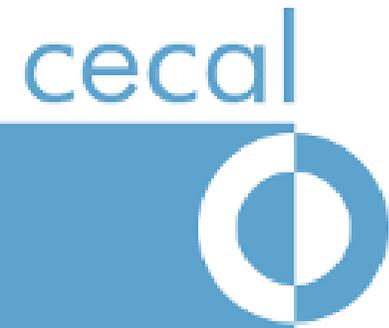


# COMPUTACIÓN DE ALTA PERFORMANCE

Curso 2024

Sergio Nesmachnow (sergion@fing.edu.uy)

Centro de Cálculo





# TEMA 5

## MECANISMOS DE PROGRAMACION PARALELA EN LENGUAJE C

# CONTENIDO

1. Clasificación de los mecanismos de comunicación
2. Creación de procesos
3. Inter-Process Communication (IPC)
  - Pipes
  - Semáforos
  - Cola de mensajes
  - Memoria compartida
  - Señales
4. Sockets
5. Remote Procedure Call (RPC)
6. Middleware de comunicación orientado a mensajes

# 5.1: CLASIFICACIÓN DE LOS MECANISMOS DE COMUNICACIÓN

- Las comunicaciones y sincronizaciones son necesarias e imprescindibles para lograr la cooperación en sistemas multiproceso y en especial en sistemas distribuidos.
- Las comunicaciones entre procesos pueden clasificarse de diferentes maneras de acuerdo a sus características:
  - sincrónica o asincrónica
  - persistente o transitoria
  - directa o indirecta
  - simétrica o asimétrica

- **Comunicaciones sincrónicas**
  - Modelo de puntos de sincronización (*rendezvous*): el proceso emisor permanece bloqueado esperando a que el proceso receptor invoque la primitiva de recepción (o viceversa).
  - Es **no local**: requiere de una acción en el proceso remoto.
  - La comunicación funciona como **punto de sincronización** entre procesos y permite intercambiar datos de forma coordinada (durante el rendezvous).
- **Comunicaciones asincrónicas**
  - El proceso emisor continúa con su ejecución inmediatamente después de enviar el mensaje al receptor, independientemente de su recepción.
  - Es una operación completamente **local**.
  - Permite explotar la ejecución simultánea en sistemas distribuidos.
  - Se implementa mediante estructuras de datos que permiten almacenar los mensajes (buffers, colas, etc.).

- **Comunicaciones persistentes**
  - No es necesario que el receptor esté operativo al mismo tiempo que se realiza la comunicación (ej.: correo electrónico).
  - El mensaje se almacena tanto tiempo como sea necesario para poder ser entregado.
  - Requiere de espacios de almacenamiento (colas, buffers, etc.)
- **Comunicaciones transitorias**
  - Requieren que el receptor esté operativo en el instante en que se lleva a cabo la comunicación.
  - El mensaje se descarta (no será entregado) si el receptor no está operativo.

- **Comunicaciones directas**
  - Se trabaja con primitivas explícitas para enviar y recibir, que indican los identificadores de los procesos que se comunican.
  - Primitiva `send(mensaje, D)`, para enviar un mensaje al proceso D.
    - Siempre se debe especificar el proceso destino.
  - Primitiva `receive(mensaje, O)`, para recibir un mensaje desde el proceso O.
    - La primitiva de recepción **siempre** debe especificar el proceso origen y el mensaje.
    - En general, existen modos promiscuos para la primitiva de recepción, que puede esperar a recibir un mensaje de un proceso cualquiera.
- **Comunicaciones indirectas**
  - La comunicación está desacoplada del receptor y esta basada en una herramienta o instrumento específico que permite anonimizarlo.

- **Comunicaciones simétricas**
  - Todos los procesos tienen las mismas capacidades para enviar o recibir.
  - Es llamada comunicación bidireccional o full-duplex para el caso de dos procesos.
- **Comunicaciones asimétricas**
  - Un conjunto de procesos distinguidos tienen las capacidades para enviar. El resto de los procesos solo tienen la capacidad de recibir información.
  - Es llamada comunicación unidireccional o half-duplex para el caso de dos procesos.
  - Es un tipo común para sistemas que hospedan servidores en Internet.

- Durante el curso se hará énfasis en comunicaciones de tipo:
  - Sincrónicas y asincrónicas
  - Transitorias y persistentes
  - Directas
  - Simétricas

# 5.2: CREACION DE PROCESOS

# fork

- La primitiva **fork** crea un proceso hijo del proceso que realiza la invocación.
- Se crea una copia exacta del proceso que la invoca, pero con otro PID (Process Identifier).
- Valor de retorno: el padre recibe el PID del hijo, y el hijo recibe 0.

```
#include <sys/types.h>
#include <unistd.h>
...
pid_t pid;
...
if ((pid = fork()) == 0) {
    /* proceso hijo */
} else {
    /* proceso padre */
}
...
```

PID    PPID  
↓      ↓

user	11700	10875	0	11:04	pts/4	00:00:55	./proceso_fork
user	11701	11700	0	11:04	pts/4	00:00:02	./proceso_fork

# fork

- Se copia el área de datos, incluyendo archivos abiertos y sus descriptores asociados.
- Bloqueos sobre archivos y señales pendientes no son heredados.
- En Linux se implementa mediante *copy-on-write* de las páginas de memoria.
- El único costo es duplicar las tablas de páginas y crear la estructura del bloque de control de proceso (PCB).
- Puede retornar errores por memoria insuficiente:
  - **EAGAIN**: `fork()` no puede reservar memoria suficiente para copiar las tablas de páginas del proceso padre y crear la estructura de control del proceso hijo.
  - **ENOMEM**: `fork()` falló al reservar la memoria necesaria para las estructuras necesarias del núcleo porque la memoria es insuficiente.

# fork - wait

- La primitiva `wait` le notifica a un proceso cuando alguno de sus hijos termina su ejecución y le retorna su valor de finalización
- Si el padre muere antes que el hijo, el hijo queda huérfano
  - Su padre pasa a ser el proceso "init" (PID=1) de Linux
- Si el hijo termina pero el padre no acepta su valor de retorno (usando `wait`), el proceso pasa a estado *zombie*
  - Un proceso en estado *zombie* libera todos sus recursos, salvo porque mantiene una entrada en la tabla de procesos (mantiene su PID)
- `wait` **bloquea la ejecución del padre** hasta que termina la ejecución de un hijo y retorna el PID y el valor de retorno del hijo que terminó
- El proceso hijo puede devolver un valor de retorno usando la función `exit` de C

# fork - wait

```
1. main () {
2.     int pid, mypid, status;
3.     if ((pid = fork()) == 0) {
4.         // Proceso hijo
5.         mypid = getpid();
6.         fprintf(stdout, "H: %d\n", mypid);
7.         exit(0);
8.     } else {
9.         // Proceso padre
10.        mypid = getpid();
11.        fprintf(stdout, "P: %d, H: %d\n", mypid, pid);
12.        pid = wait(&status);
13.        fprintf(stdout, "pid wait: %d\n", pid);
14.        exit(0);
15.    }
16. }
```

```
$ ./proceso_fork
H: 14952
P: 14951, H: 14952
pid wait: 14952
```



# 5.3: INTER-PROCESS COMMUNICATION en LINUX

# INTER-PROCESS COMMUNICATION

- IPC permite la comunicación de información y sincronización entre procesos que ejecutan en un mismo hardware
- Está disponible y es ampliamente usada en los sistemas Unix/Linux
- Las dos implementaciones más aceptadas de IPC son:
  - System V: una de las primeras versiones comerciales de Unix, fue y es un estándar de facto
  - Portable Operating System Interface (POSIX): estándar definido por IEEE

# INTER-PROCESS COMMUNICATION

- Se divide en cuatro herramientas:
  1. Pipes: sincronización y comunicación de información
  2. Memoria compartida: comunicación de información
  3. Semáforos: sincronización
  4. Colas de mensajes: sincronización y comunicación de información
- POSIX define versiones **anónimas** y **nombradas** de estos mecanismos
  - Las versiones anónimas solamente sirven para comunicar procesos emparentados
  - Las versiones nombradas permiten comunicar cualquier tipo de procesos.
- Adicionalmente, las señales permiten la comunicación de información

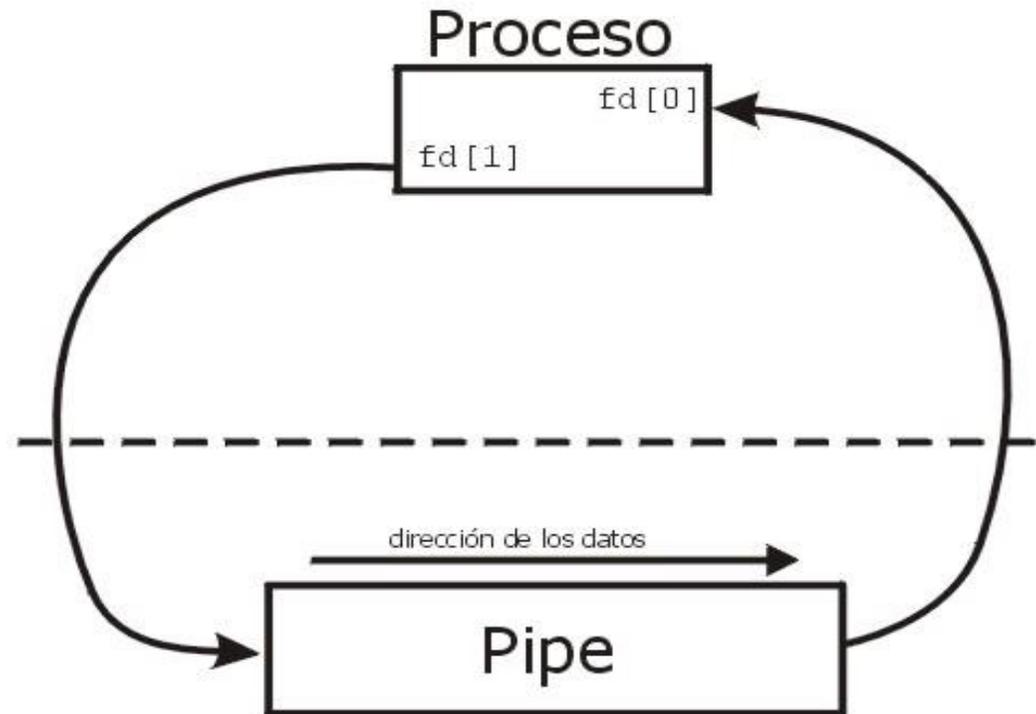
# PIPES

- Mecanismo de comunicación y sincronización entre procesos padre e hijo
- Es asimétrico: define pipes unidireccionales (*half-duplex pipe*)
- Es un *file descriptor* lo que permite acceso a través de las operaciones `read()` y `write()`
- La operación `read()` es **bloqueante**
- Implementado en el núcleo del sistema operativo
- En Linux, la capacidad de los pipes es de 64KB, pero se asegura una escritura atómica solamente para buffers de 4KB o menos

# PIPES

- Es necesario **cerrar** el file descriptor de la dirección del pipe que no se vaya a utilizar
- Solo existen versiones **anónimas** de los pipes por lo que es necesario contar con un mecanismo para compartir un pipe entre padre e hijo

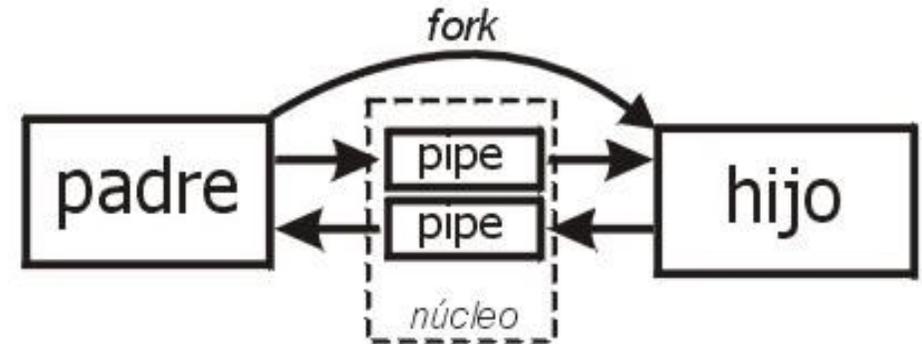
```
int main () {
    int fd[2];
    if (pipe(fd)) {
        perror("Error");
        exit(1);
    }
    int pid=fork();
    ...
}
```



# PIPES

```
int main () {
int fd1[2], fd2[2], pid;

/* creación del pipe */
if (pipe(fd1) || pipe(fd2)) {
    perror("Error\n");
    exit (1) ;
}
if ((pid = fork ()) == 0) {
    /* proceso hijo */
    close (fd1[1]);
    close (fd2[0]);
    procHijo(fd1[0],fd2[1]);
    close (fd1[0]);
    close (fd2[1]);
}
...
}
```



```
...
else {
    /* proceso padre */
    close (fd1[0]);
    close (fd2[1]);
    procPadre(fd1[1],fd2[0]);
    close (fd1[1]);
    close (fd2[0]);
    waitpid(pid,NULL,0);
}
...
}
```

# FIFO (PIPES NOMBRADOS)

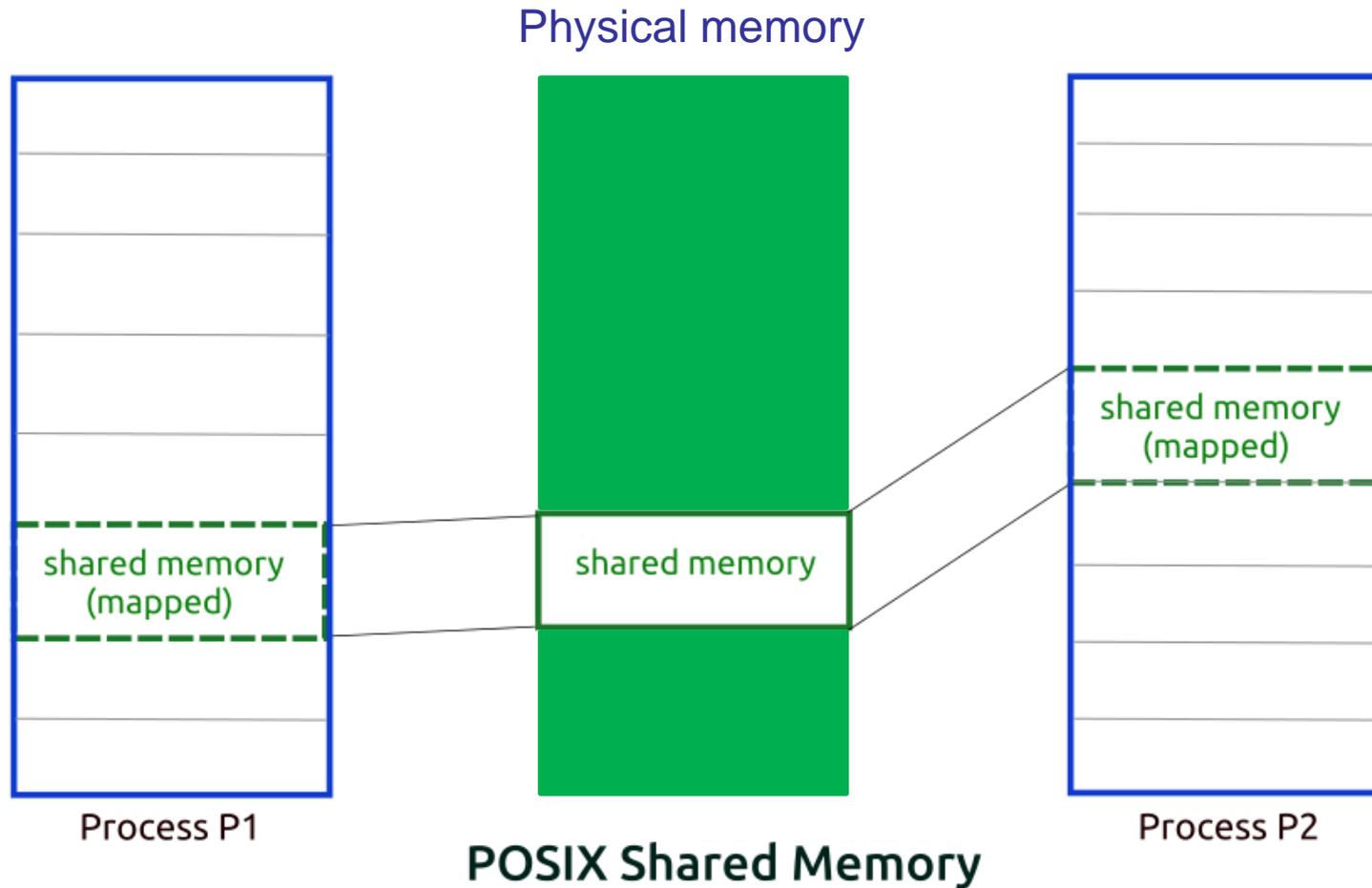
- Permiten la comunicación entre procesos no emparentados
- Al igual que los pipes son unidireccionales (half-duplex)
- Tienen persistencia a nivel del núcleo
  - El pipe con nombre **persiste en el sistema de archivos** para uso posterior
- Existe en el sistema de archivos como un archivo especial del VFS
  - El comando/función `mkfifo` permite crear un FIFO
  - El comando `ls -l` identifica el FIFO con el carácter descriptor `p`
  - El comando/función `unlink` permite eliminar un FIFO

```
$ mkfifo MIFIFO  
$ ls -l MIFIFO  
prw-r--r-- ... MIFIFO
```

# MEMORIA COMPARTIDA

- La forma mas eficiente de comunicar dos procesos **en la misma máquina**
  - El proceso accede a través de **direccionamiento directo** y no a través del núcleo del sistema
- Requiere sincronización explícita para acceder al recurso compartido
- POSIX solamente define memoria compartida nombrada con persistencia a nivel de núcleo
- Implementación usando mapeo de archivos
  - Un espacio de memoria en el núcleo del sistema es compartido por todos los procesos que quieran acceder y posean permisos
  - En Linux: disponibles en `/dev/shm`

# MEMORIA COMPARTIDA



# MEMORIA COMPARTIDA

- `shm_open(const char * name, ...)`
  - Obtiene un descriptor a un segmento de memoria compartida
- `mmap(void *, size_t len, int, int fd, int, off_t)`
  - Mapea un archivo (a través del descriptor `fd`) al espacio de direccionamiento de memoria del proceso
- `shm_unlink(const char *name)`
  - Destruye un espacio de memoria compartida

```
void* ptr;  
int fd;  
  
fd = shm_open(pathname, flags, FILE_MODE);  
ptr = mmap(NULL, length, modo, MAP_SHARED, fd, 0);
```

# MEMORIA COMPARTIDA

```
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int fd = shm_open("/shmem-ej", O_CREAT, 0777);
    ftruncate(fd, 1024);

    void* addr = mmap(NULL, 1024, PROT_READ |
        PROT_WRITE, MAP_SHARED, fd, 0);

    char* msg = "hola mundo!";
    memcpy(addr, msg, strlen(msg));

    exit(EXIT_SUCCESS);
}
```

# MEMORIA COMPARTIDA

```
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int fd = shm_open("/shmem-ej", O_RDONLY, 0);

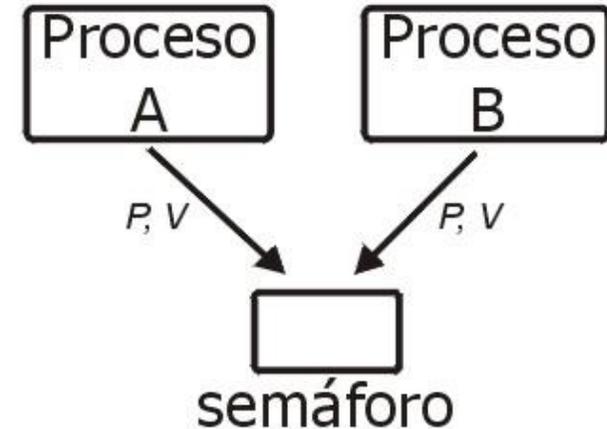
    void* addr = mmap(NULL, 1024, PROT_READ,
MAP_SHARED, fd, 0);

    printf("%s\n", addr);

    shm_unlink("/shmem-ej");
    exit(EXIT_SUCCESS);
}
```

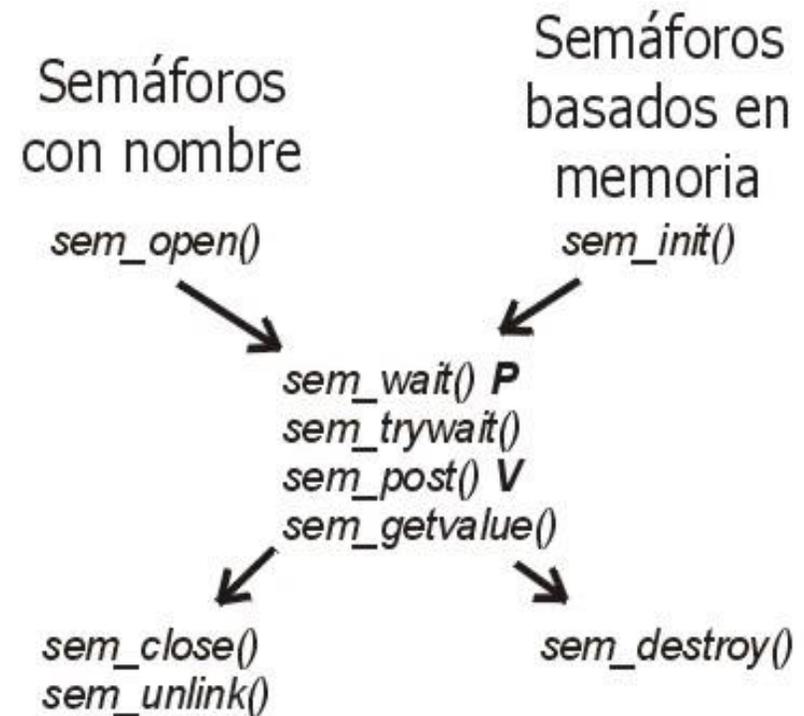
# SEMÁFOROS

- Son una herramienta de sincronización
- Tipos:
  - Binarios: toman valores entre 0 y 1
  - No binarios: toman valores enteros
- Operaciones:
  - P: Proberen (“test”)
  - V: Verhogen (“incrementar”)
- La operación P decrementa el valor del semáforo
  - Si el valor del semáforo es 0, entonces P bloquea al proceso
- La operación V incrementa el valor del semáforo



# SEMÁFOROS

- POSIX provee semáforos no binarios, anónimos y nombrados
- Semáforos anónimos
  - Implementados a través de memoria compartida
- Semáforos nombrados
  - A través del sistema de archivos. En Linux: disponible en /dev/shm
- Persistencia a nivel de núcleo
- Uso entre procesos emparentados y no emparentados
- POSIX no define un orden de espera para los procesos



# SEMÁFOROS ANÓNIMOS

Semáforos basados en memoria compartida:

```
1. int main(int argc, char *argv[])
2. {
3.     int fd = shm_open("/sem-ej", O_CREAT, 0);
4.     ftruncate(fd, sizeof(sem_t));

5.     sem_t* s = mmap(NULL, sizeof(sem_t),
6.                     PROT_READ | PROT_WRITE,
7.                     MAP_SHARED, fd, 0);

8.     sem_init(s, 1, 0);
9.     sem_wait(s);
10.    sem_destroy(s);
11.    shm_unlink("/sem-ej");
12.
13.    exit(EXIT_SUCCESS);
14. }
```

# SEMÁFOROS ANÓNIMOS (2)

```
1. int main(int argc, char *argv[])
2. {
3.     int fd = shm_open("/sem-ej", O_RDONLY, 0);

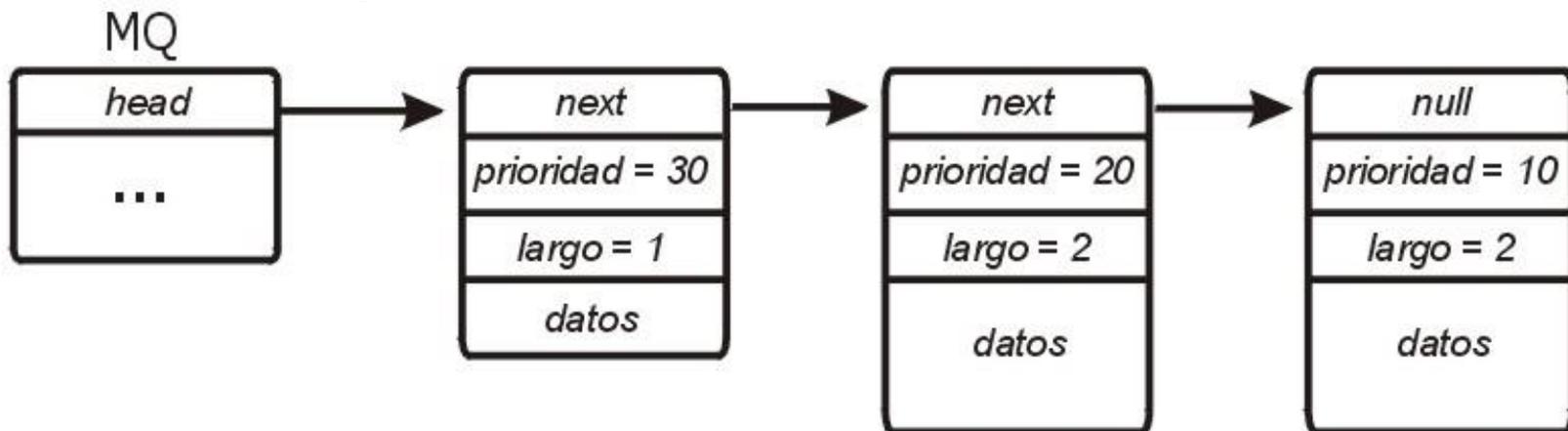
4.     sem_t* s = mmap(NULL, sizeof(sem_t),
5.                     PROT_READ | PROT_WRITE,
6.                     MAP_SHARED, fd, 0);

7.     sem_post(s);

8.     exit(EXIT_SUCCESS);
9. }
```

# COLA DE MENSAJES

- Herramienta de sincronización y de envío de información
- Los procesos intercambian información a través del envío de mensajes a diferentes colas
- Manejan prioridades de envío y recepción
- Implementadas a través de un archivo en el VFS con persistencia a nivel de núcleo
- Permite la generación de una señal o el inicio de un hilo de ejecución cuando un mensaje es recibido en la cola



# COLA DE MENSAJES

- Operaciones principales:
  - `mq_open(const char * name, int oflag, ...)`
    - Crea o abre una cola de mensajes
  - `mq_send(mqd_t mqdes, const char *ptr, size_t len, unsigned int prio)`
    - Envía un mensaje a la cola con la prioridad prio
  - `mq_receive(mqd_t mqdes, char * ptr, size_t len, unsigned int * prio)`
    - Recibe el mensaje de mayor prioridad y que hace más tiempo que está en la cola (operación bloqueante)
  - `mq_close(mqd_t mqdes)`
    - Cierra el acceso a la cola de mensajes para el proceso
  - `mq_unlink(const char * name)`
    - Elimina una cola de mensajes

# COLA DE MENSAJES

```
#include <mqueue.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]){
    mqd_t mqd;
    mqd = mq_open("/mq-ej", O_CREAT, 0777, NULL);

    char* msg = "hola mundo!";
    mq_send(mqd, msg, strlen(msg), 1);

    exit(EXIT_SUCCESS);
}
```

# COLA DE MENSAJES

```
#include <mqueue.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    mqd_t mqd = mq_open("/mq-ej", O_RDWR, 0777, NULL);

    struct mq_attr attr;
    mq_getattr(mqd, &attr);
    void* buffer = malloc(attr.mq_msgsize);

    int prio;
    int numRead = mq_receive(mqd, buffer,
                             attr.mq_msgsize, &prio);

    exit(EXIT_SUCCESS);
}
```

- Interrupciones generadas a través del software que permiten generar eventos en los procesos
- Sirven para la comunicación entre dos procesos y entre el núcleo del sistema operativo y un proceso
- Un proceso puede enviarle una señal a otro proceso con la función  

```
int kill(pid_t pid, int sig)
```
- Cuando un proceso recibe una señal ocurre una acción por defecto, a menos que el proceso defina una función o procedimiento (*handler*) para atender la señal
- Cuando llega una señal al proceso interrumpido, los registros son salvados, y el handler de la señal es ejecutado
  - El proceso puede ser interrumpido en cualquier momento, las estructuras de datos podrían estar inconsistentes

- Las tres acciones más comunes en un handler son:
  - Establecer el valor de una variable booleana (bandera) y retornar
  - Reiniciar el proceso en algún punto conveniente
  - Liberar recursos y terminar el proceso
- Sin embargo, pueden definirse acciones específicas mas complejas para atender la interrupción correspondiente

# SEÑALES

- Cada proceso tiene una mascara (bitmask) que permite bloquear señales
- Mientras se ejecuta el handler de una señal, esa señal en particular se mantiene bloqueada
- Cuando una señal se encuentra bloqueada, esta no es despachada y permanece pendiente (aún cuando su acción por defecto sea ignorarla)
- La función `sigprocmask()` cambia la lista de señales bloqueadas
- La función `sigpending()` muestra las señales pendientes
- La función `sigsuspend()` permite suspender el proceso actual hasta la llegada una señal especifica

# SEÑALES

Señal	Acción	Descripción
SIGINT	Term	Interrumpido desde el teclado (Ctrl+C)
SIGILL	Core	Instrucción ilegal
SIGABRT	Core	Abortar proceso
SIGFPE	Core	Error de punto flotante
SIGKILL*	Term	Matar proceso
SIGSEGV	Core	Referencia inválida a memoria
SIGTERM	Term	Terminar proceso
SIGUSR1	Term	Reservada para el usuario
SIGUSR2	Term	Reservada para el usuario
SIGCHLD	Ign	Hijo finalizó
SIGCONT	Cont	Continuar proceso
SIGSTOP*	Stop	Detener proceso

- Las señales SIGKILL y SIGSTOP no pueden ser manejadas, bloqueadas o ignoradas

# SEÑALES

- El programador crea *handlers* llamando a la función `signal`, pasando como parámetro el numero de señal y un puntero a la función que la va a atender

```
#include <stdio.h>
void sigproc(void) {
    ...
}
void quitproc(void) {
    ...
}

main() {
    signal(SIGINT, sigproc);
    signal(SIGQUIT, quitproc);
    for(;;); /* infinite loop */
}
```

# 5.4: SOCKETS

# SOCKETS

- Mecanismo de comunicación en el modelo cliente/servidor
- Un proceso servidor “escucha” mensajes que llegan al socket y un proceso cliente es quién los envía
- Se establece luego una comunicación full-duplex
- Operaciones básicas con sockets: creación, apertura, lectura, escritura y cierre
- Dos dominios para los sockets:
  - Internet Domain Sockets: permiten la comunicación entre procesos a través de una interfaz de red (`AF_INET` o `AF_INET6`)
  - Unix Domain Sockets: permiten comunicar dos procesos en el mismo equipo (`AF_UNIX`)

# SOCKETS

- Dos tipos de sockets más usados:
- Datagram:
  - No usa conexión, mensajes empaquetados (SOCK\_DGRAM), protocolo UDP (no hay garantía de arribo, no se garantiza orden correcto).
  - Aplicaciones donde entregar la información no es muy crítico, y si es muy relevante alcanzar la eficiencia: e.g., streaming de video, VoIP.
- Stream:
  - Orientado a conexión, comunicación secuencial con ACKs, flujo de datos (SOCK\_STREAM), protocolo TCP (libre de errores, garantiza ordenación correcta).
  - Aplicaciones donde es muy importante información: e.g., transferencia de archivos.

# ESQUEMA de DATAGRAM SOCKETS - CLIENTE

```
int main(int argc, char**argv){
    int sockfd, n;
    struct sockaddr_in servaddr;
    char sendline[1000]; char recvline[1000];

    sockfd = socket(AF_INET,SOCK_STREAM,0);

    bzero(&servaddr,sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    servaddr.sin_port = htons(32000);

    connect(sockfd,(struct sockaddr *)&servaddr, sizeof(servaddr));

    sendto(sockfd,sendline,1000,0,(struct sockaddr *)&servaddr,
           sizeof(servaddr));
    n = recvfrom(sockfd,recvline,1000,0,NULL,NULL);
}
```

# ESQUEMA de STREAM SOCKETS - SERVIDOR

```
int main(int argc, char**argv)
{
    int connfd, n;
    struct sockaddr_in servaddr,cliaddr;
    socklen_t clilen;
    char mesg[1000];

    int fd = socket(AF_INET,SOCK_STREAM,0);

    bzero(&servaddr,sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(32000);

    bind(fd,(struct sockaddr *)&servaddr,sizeof(servaddr));
    listen(fd,1024);

    /* continúa en slide 40 ... */
}
```

# ESQUEMA de STREAM SOCKETS - SERVIDOR

```
for(;;)
{
    clilen = sizeof(cliaddr);
    connfd = accept(fd, (struct sockaddr*)&cliaddr, &clilen);

    n = recvfrom(connfd, msg, 1000, 0,
                 (struct sockaddr*)&cliaddr, &clilen);
    sendto(connfd, msg, n, 0,
           (struct sockaddr*)&cliaddr, clilen);

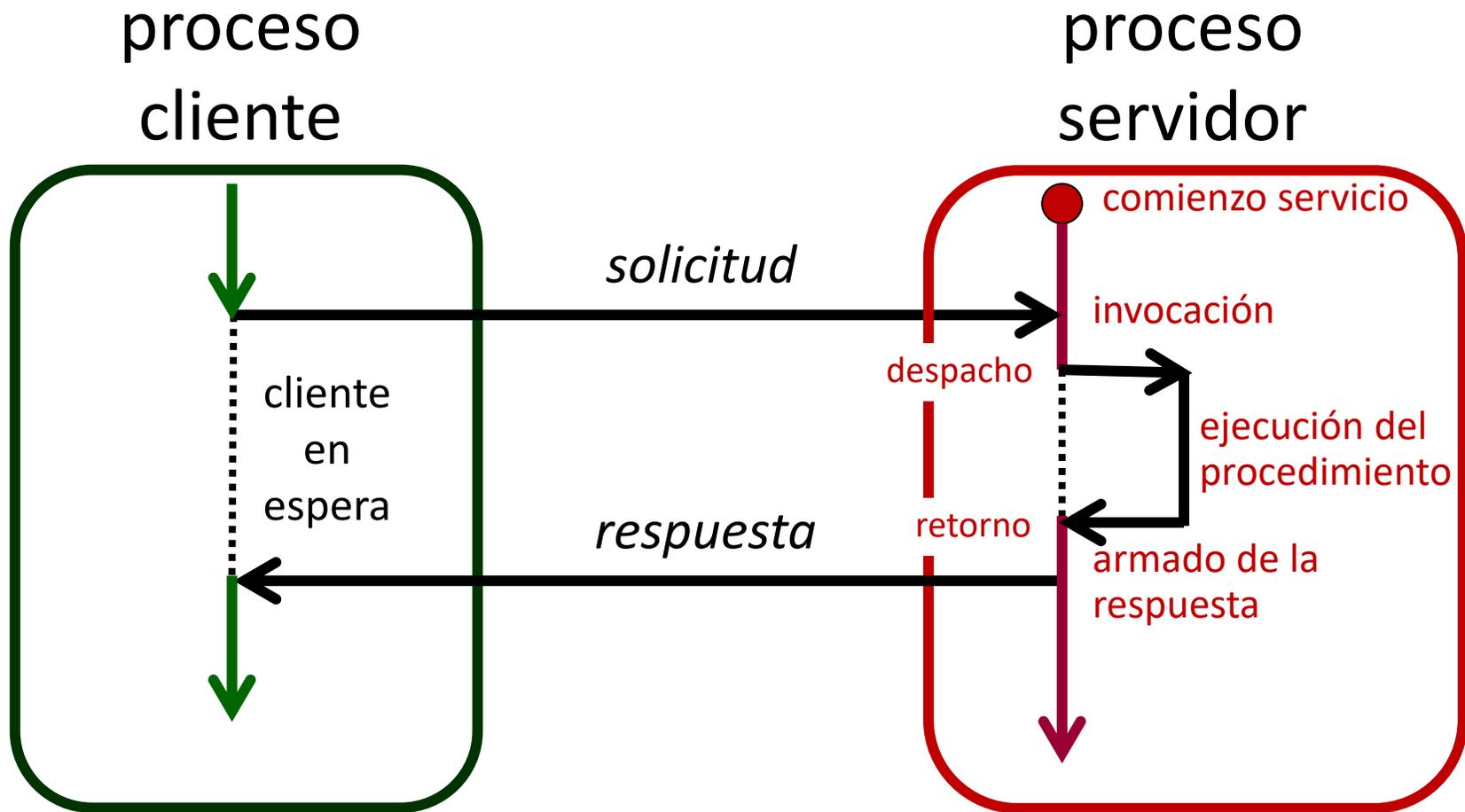
    close(connfd);
}
}
```

# 5.5: REMOTE PROCEDURE CALL

# REMOTE PROCEDURE CALL

- Es una herramienta que permite la invocación a una función o procedimiento que se encuentra en un **proceso remoto**.
- Los procedimientos son declarados en un proceso servidor y luego son accedidos por clientes que ejecutan en equipos remotos o locales.
- El modelo es conceptualmente simple y permite a los desarrolladores de aplicaciones *evitar los detalles de la programación de las comunicaciones* a través de la red.
- Desde el punto de vista de la aplicación invocante (y por tanto, para el programador) la llamada a una función remota es muy similar (casi idéntica) a la invocación a una función definida localmente.
- El modelo provee transparencia respecto al entorno de ejecución de una aplicación distribuida.

# REMOTE PROCEDURE CALL



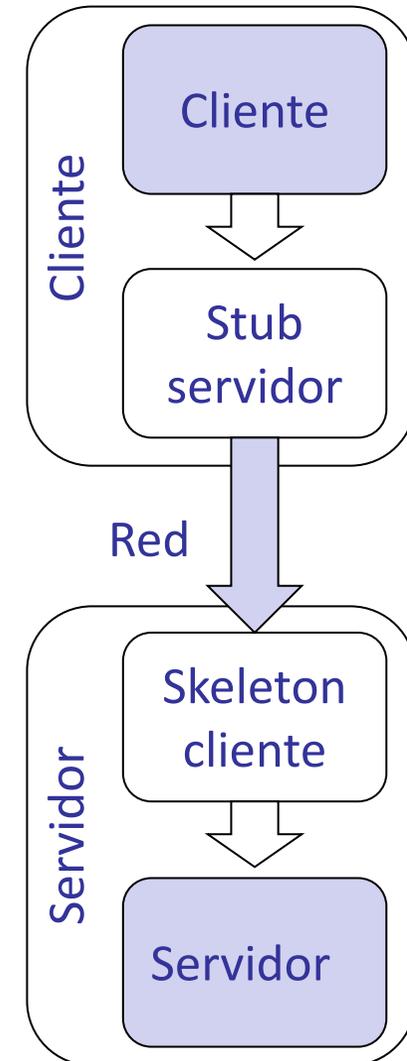
Esquema de invocación a un procedimiento remoto

# REMOTE PROCEDURE CALL

- Implementaciones de ejemplo:
  - Open Network Computing (ONC)
  - Apache Thrift
  - Apache Avro
- Permiten integrar diferentes lenguajes y representaciones de datos.
  - Incluyendo C/C++, Java, .NET, Python, y otros.
- La primera implementación popular de RPC fue desarrollada por SUN para sus sistemas UNIX y luego se transformó en un estándar: **ONC RPC**.

# REMOTE PROCEDURE CALL

- Idea clave: uso de “representantes” del cliente y del servidor.
- Representante del servidor en la máquina cliente (*stub*)
  - Juega el rol del servidor en la máquina cliente.
- Representante del cliente en la máquina servidor (*skeleton*)
  - Juega el rol del cliente en la máquina servidor.
- Proporcionan transparencia en la llamada remota.
- Son generados **automáticamente** en base a una *interfaz* definida para el procedimiento remoto.
  - El programador sólo debe programar el código del procedimiento remoto (servidor) y el código que realiza la invocación remota (cliente).



# RPC: COMPONENTES

- **Stub** (representante del servidor → recibe la llamada del cliente)
  - Proporciona transparencia en el lado del cliente.
  - Posee una interfaz idéntica al del procedimiento remoto.
  - Cada procedimiento remoto debe tener su propio stub.
- El cliente realiza una **llamada local** al procedimiento del stub como si fuera el **servidor real**
- Tareas realizadas por el stub:
  - Localizar al servidor que implemente el procedimiento remoto usando un servicio de *binding*.
  - Empaquetar los parámetros de entrada (aplanado, *marshalling*) en un formato común para cliente y servidor.
  - Enviar el mensaje al servidor.
  - Esperar la recepción del mensaje de respuesta.
  - Extraer resultados (desaplanado, *unmarshalling*) y retornarlos a la función que hizo la llamada.

# RPC: COMPONENTES

- **Skeleton** (representante del cliente → realiza la llamada al servidor)
  - Proporciona transparencia en el lado del servidor.
  - Conoce la interfaz ofrecida por el procedimiento remoto.
  - Cada procedimiento remoto debe tener su propio skeleton.
- **Realiza llamadas locales al servidor como si fuera el cliente real.**
- Es responsable de la *invocación real* al procedimiento remoto.
- Tareas realizadas por el skeleton:
  - Registrar el procedimiento en el servicio de binding.
  - Ejecutar un bucle de espera de mensajes y recibir peticiones.
  - Desempaquetar el mensaje (desaplanado, *unmarshalling*).
  - Determinar qué procedimiento concreto invocar, y luego invocarlo con los argumentos recibidos y recuperar el resultado.
  - Empaquetar el valor devuelto (aplanado, *marshalling*) en un mensaje.
  - Enviar el mensaje al stub del cliente.

- **Servicio de binding**
  - Responsable de la transparencia de localización.
  - Gestiona la asociación entre el nombre (y versión) del procedimiento remoto con su localización en la máquina servidor (dirección, puertos, skeleton, etc).
  - Realiza la búsqueda del skeleton de la implementación concreta del procedimiento remoto invocado por un cliente.
  - Selecciona el par (skeleton+servidor) que atenderá a cada invocación remota.
  - Ejemplos: portmapper (en ONC-RPC), rmiregistry (en Java-RMI), UDDI (en servicios web).

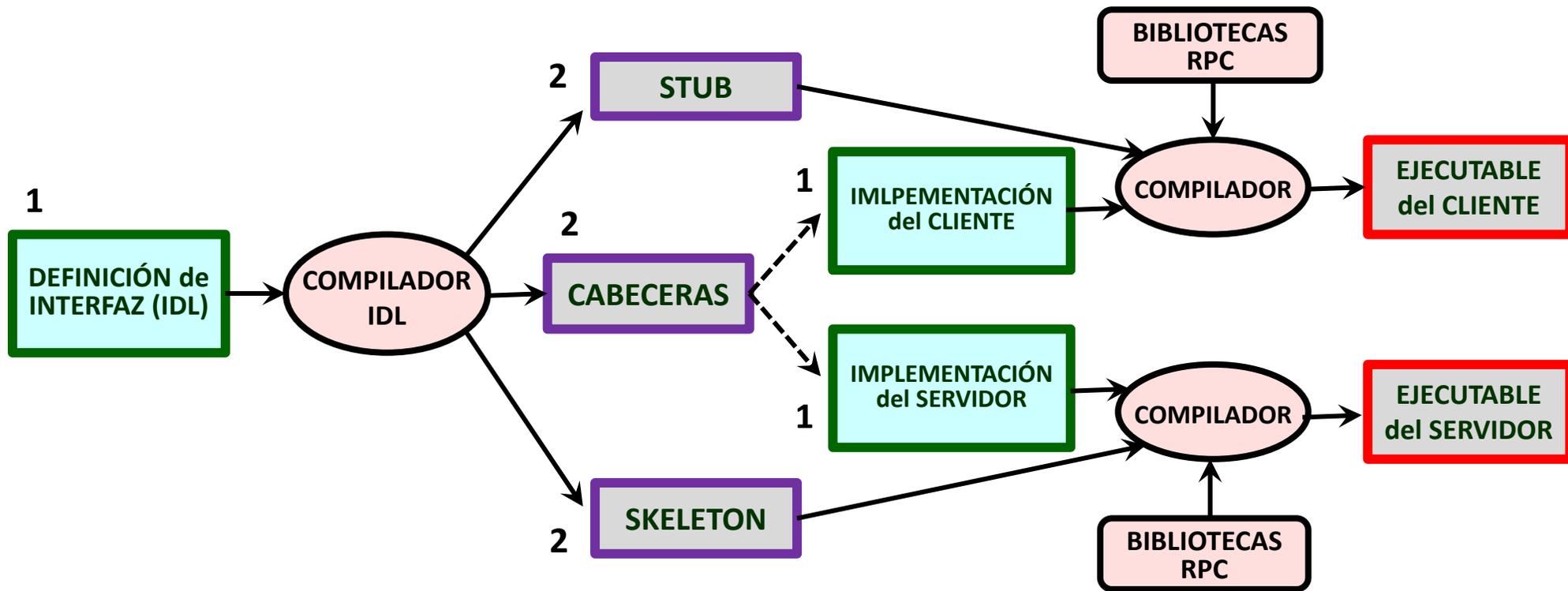
- **Compilador de interfaces**
- A partir de la descripción de la interfaz del procedimiento remoto, genera de forma automática el código del stub y del skeleton.
  - stub+skeleton **solo dependen de la interfaz** del procedimiento remoto.
- Dependiendo del entorno RPC, puede generar otros códigos adicionales que sean necesarios.
- La interfaz del procedimiento remoto contiene los datos necesarios para generar el par (stub+skeleton):
  - especifica la interfaz ofrecida por el procedimiento (parámetros de entrada, valor devuelto como resultado).
  - especifica cómo se realizará el aplanado/desaplanado.
  - opcionalmente puede aportar información que se usará para localizar el procedimiento remoto (e.g., número de versión).

# RPC: INTERFACES

- La interfaz del procedimiento remoto contiene los datos necesarios para generar el par (stub+skeleton).
- Existen dos enfoques para su definición:
  1. Puede estar definida en el mismo lenguaje de programación que implementa el servicio.
  2. Puede estar definida mediante un lenguaje de definición de interfaces independiente del lenguaje de programación (IDL: *interface definition language*).
    - Un compilador IDL lleva a cabo la traducción al lenguaje de implementación correspondiente.
    - Ejemplos: XDR usado en ONC-RPC, Corba-IDL en CORBA, especificaciones WSDL en servicios web.

# RPC: GENERACIÓN de CÓDIGO

- Esquema de componentes para la generación de código RPC



## REFERENCIAS

1

código del usuario

2

código autogenerado

3

utilitarios

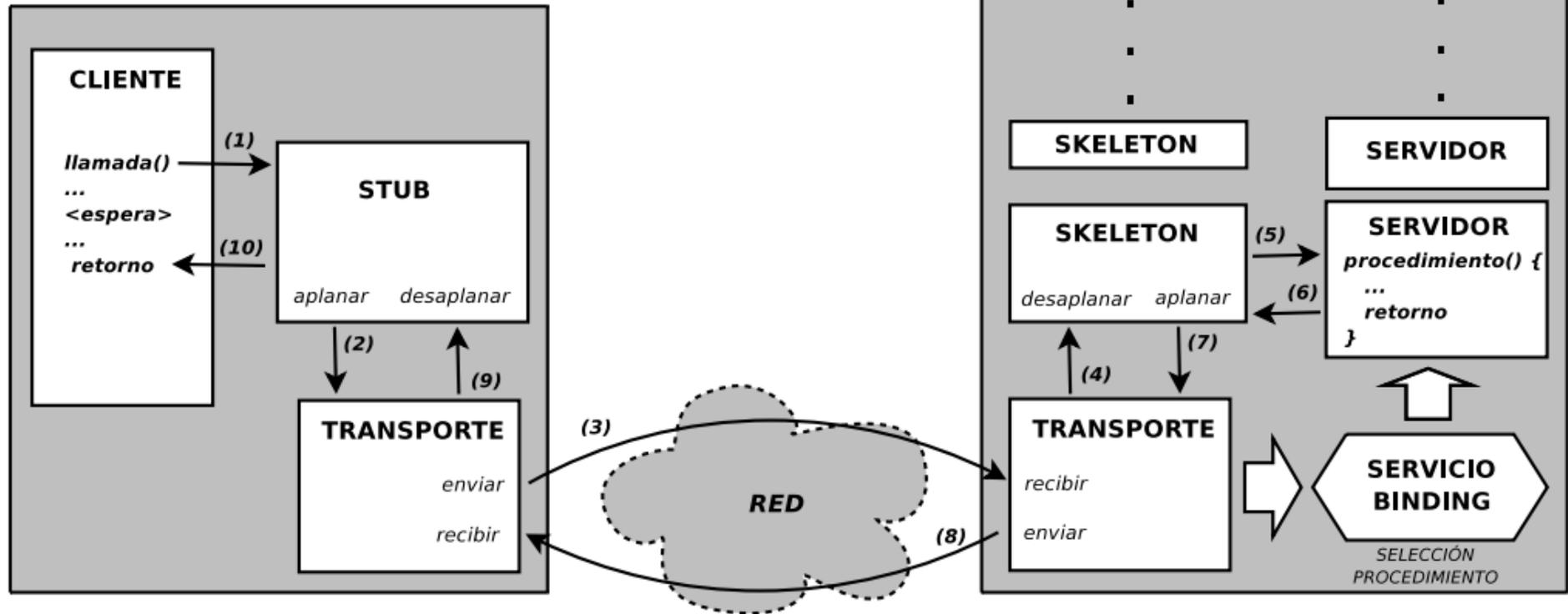
4

ejecutables

# RPC: FUNCIONAMIENTO

## Equipo servidor

## Equipo cliente

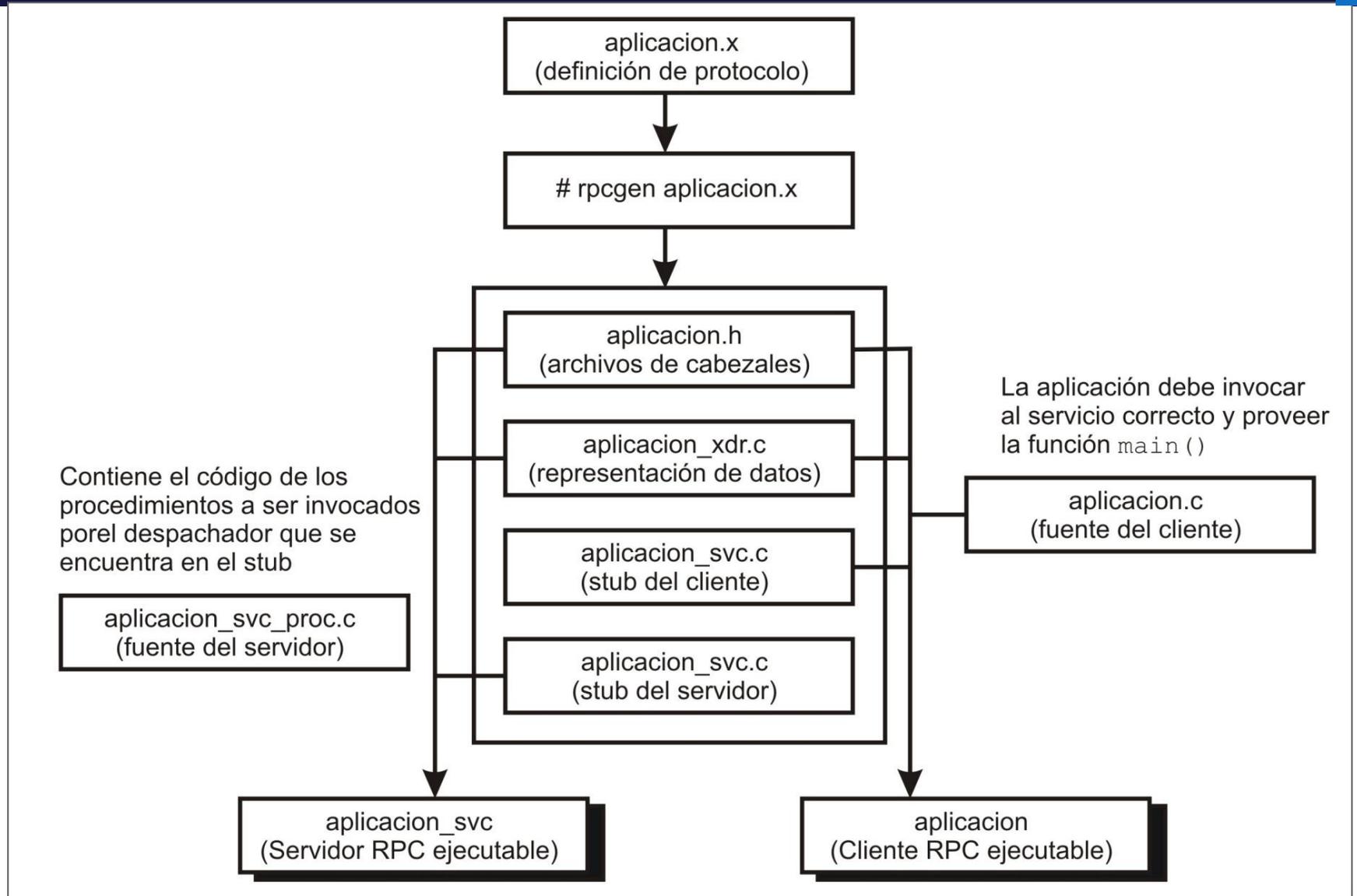


- |  |  |                      |   |   |
|--|--|----------------------|---|---|
| (1) parámetros                         | (3) MENSAJE DE PETICIÓN                | (5) parámetros       | (7) bloque con valor de retorno empaquetado | (9) bloque con valor de retorno empaquetado |
| (2) bloque con parámetros empaquetados | (4) bloque con parámetros empaquetados | (6) valor de retorno | (8) MENSAJE DE RESPUESTA                    | (10) valor de retorno                       |

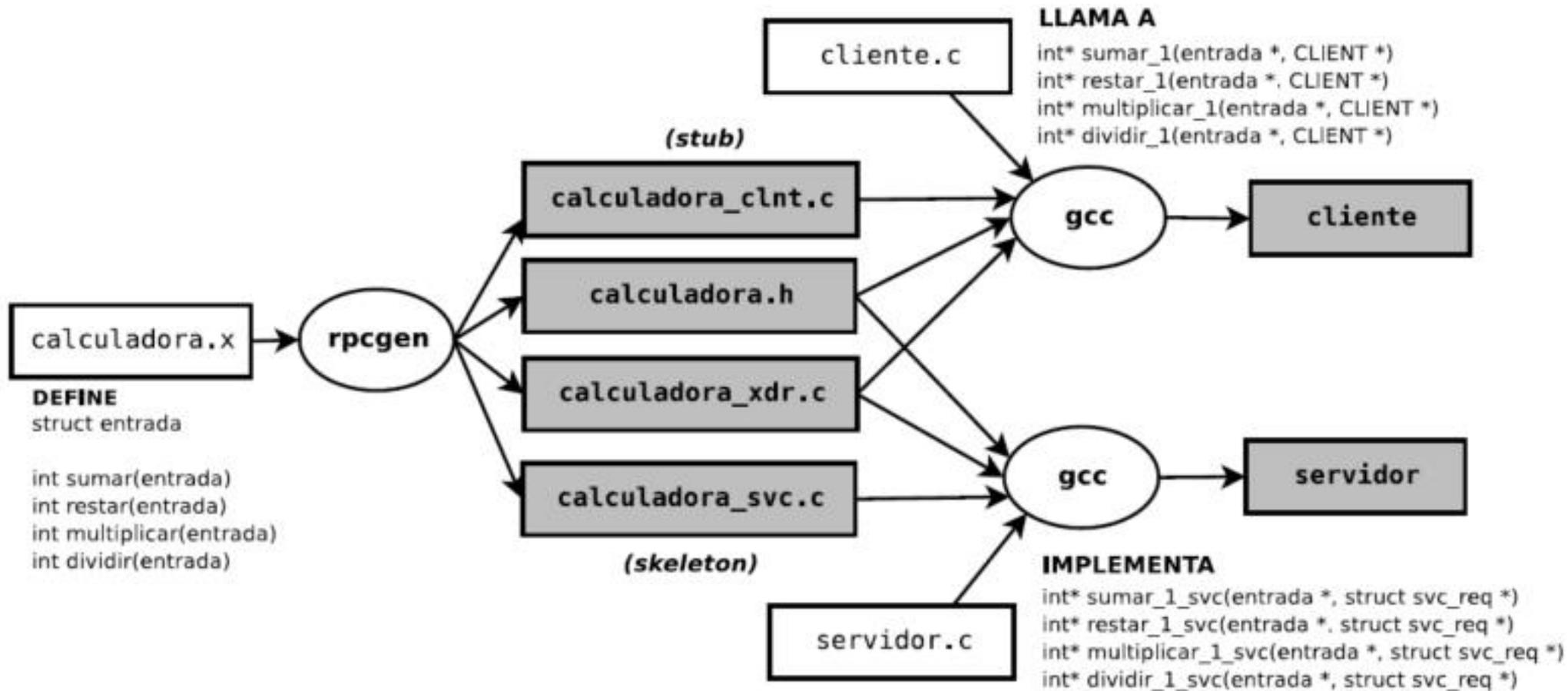
# CASO DE ESTUDIO: ONC RPC

- Se debe especificar la interfaz expuesta por el servidor mediante un lenguaje de descripción (Interface Description Language, IDL).
- El comando `rpcgen` (compilador de IDL) genera el código de los stubs para el cliente y de los skeletons para el servidor.
- Los stubs/skeletons se encargan de encapsular las tareas de comunicación a bajo nivel:
  - Transformación de datos (al formato eXternal Data Representation, XDR)
  - Invocación a procedimiento remoto (usando sockets)
- El cliente debe compilar utilizando los stubs del cliente generados por el comando `rpcgen`.
- El servidor debe compilar utilizando los skeletons del servidor generados por el comando `rpcgen`.

# CASO DE ESTUDIO: ONC RPC



# ONC RPC: FUNCIONAMIENTO

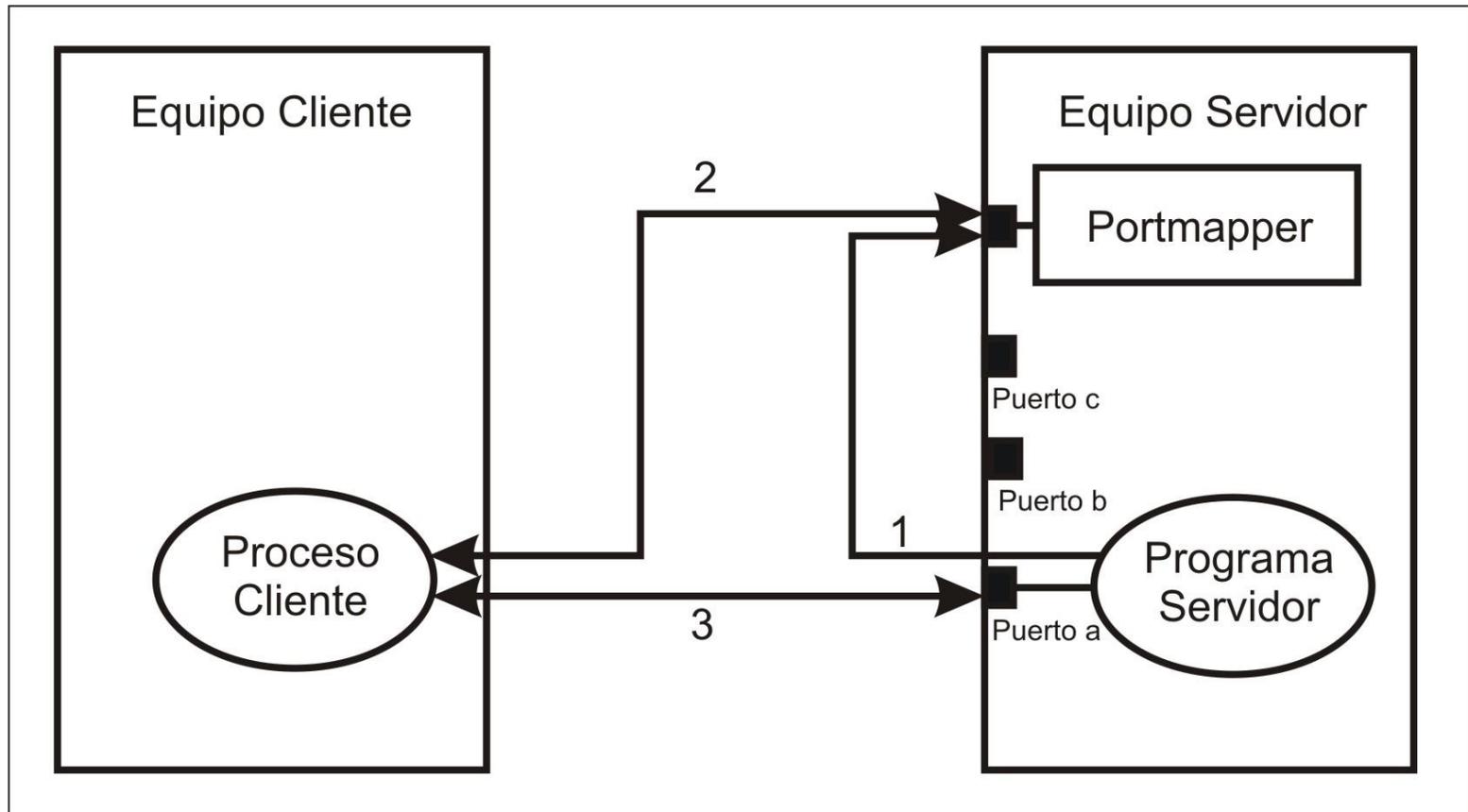


# GESTIÓN DE SERVICIOS: PORTMAPPER

- **Portmapper**: proceso que identifica y localiza procedimientos remotos ofrecidos por un servidor
- Es responsable de las tareas de registro y binding
- Inicialmente, cada servidor registra en el portmapper los procedimientos remotos que exporta.
- **rpcbind** realiza el mapeo (binding) de servicios a puertos.
- El portmapper queda a la escucha (en el puerto 111) y redirecciona las peticiones de acceso a procedimientos remotos hacia sus respectivos puertos locales de escucha.
- Los clientes verifican cada dirección a través del portmapper y luego acceden al servicio.

# GESTIÓN DE SERVICIOS: PORTMAPPER

1. Registro y binding de procedimientos (por parte del servidor)
2. Consulta de servicios y obtención de puertos (por parte del cliente)
3. Acceso a los procedimientos remotos



# RPC: EJEMPLO

- Ejemplo simple de aplicación RPC: servicio para calcular perímetro y área de un cuadrado

## 1. Definir el protocolo (XDR)

- Archivo xdr.x en el cual se definen los métodos que ofrece el servidor.

```
/* Programa perímetro/área del cuadrado */
/* cuadro.x */
```

```
program CUADRADO_PROG {
    version CUADRADO_VERS{
        int PERIMETRO(int) = 1;
        double AREA(int) = 2;
    } = 1;
} = 0x31111111;
```

versión del  
servidor

procedimiento  
remoto #1

procedimiento  
remoto #2

program number, de acuerdo a una  
especificación del protocolo (entre 20000000  
y 3FFFFFFF, definidos por el usuario)

# RPC: perímetro y área de un cuadrado

- Implementación, compilación y ejecución:
  2. Generar stubs: `rpcgen xdr.x`
    - Genera varios archivos:
      - `xdr_clnt.c`: código del stub del cliente. Funciones C a llamar desde las aplicaciones clientes para realizar la llamada remota (la comunicación es transparente).
      - `xdr_svc.c`: código del skeleton del servidor. Incluye un procedimiento `main()` que registra el procedimiento remoto en el port mapper. Realiza las llamadas "reales" a las implementaciones de los procedimientos remotos.
      - `xdr_xdr.c`: código para aplanar/desaplanar (marshalling) los tipos XDR definidos. Sólo se genera si se definen tipos de datos complejos.
      - `xdr.h`: define tipos, constantes, funciones generales, etc. a ser incluido en cliente y servidor.

# RPC: perímetro y área de un cuadrado

- Implementación, compilación y ejecución:
  1. Generar stubs: `rpcgen xdr.x`
    - Opciones de `rpcgen`
    - `rpcgen -N`: permite procedimientos con dos o más parámetros.
    - `rpcgen -Ss`: genera una implementación vacía de los procedimientos remotos (útil como punto de partida para escribir el servidor).
    - `rpcgen -Sc`: genera código para un cliente por defecto (útil como punto de partida para escribir los clientes).
    - `rpcgen -C -a`: genera código para todos los archivos.

# RPC: perímetro y área de un cuadrado

- Diseñar el servidor
  - Implementa las funciones que ofrece el servicio.
  - Su versión local sería:

```
#include "cuad_local.h"
int perimetro(int a){
    int res;
    res = 4*a;
    return(res);
}

double area(int a){
    double res;
    res = a*a;
    return(res);
}
```

```
/* Archivo cuad_local.h */
int perimetro(int a);
double area(int a);
```

# RPC: perímetro y área de un cuadrado

- Diseñar el cliente
  - Invoca los procedimientos que ofrece el servidor.
  - Su versión local sería:

```
#include <stdio.h>
#include "cuad_local.h"
main(int argc, char *argv[]){
    int a, per;
    double sup;
    if (argc !=2 ) {
        fprintf(stderr,"Error, uso: %s a \n",argv[0]);
        exit(1);
    }
    a = atoi(argv[1]);
    per = perimetro(a);
    sup = area(a);
    printf("El perímetro del cuadrado es %d \n",per);
    printf("El área del cuadrado es %d \n",sup);
}
```

# RPC: USANDO los ARCHIVOS GENERADOS

- Implementar el servidor
- Servidor generado automáticamente por rcpngen

```
#include "cuadro.h"

int *perimetro_1_svc(int *argp, struct svc_req *rqstp){
    static int result;
    result = 4* (*argp);
    return &result;
}

double *area_1_svc(int *argp, struct svc_req *rqstp){
    static double result;
    result = (*argp) * (*argp);
    return &result;
}
```

incorporar  
definición de  
funciones

# RPC: USANDO los ARCHIVOS GENERADOS

- Implementar el cliente
- Cliente generado automáticamente por rpcgen

```
#include "cuadro.h"
int cuadrado_prog_1(char *host, int lado){
```

Incluir parámetro

```
CLIENT *clnt;
char *host;
```

```
int a, int *per;
double *sup;
```

```
perimetro_1_arg = lado;
area_1_arg = lado;
```

Incluir parámetro

```
if (argc != 3 ) {
    fprintf(stderr, "Error, uso: %s host a \n", argv[0]);
    exit(1);
}
```

```
a = atoi(argv[1]);
```

```
clnt = clnt_create (host, CUADRO_PROG, CUADRO_VERS, "udp");
if (clnt == NULL) {
    clnt_pcreateerror (host);
    exit (1);
}
```

establecer la conexión remota (no hay que incluirlo, lo genera rpcgen)

# RPC: ARCHIVOS GENERADOS

- Implementar el cliente
- Cliente generado automáticamente por rpcgen (continuación)

```
per = perimetro_1(&a, clnt);  
if (per == (int *) NULL) {  
    clnt_perror (clnt, "call failed");  
}  
sup = area_1(&a, clnt);  
if (sup == (double *) NULL) {  
    clnt_perror (clnt, "call failed");  
}  
  
printf("El perímetro del cuadrado es %d \n", *per);  
printf("El área del cuadrado es %d \n", *sup);  
clnt_destroy(clnt);  
}
```

invocar  
procedimientos  
remotos

cerrar la  
conexión

# RPC: ARCHIVOS GENERADOS

- Implementar el cliente
- Cliente generado automáticamente por rpcgen (continuación)

```
int
main (int argc, char *argv[])
{
    char *host;
    int par;

    if (argc < 2) {
        printf ("usage: %s server_host lado\n", argv[0]);
        exit (1);
    }

    host = argv[1];
    par = atoi(argv[2]);
    cuadrado_prog_1(host,par);

    exit (0);
}
```

determinar parámetros  
y usarlos para invocar el  
procedimiento



# RPC: perímetro y área de un cuadrado

- Implementación, compilación y ejecución:
  1. Definir protocolo
  2. Generar stubs
  3. Definir cliente y servidor
  4. Compilar cliente y servidor:
    - `gcc servidor.c xdr_svc.c [xdr_xdr.c] -o servidor -lnsl`
    - `gcc cliente.c xdr_clnt.c [xdr_xdr.c] -o cliente -lnsl`

# OTROS MECANISMOS de RPC

- Invocación a procedimientos remotos
  - Java Remote Procedure Interface (RMI)
  - Windows Communication Foundation (WCF), .NET Remoting
- Web services
  - Simple Object Access Protocol (SOAP)
- Middleware
  - CORBA
  - J2EE
  - COM/DCOM

# 5.6: MIDDLEWARE DE COMUNICACIÓN ORIENTADO A MENSAJES

# COLAS DE MENSAJES DISTRIBUIDAS

- En un middleware orientado a mensajes, además de emisor y receptor aparece un tercer componente de la conexión encargado de las tareas de almacenamiento de mensajes.
- El mecanismo de almacenamiento permite al emisor y/o al receptor estar inactivos sin que se pierdan mensajes.
- El modelo permite las comunicaciones **persistentes** y **asincrónicas**.
- Los tiempos de transferencia se incrementan notoriamente.
- Habitualmente implementado mediante **colas de mensajes**.
  - La idea básica consiste en insertar (put) o quitar (take/get) mensajes en una cola ordenada (first-in-first-out, FIFO).
  - Al emisor sólo se le puede garantizar que el mensaje ha sido insertado correctamente en la cola. No existen garantías de cuándo será leído dicho mensaje.
  - No está ligado a ningún tipo específico de modelo de red.

# COLAS DE MENSAJES DISTRIBUIDAS

- API básica de un sistema de colas de mensajes:
  - **put**: añadir un mensaje a una cola de mensajes.
  - **get**: primitiva bloqueante; el proceso invocante se bloquea mientras la cola de mensajes esté vacía, y a su retorno quita el primer elemento.
  - **poll**: primitiva no bloqueante; verifica si hay mensaje en la cola de mensajes, y en tal caso quita el primero. Nunca se bloquea.
  - **notify**: instala en la cola de mensajes un mecanismo que notificará cuando un nuevo mensaje se inserte en la cola.
- Algunas implementaciones de este tipo son:
  - RabbitMQ, Microsoft Message Queue (MSMQ), Java Message Service (JMS), etc.

# RESUMEN

Método	Reseña
Pipe y colas FIFO	Half-duplex stream de bytes
Semáforos	Estructura simple de sincronización
Memoria compartida	Múltiples procesos comparten el mismo segmento de memoria
Cola de mensajes	Full-duplex stream de paquetes
Señales	Mensajes de sistema entre procesos
Socket	Full-duplex stream de paquetes mediante interfaz de red
RPC	Invocación transparente de métodos remotos
Colas de mensajes distribuidas	Comunicación totalmente desacoplada

# REFERENCIAS

- UNIX Network Programming, 2nd Edition, Richard Stevens
- UNIX System Calls and Subroutines using <http://www.cm.cf.ac.uk/Dave/C/CE.html>
- The Linux Programming Interface, Michael Kerrisk <http://man7.org/tlpi/>