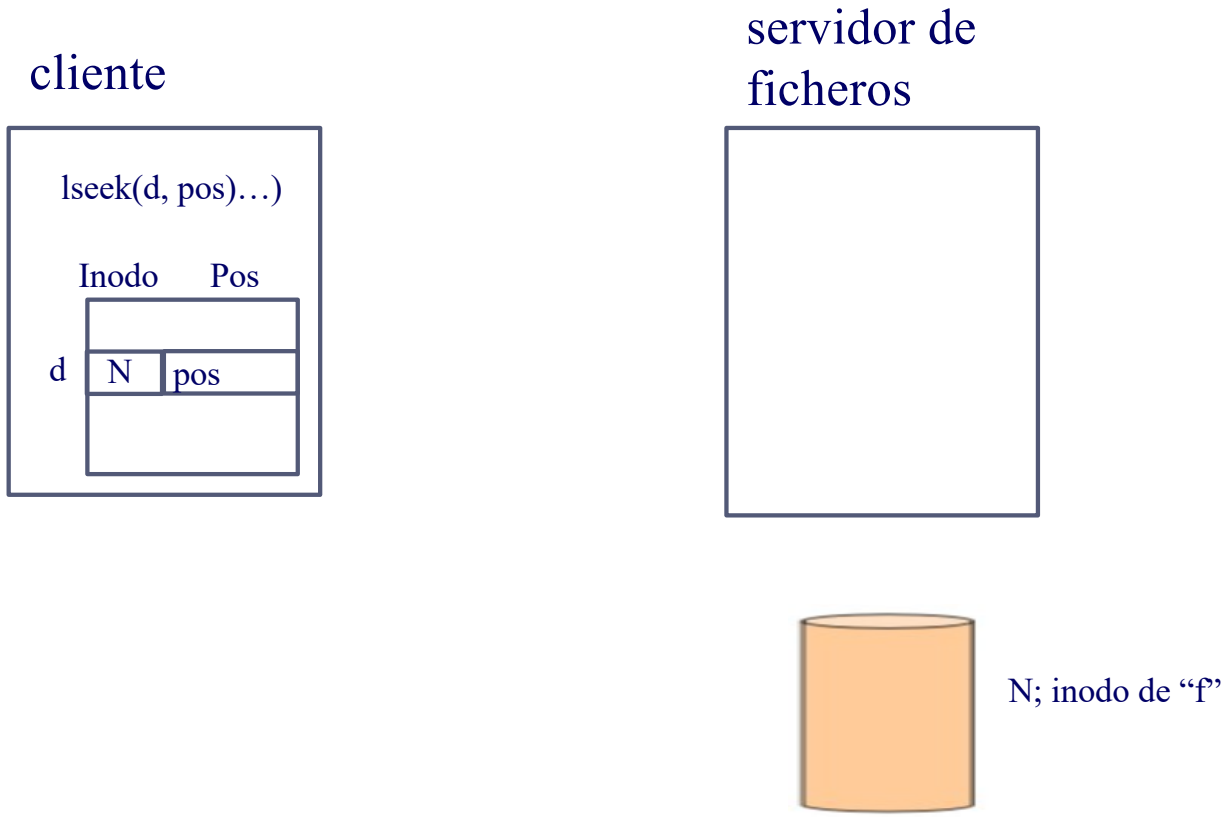
Servicio sin estado
Ejemplo: servicio de ficheros



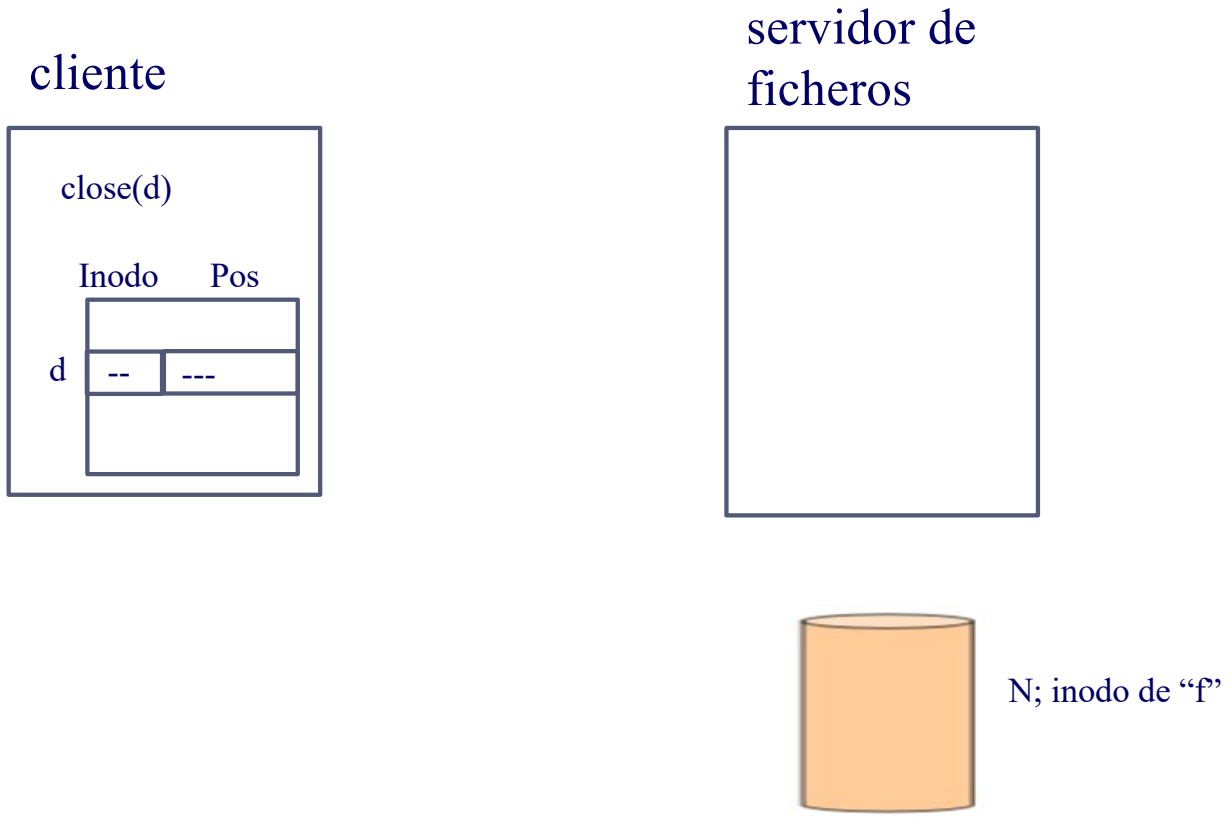
Servicio sin estado
Ejemplo: servicio de ficheros



Servicio sin estado
Ejemplo: servicio de ficheros



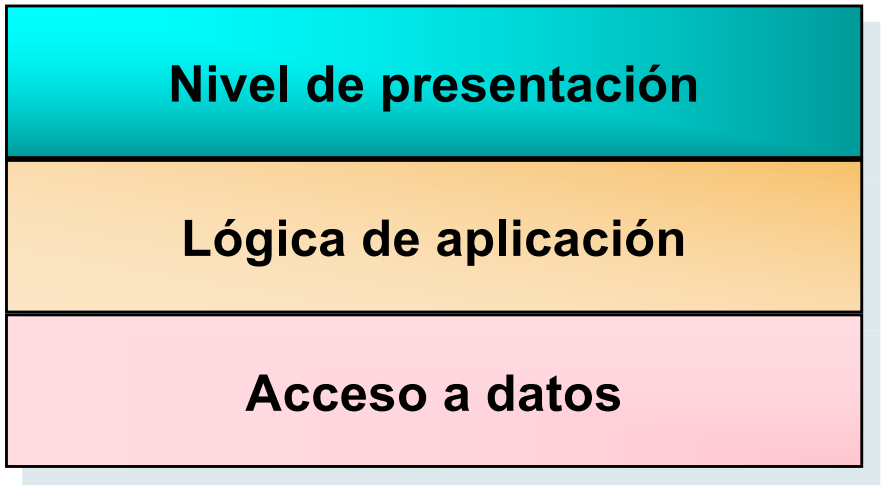
Servicio sin estado
Ejemplo: servicio de ficheros



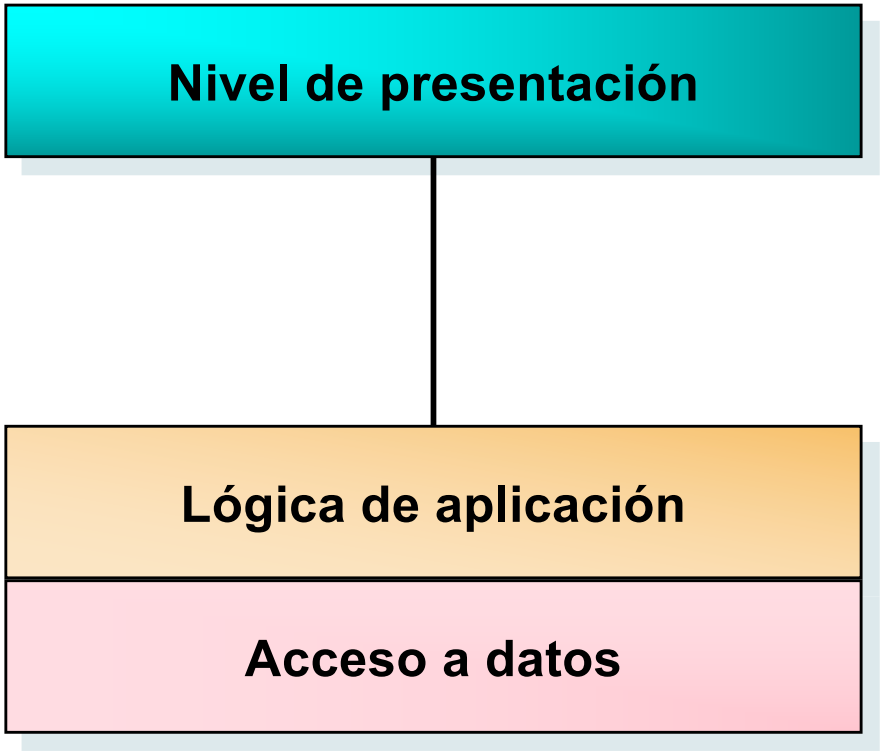
Servicio con/sin estado

- Ventaja del servicio con estado:
  - Mensajes de petición más cortos
  - Mejor rendimiento (se mantiene información en memoria)
  - Favorece la optimización del servicio: estrategias predictivas
- Ventajas del servicio sin estado:
  - Más tolerante a fallos (ante rearranque del servidor)
    - Peticiones autontenidas
  - Reduce el número de mensajes: sin comienzo ni fin de session
  - Más económico para el servidor (sin consume de memoria)
  - Facilita el reparto de carga entre servidores

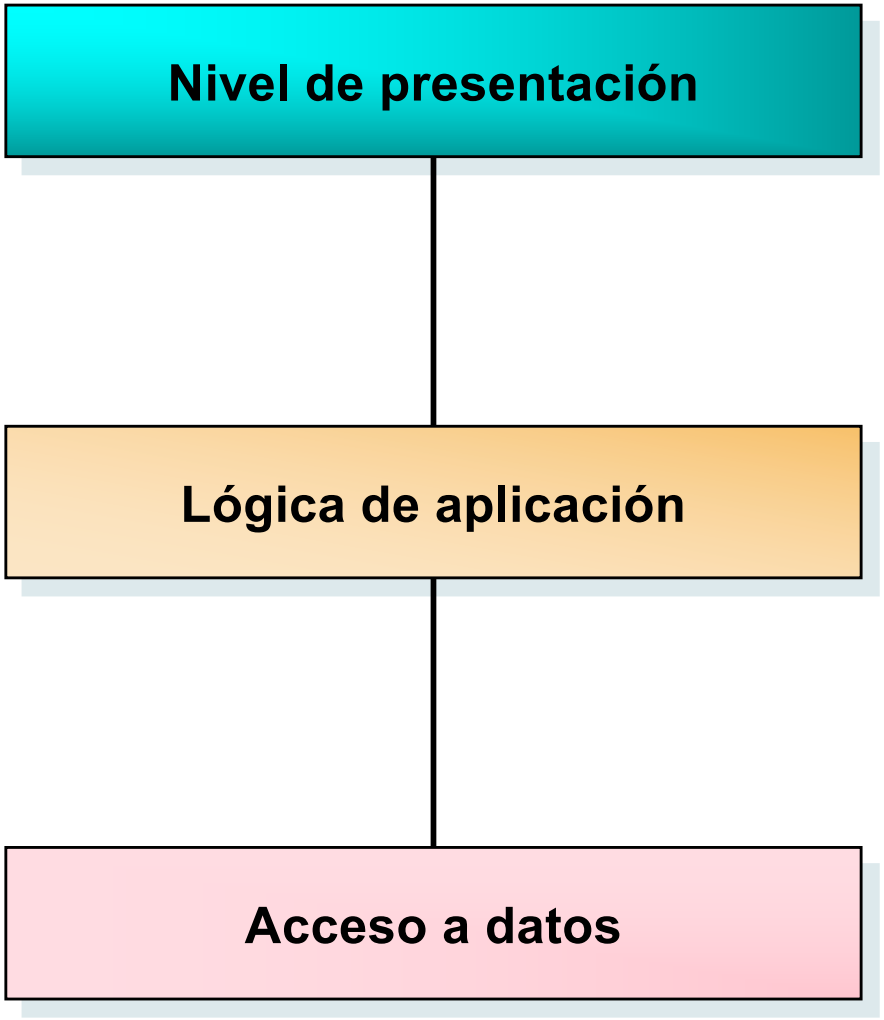
Arquitectura SW de las aplicaciones



Arquitectura two-tier



Arquitectura three-tier



¿Donde se ejecutan las tareas?

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

Responsabilidades en el cliente

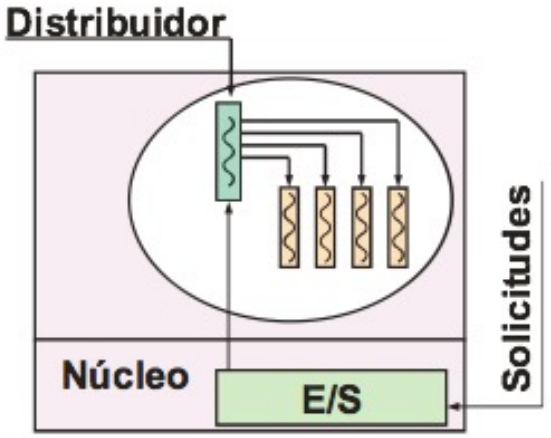
- Cliente:
  - Genera un mensaje de petición de servicio
  - Se conecta al servidor (dirección IP y puerto) [Solo orientado a conexión]
  - Envía el mensaje de petición de servicio
  - Espera por la respuesta
  - Procesa la respuesta: imprimir, almacenar, etc.
  - Desconexión [Solo orientado a conexión]

Responsabilidades en el servidor

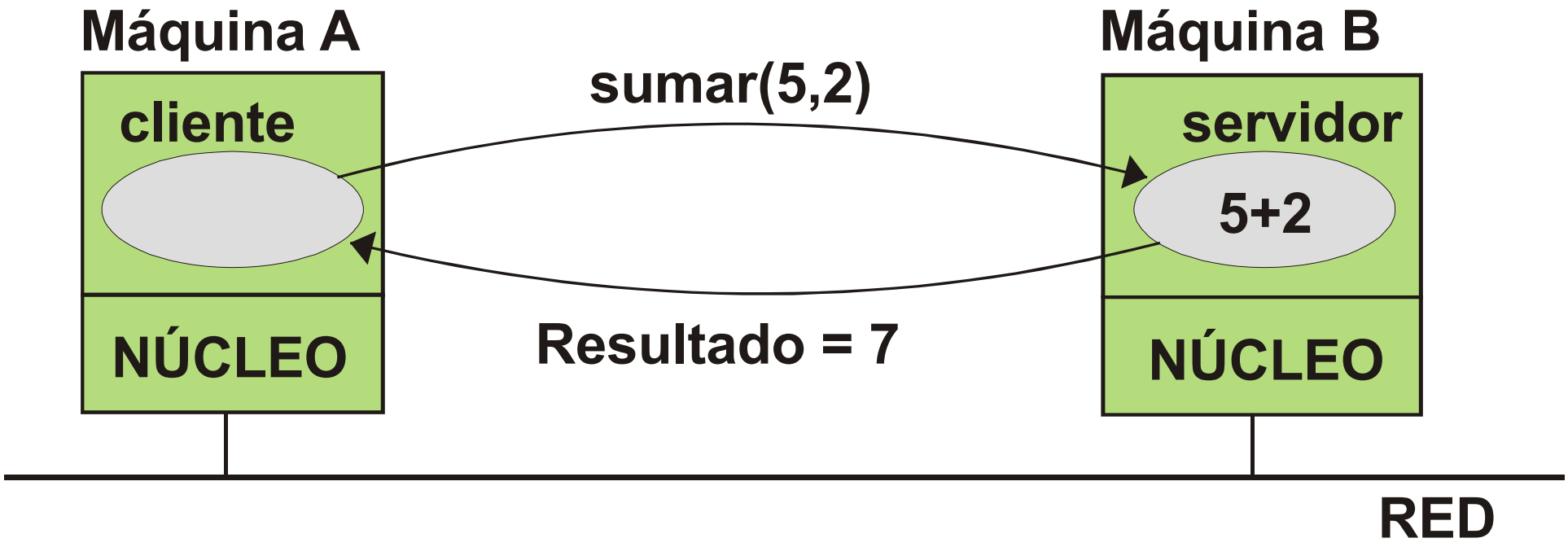
- Servidor:
  1. Espera conexiones entrantes de los clientes
     - Una conexión entrante es una petición de servicio
  2. Por cada conexión:
     - Genera un *thread* de servicio [Solo servidores concurrentes]
     - El proceso principal:
       - Vuelve a esperar por nuevas conexiones entrantes
  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

Aplicaciones cliente-servidor usando colas de mensajes

- Modelo proceso ligero distribuidor:
  - Cada petición al proceso ligero distribuidor supone la creación de un proceso ligero trabajador
  - El proceso ligero trabajador responde al proceso cliente
    - Procesa la petición
    - Envía la respuesta al servidor
  - Una vez finalizada la sesión con el cliente, el proceso ligero se destruye
- Modelo concurrente:
  - Los procesos distribuidor y trabajador ejecutan de forma concurrente



Ejemplo: Servidor de operaciones usando colas de mensajes



Ejemplo: Definición de tipos

mensaje.h

```
#define MAXSIZE         256
#define SUMA   0
#define RESTAR 1
//. . .

struct peticion  {
   int op;                      /* operación, 0 (+) 1 (-) */
   int a;                       /* operando 1 */
   int b;                       /* operando 2 */
   char q_name[MAXSIZE];        /* nombre de la cola cliente
                                   donde debe enviar la respuesta
                                   el servidor */
};
```

Ejemplo: Proceso servidor secuencial

servidor.c

```
#include "mensaje.h"
#include <mqueue.h>

int main(void) {
        mqd_t q_servidor;               /* cola de mensajes del servidor */
        mqd_t q_cliente;                /* cola de mensajes del cliente */
        struct peticion pet;      int res;
        struct mq_attr attr;

        attr.mq_maxmsg = 10;                attr.mq_msgsize = sizeof(struct peticion);
        q_servidor = mq_open("/SERVIDOR_SUMA", O_CREAT|O_RDONLY, 0700, &attr);

        while(1) {
                mq_receive(q_servidor, (char *) &pet, sizeof(pet), 0);
                if (pet.op ==0)   res = pet.a + pet.b;
                else              res = pet.a - pet.b;

                /* se responde al cliente abriendo reviamente su cola */
                q_cliente = mq_open(pet.q_name, O_WRONLY);
                mq_send(q_cliente, (const char *)&res, sizeof(int), 0);
                mq_close(q_cliente);
        }
}
```

Ejemplo: Proceso cliente

cliente.c

```
#include "mensaje.h"
#include <mqueue.h>
#include <string.h>

int main(void) {
        mqd_t q_servidor;       /* cola de mensajes del proceso servidor */
        mqd_t q_cliente;        /* cola de mensajes para el proceso cliente */
        struct peticion pet;
        int res;        struct mq_attr attr;
        attr.mq_maxmsg = 1;     attr.mq_msgsize = sizeof(int);
        q_cliente = mq_open("/CLIENTE_UNO", O_CREAT|O_RDONLY, 0700, &attr);
        q_servidor = mq_open("/SERVIDOR_SUMA", O_WRONLY);

        /* se rellena la petición */
        pet.op = 0; pet.a = 5;  pet.b = 2;   strcpy(pet.q_name, "/CLIENTE_UNO");
        mq_send(q_servidor, (const char *)&pet, sizeof(pet), 0);
        mq_receive(q_cliente, (char *) &res, sizeof(int), 0);
        printf("Resultado = %d\n", res);
        mq_close(q_servidor);
        mq_close(q_cliente);
        mq_unlink("/CLIENTE_UNO");
        return 0;
}
```

Ejemplo: Proceso cliente

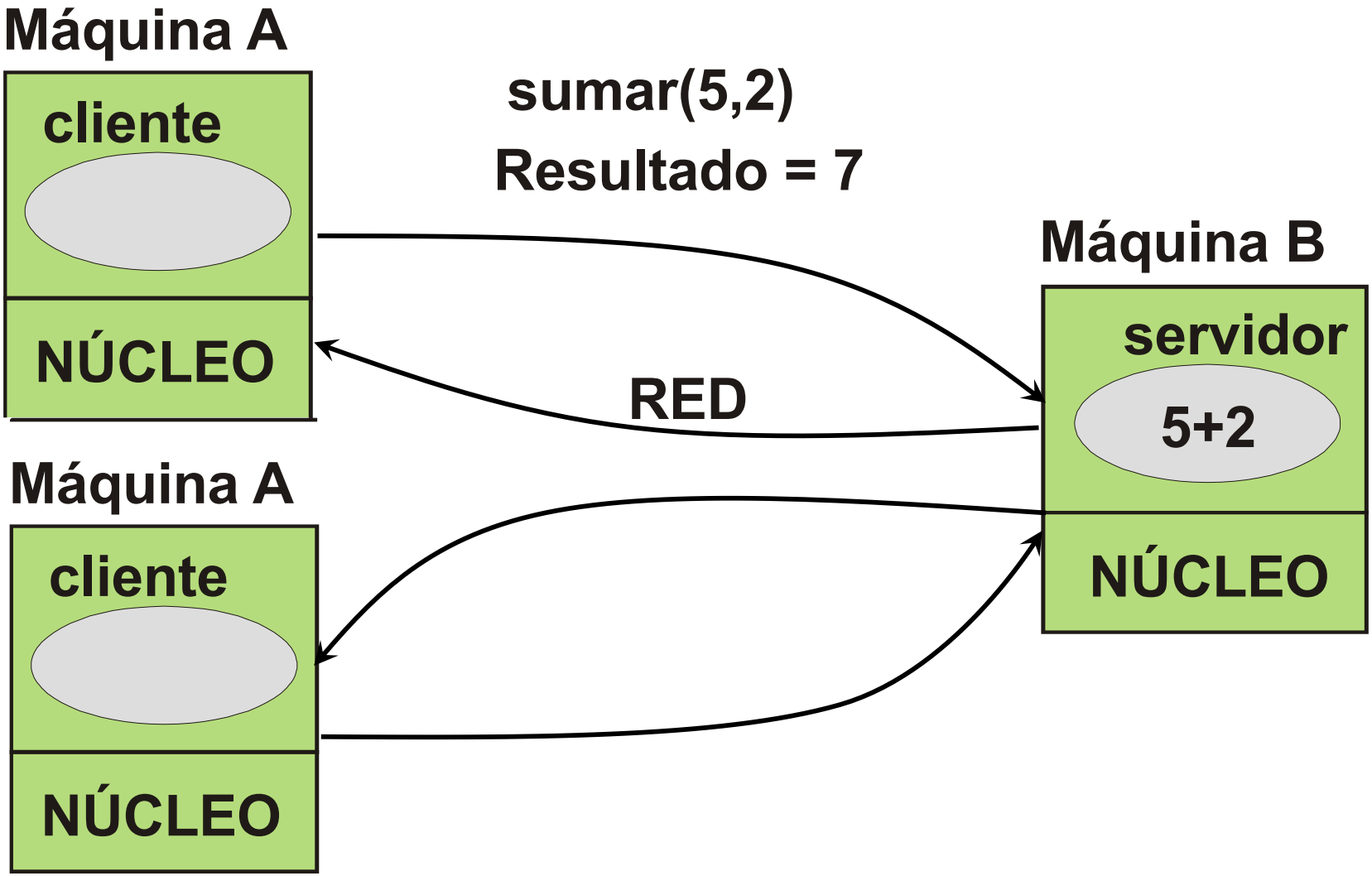
```
#include "mensaje.h"
#include <mqueue.h>
#include <string.h>

void main(void) {
        mqd_t q_servidor;       /* cola de mensajes del proceso servidor */
        mqd_t q_cliente;        /* cola de mensajes para el proceso cliente */
        struct peticion pet;
        int res;        struct mq_attr attr;
        attr.mq_maxmsg = 1;     attr.mq_msgsize = sizeof(int);
        q_cliente = mq_open("/CLIENTE_UNO", O_CREAT|O_RDONLY, 0700, &attr);
        q_servidor = mq_open("/SERVIDOR_SUMA", O_WRONLY);

        /* se rellena la petición */
        pet.op = 0; pet.a = 5;  pet.b = 2;   strcpy(pet.q_name, "/CLIENTE_UNO");
        mq_send(q_servidor, (const char *)&pet, sizeof(pet), 0);
        mq_receive(q_cliente, (char *) &res, sizeof(int), 0);
        printf("Resultado = %d\n", res);
        mq_close(q_servidor);
        mq_close(q_cliente);
        mq_unlink("/CLIENTE_UNO");
}
```

Problema:
Nombre único para todos los clientes

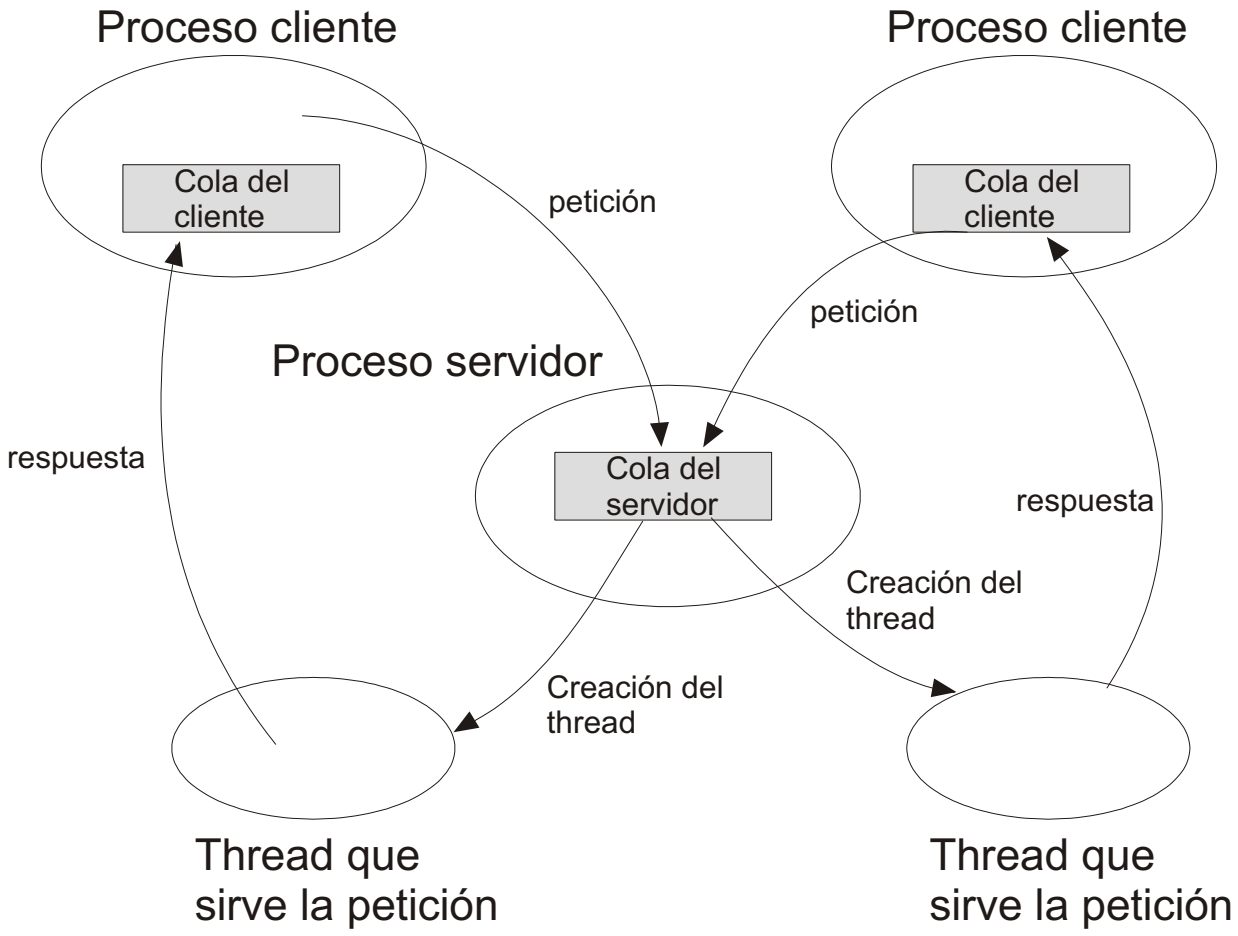
Ejemplo: Proceso cliente
Solución al nombre único de cola

```
#include "mensaje.h"
#include <mqueue.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

void main(void) {
        mqd_t q_servidor;       /* cola de mensajes del proceso servidor */
        mqd_t q_cliente;        /* cola de mensajes para el proceso cliente */
        struct peticion pet;
        char queuename[MAXSIZE];

        int res;        struct mq_attr attr;
        attr.mq_maxmsg = 1;     attr.mq_msgsize = sizeof(int);
        sprintf(queuename,  »/Cola-%d", getpid());
        q_cliente = mq_open(queuename, O_CREAT|O_RDONLY, 0700, &attr);
                    .
                    .
       mq_unlink(queuename);
}
```

Ejemplo: Servidor concurrente



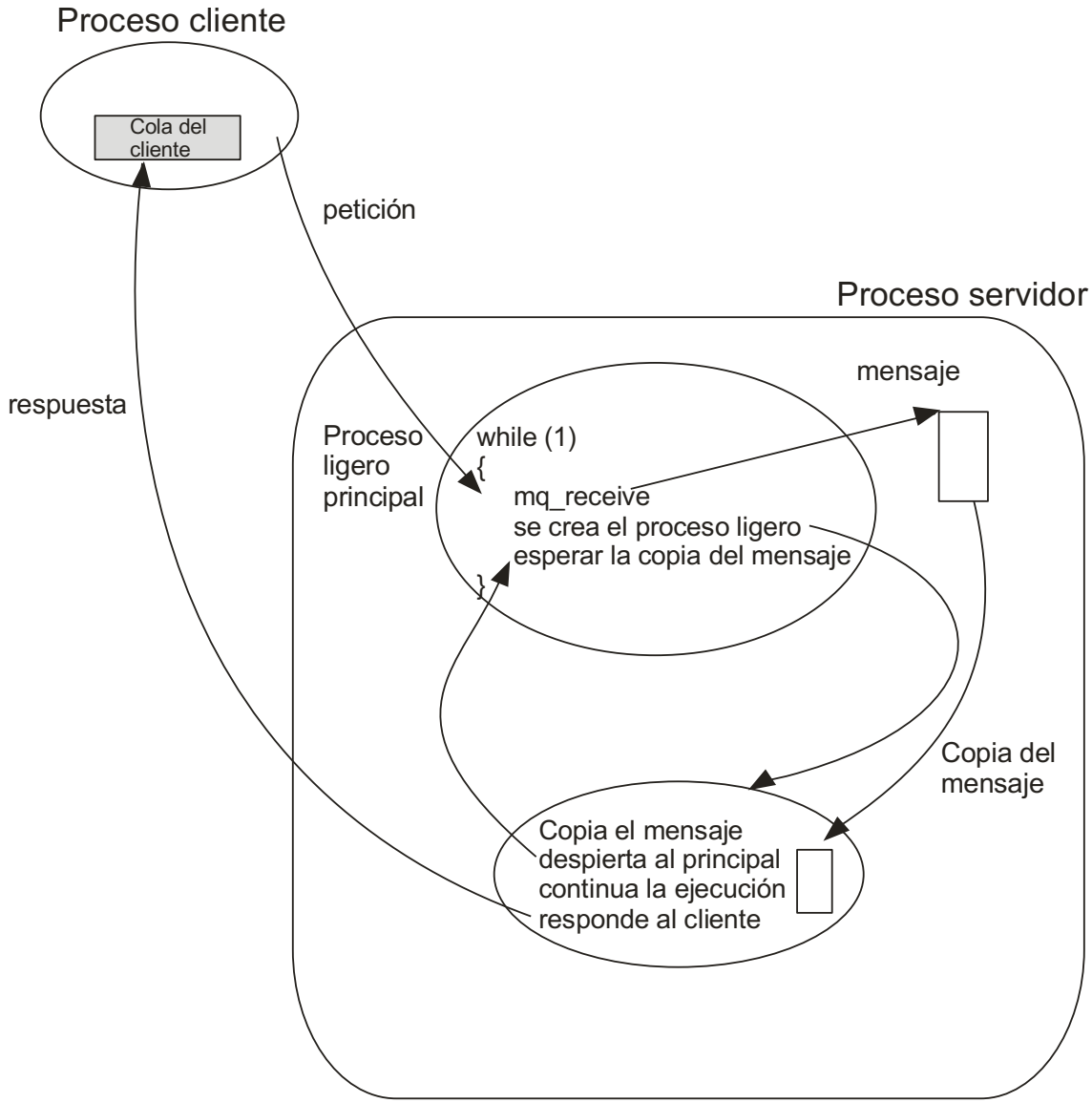
Estructura de un servidor multihread



Creación de threads

- Bajo demanda
  - Se crea un thread cada vez que se recibe una petición
- Pool de threads
  - Se tiene un número fijo de threads creados.
  - Cada vez que se recibe una petición se busca un hilo libre ya creado para que atienda la petición.
  - Uso de una cola de peticiones (modelo productor/consumidor)

Cliente-servidor con colas de mensajes
Creación de threads bajo demanda



Servidor multithread con colas de mensajes (threads bajo demanda) (I)

servidor.c

```
#include "mensaje.h"
#include <mqueue.h>
#include <pthread.h>
#include <stdio.h>
#include <stdbool.h>
#include <string.h>

/* mutex y variables condicionales para proteger la copia del mensaje*/
pthread_mutex_t mutex_mensaje;
int mensaje_no_copiado = true;
pthread_cond_t cond_mensaje;

int main(void) {
   mqd_t q_servidor;                    /* cola del servidor */
   struct peticion mess;                /* mensaje a recibir */
   struct mq_attr q_attr;               /* atributos de la cola */
   pthread_attr_t t_attr;               /* atributos de los threads */
   pthread_t thid;

   q_attr.mq_maxmsg = 10;
   q_attr.mq_msgsize = sizeof(struct peticion);
```

Servidor multithread con colas de mensajes (threads bajo demanda) (II)

servidor.c

```
 q_servidor = mq_open(”/SERVIDOR_SUMA", O_CREAT|O_RDONLY,  0700, &q_attr);
 if (q_servidor == -1) {
      perror(”No se puede crear la cola de servidor”);
      return 1;
}

pthread_mutex_init(&mutex_mensaje, NULL);
pthread_cond_init(&cond_mensaje, NULL);
pthread_attr_init(&t_attr);

/* atributos de los threads, threads independientes */
pthread_attr_setdetachstate(&t_attr, PTHREAD_CREATE_DETACHED);

…..
```

Servidor multithread con colas de mensajes (threads bajo demanda) (III)

```
while (TRUE) {
        mq_receive(q_servidor, (char *) &mess,
                                sizeof(struct peticion), 0);

        pthread_create(&thid, &t_attr, tratar_mensaje, (void *)&mess);









    }
}
```

Servidor multithread con colas de mensajes (threads bajo demanda) (IV)

```
while (true) {
        mq_receive(q_servidor, (char *) &mess,
                                sizeof(struct peticion), 0);

        pthread_create(&thid, &t_attr, tratar_mensaje, (void *)&mess);
    }
}
```

Condición de carrera

Servidor multithread con colas de mensajes (threads bajo demanda) (V)

servidor.c

```
while (true) {
  mq_receive(q_servidor, (char *) &mess,
                              sizeof(struct peticion), 0);
  if (pthread_create(&thid, &t_attr, tratar_mensaje, (void *)&mess)
                                                    == 0) {
```

Sección crítica

```
    /* se espera a que el thread copie el mensaje */
    pthread_mutex_lock(&mutex_mensaje);
    while (mensaje_no_copiado)
            pthread_cond_wait(&cond_mensaje, &mutex_mensaje);
    mensaje_no_copiado = true;
    pthread_mutex_unlock(&mutex_mensaje);
  }     /*IF */
 }  /* FIN while */
} /* Fin main */
```

Servidor multithread con colas de mensajes (threads bajo demanda)(VI)

servidor.c

```
void tratar_mensaje(void  *mess){
  struct peticion mensaje;    /* mensaje local */
  mqd_t q_cliente;            /* cola del cliente */
  int resultado;              /* resultado de la operación */

  /* el thread copia el mensaje a un mensaje local */
  pthread_mutex_lock(&mutex_mensaje);

  mensaje = (*(struct peticion *) mess);

  /* ya se puede despertar al servidor*/
  mensaje_no_copiado = false;

  pthread_cond_signal(&cond_mensaje);

  pthread_mutex_unlock(&mutex_mensaje);
```

Servidor multithread con colas de mensajes (threads bajo demanda)(VII)

servidor.c

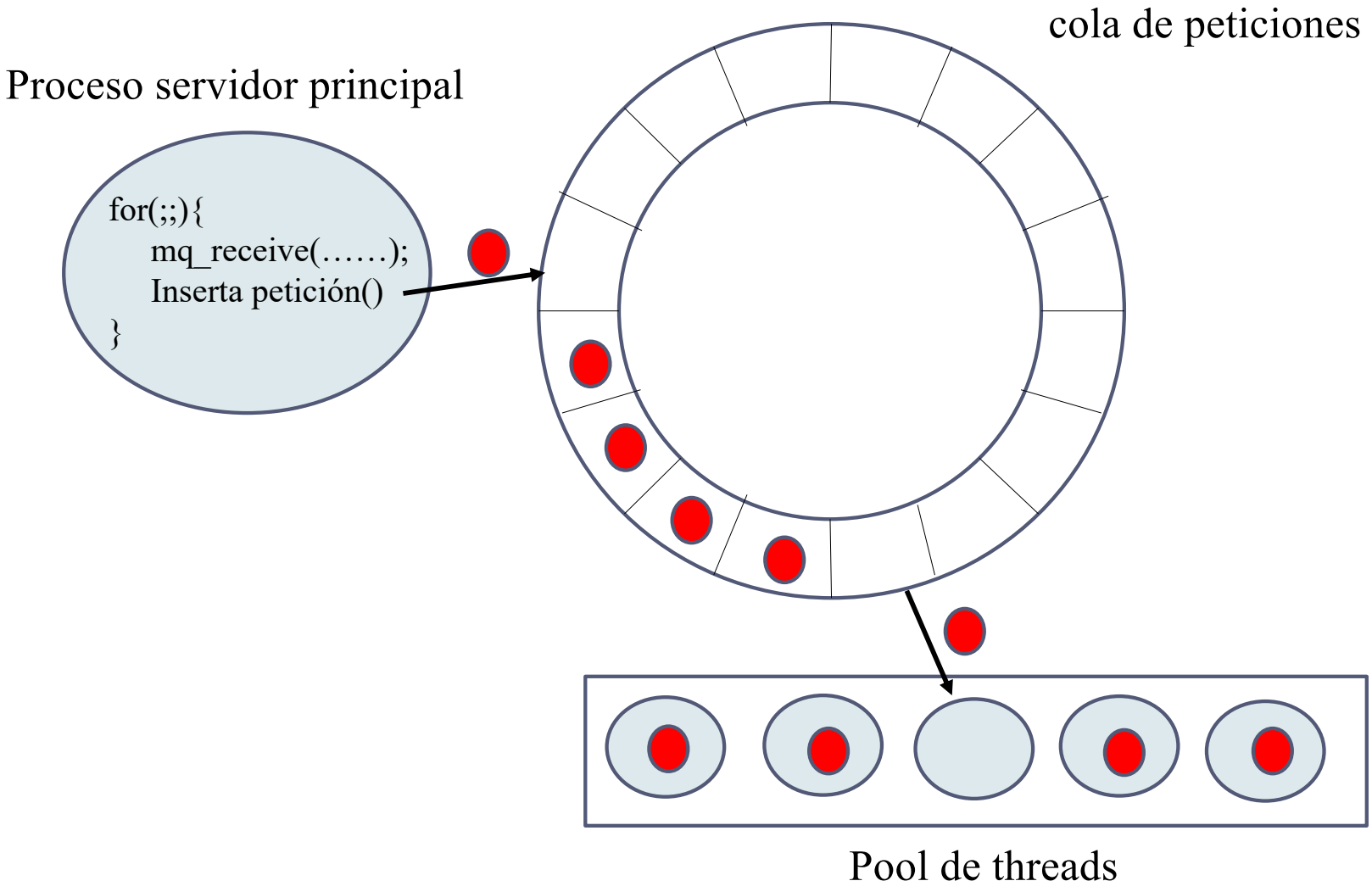
```
  /* ejecutar la petición del cliente y preparar respuesta */
  if (mensaje.op ==0)
        resultado = mensaje.a + mensaje.b;
  else
        resultado = mensaje.a - mensaje.b;
  /* Se devuelve el resultado al cliente */
  /* Para ello se envía el resultado a su cola */
  q_cliente = mq_open(mensaje.q_name, O_WRONLY);
  if (q_cliente == -1)
        perror("No se puede abrir la cola del cliente");
  else {
        mq_send(q_cliente, (const char *) &resultado, sizeof(int), 0);
        mq_close(q_cliente);
  }
  pthread_exit(0);
}
```

Proceso cliente

cliente.c

```
#include "mensaje.h"
#include <mqueue.h>
#include <pthread.h>
#include <stdio.h>

void main(void) {
   mqd_t q_servidor;  /* cola de mensajes del proceso servidor */
   mqd_t q_cliente;              /* cola de mensajes para el proceso cliente */
   struct peticion pet;     int res;       struct mq_attr attr;
   char queuename[MAXSIZE];

   attr.mq_maxmsg = 1;
   attr.mq_msgsize = sizeof(int);
   sprintf(queuename,  »/Cola-%d", getpid());
   q_cliente = mq_open(queuename, O_CREAT|O_RDONLY, 0700, &attr);

   q_servidor = mq_open(”/SERVIDOR_SUMA", O_WRONLY);

   /* se rellena la petición */
   pet.opm = 0;       pet.a = 5; pet.b = 2;    strcpy(pet.q_name, queuename);

   mq_send(q_servidor, (const char *) &pet, sizeof(struct petiticion), 0);
   mq_receive(q_cliente, (char*) &res, sizeof(int), 0);

   mq_close(q_servidor);
   mq_close(q_cliente);
   mq_unlink(queuename);
}
```

Servidor multithread con pool de threads



Servidor multithread con pool de threads

servidor.c

```
#include "mensaje.h"
#include <mqueue.h>
#include <pthread.h>
#include <stdbool.h>
#include <string.h>
#include <stdio.h>


#define MAX_THREADS  10
#define MAX_PETICIONES  256

struct peticion buffer_peticiones[MAX_PETICIONES];   // buffer

int n_elementos;                    // elementos en el buffer de peticiones
int pos_servicio = 0;

pthread_mutex_t mutex;
pthread_cond_t no_lleno;
pthread_cond_t no_vacio;

pthread_mutex_t mfin;
int fin=false;
```

Servidor multithread con pool de threads

servidor.c

```
int main(void)
{
    mqd_t q_servidor;                   /* cola del servidor */
    struct peticion mess;               /* mensaje a recibir */
    struct mq_attr q_attr;              /* atributos de la cola */
    pthread_attr_t t_attr;              /* atributos de los threads */
    pthread_t thid[MAX_THREADS];
    int error;
    int pos = 0;

    q_attr.mq_maxmsg = 10;
    q_attr.mq_msgsize = sizeof(struct peticion);
    q_servidor = mq_open("/SERVIDOR_SUMA", O_CREAT|O_RDONLY,  0700, &q_attr);
    if (q_servidor == -1) {
            perror("No se puede crear la cola de servidor");
            return 1;
    }
    pthread_mutex_init(&mutex,NULL);
    pthread_cond_init(&no_lleno,NULL);
    pthread_cond_init(&no_vacio,NULL);
    pthread_mutex_init(&mfin,NULL);
```

Servidor multithread con pool de threads

servidor.c

```
/* Creación del pool de threads */
 pthread_attr_init(&t_attr);
 for (int i = 0; i < MAX_THREADS; i++)
      if (pthread_create(&thid[i], NULL, servicio, NULL) !=0){
            perror("Error creando el pool de threads\n");
            return 0;
      }

 while (true) {
    error = mq_receive(q_servidor, (char *) &mess, sizeof(struct peticion), 0);
    if (error == -1 )  break;

    pthread_mutex_lock(&mutex);
    while (n_elementos == MAX_PETICIONES)
             pthread_cond_wait(&no_lleno, &mutex);
    buffer_peticiones[pos] = mess;
    pos = (pos+1) % MAX_PETICIONES;
    n_elementos++;
    pthread_cond_signal(&no_vacio);
    pthread_mutex_unlock(&mutex);

}  /* FIN while */
```

Servidor multithread con pool de threads

servidor.c

```
        pthread_mutex_lock(&mfin);
        fin=true;
        pthread_mutex_unlock(&mfin);


        pthread_mutex_lock(&mutex);
        pthread_cond_broadcast(&no_vacio);
        pthread_mutex_unlock(&mutex);


        for (int i=0;i<MAX_THREADS;i++)
                pthread_join(thid[i],NULL);


        pthread_mutex_destroy(&mutex);
        pthread_cond_destroy(&no_lleno);
        pthread_cond_destroy(&no_vacio);
        pthread_mutex_destroy(&mfin);


        return 0;
} /* Fin main */
```

Servidor multithread con pool de threads

servidor.c

```
void servicio(void ){
        struct peticion mensaje;      /* mensaje local */
        mqd_t q_cliente;              /* cola del cliente */
        int resultado;                /* resultado de la operación */

        for(;;){
                pthread_mutex_lock(&mutex);
                while (n_elementos == 0) {
                        if (fin==true) {
                                fprintf(stderr,"Finalizando servicio\n");
                                pthread_mutex_unlock(&mutex);
                                pthread_exit(0);
                        }
                        pthread_cond_wait(&no_vacio, &mutex);
                }

                mensaje = buffer_peticiones[pos_servicio];
                pos_servicio = (pos_servicio + 1) % MAX_PETICIONES;
                n_elementos --;
                pthread_cond_signal(&no_lleno);
                pthread_mutex_unlock(&mutex);
```

Servidor multithread con pool de threads

servidor.c

```
     /* procesa la peticion */


     /* ejecutar la petición del cliente y preparar respuesta */
      if (mensaje.op ==0)
            resultado = mensaje.a + mensaje.b;
      else
            resultado = mensaje.a - mensaje.b;
      /* Se devuelve el resultado al cliente */
      /* Para ello se envía el resultado a su cola */
       q_cliente = mq_open(mensaje.q_name, O_WRONLY);
       if (q_cliente == -1)
            perror("No se puede abrir la cola del cliente");
       else {
            mq_send(q_cliente, (const char *) &resultado, sizeof(int), 0);
            mq_close(q_cliente);
       }
   } // FOR
       pthread_exit(0);

} // servicio
```

Modelo cliente-servidor en MPI

- En MPI una aplicación paralela puede conectarse a otra aplicación paralela y establecer entre ellas un modelo cliente-servidor
- Aplicación paralela (servidor):
  - MPI_Open_port
  - MPI_Comm_accept
- Aplicación paralela (cliente):
  - MPI_Comm_connect

Conexión de dos aplicaciones


```
MPI_Comm_connect( port_name, MPI_INFO_NULL, 0,
                                  MPI_COMM_WORLD, &server );
```

Comunicador con la aplicación servidora


```
MPI_Open_port(MPI_INFO_NULL, port_name);
MPI_Comm_accept(port_name, MPI_INFO_NULL, 0, MPI_COMM_WORLD,&client);
```

Comunicador con la aplicación cliente

Conexión de dos aplicaciones


```
MPI_Comm_connect( port_name, MPI_INFO_NULL, 0,
                                    MPI_COMM_WORLD, &server );
```

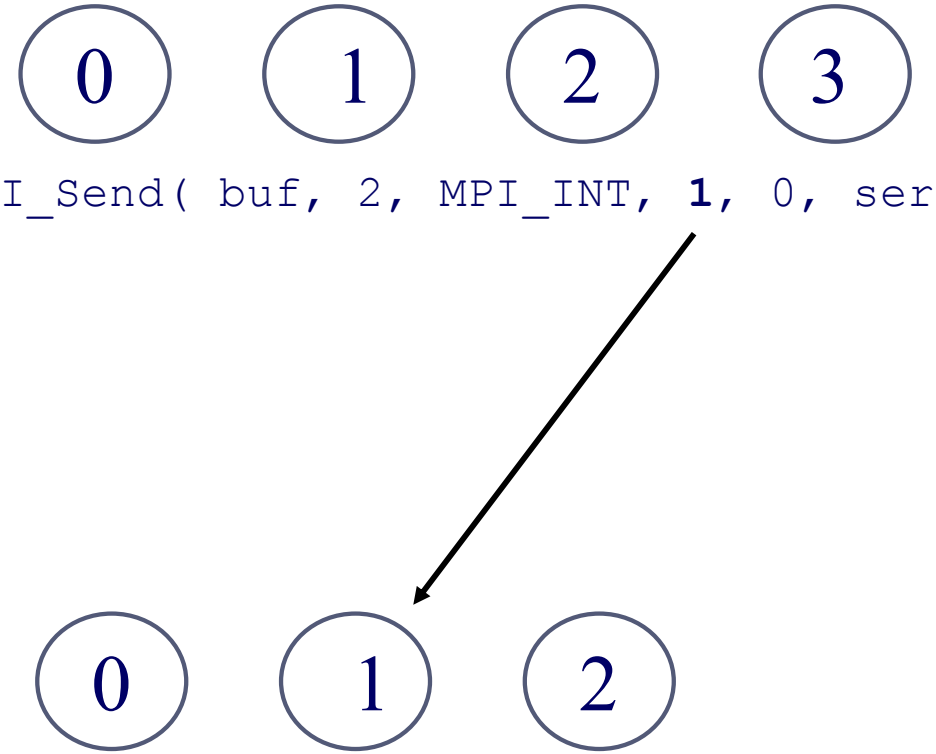
Comunicador con la aplicación servidora

Con `MPI_COMM_WORLD` son operaciones colectivas


```
MPI_Open_port(MPI_INFO_NULL, port_name);
MPI_Comm_accept(port_name, MPI_INFO_NULL, 0, MPI_COMM_WORLD,&client);
```

Comunicador con la aplicación cliente

Intercambio de datos




```
MPI_Recv(buf, 2, MPI_INT, MPI_ANY_SOURCE, 0, client, &status);
MPI_Send(buf, 1, MPI_INT,  status.MPI_SOURCE,0, client);
```

Conexión de dos aplicaciones


```
MPI_Comm_connect( port_name, MPI_INFO_NULL, 1,
                                    MPI_COMM_SELF, &server_1 );
```

Comunicador con el proceso servidor

Con `MPI_COMM_SELF` no son operaciones colectivas


```
MPI_Open_port(MPI_INFO_NULL, port_name);
MPI_Comm_accept(port_name, MPI_INFO_NULL, 0, MPI_COMM_SELF,&client);
```

Comunicador con el proceso cliente que se ha conectado

Ejercicio

Se desea diseñar un modelo de vector distribuido. Sobre un vector distribuido se definen los siguientes servicios:

- int **init** (char *nombre, int N). Este servicio permite inicializar un array distribuido de N números enteros. La función devuelve 1 cuando el array se se ha creado por primera vez. En caso de que el array ya esté creado, la función devuelve 0. La función devuelve -1 en caso de error.
- int **set**(char *nombre, int i, int valor). Este servicio inserta el valor en la posición i del array nombre.
- int **get**(char *nombre, int i, int *valor). Este servicio permite recuperar el valor del elemento i del array nombre.

Considere un sistema que utiliza colas de mensajes POSIX. Diseñe e implemente un sistema que implemente este servicio de vectores distribuidos.