



Introducción a Sockets

Comunicación entre procesos mediante paso de mensajes

Prof. Antonio López Jaimes



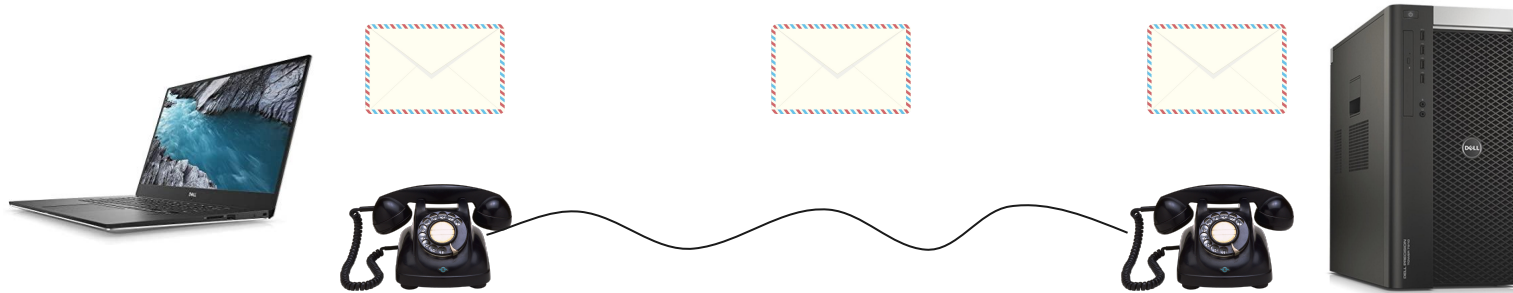
Problema

Comunicar dos procesos localizados en computadoras conectadas por red (ethernet, wifi, etc.)



Solución

De manera abstracta (no importan los detalles técnicos), necesitamos un mecanismo de **paso de mensajes**.





Solución

De manera abstracta (no importan los detalles técnicos), necesitamos un mecanismo de **paso de mensajes**.

Hay varios mecanismo que implementan paso de mensajes:

- Canales
- Puertos
- Sockets



Sockets

Los sockets son una interfaz que provee Unix/Linux para comunicar procesos usando los protocolos TCP/IP.

Al incluir una biblioteca tendremos disponibles funciones para:

- Crear un socket:
 - para comunicarse en un mismo host o hosts diferentes.
 - Para enviar mensajes individuales (datagramas) o un flujo continuo bidireccional (*streams*).
- Configurar el socket.
- Enviar y recibir mensajes.

Interfaz para usar sockets



Creación del socket

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int dominio, int tipo, int protocolo);
```

Crea un extremo del socket regresando su descriptor para usarlo más adelante. Debemos indicar:

1. dominio:
 - a. `AF_UNIX`: para comunicar procesos en el mismo host.
 - b. `AF_INET`: para comunicar procesos conectados por red.
2. tipo:
 - a. `SOCK_STREAM`: flujo de datos confiable y bidireccional.
 - b. `SOCK_DGRAM`: envía mensajes separados y sin mecanismo de confiabilidad.
3. Protocolo: elegir, TCP, UDP, IP. Poner 0 para la opción por omisión.



Configuración del socket (lado del servidor)

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

Asocia al socket `sockfd` la dirección `my_addr` (URL/IP + puerto), proporcionando el tamaño `addrlen` que ocupa la estructura de datos de la dirección.

Enviar y recibir por un socket



```
#include <sys/types.h>
#include <sys/socket.h>

int sendto(int s, const void *msg, int len, unsigned int flags, const struct sockaddr *to, int tolen);
```

Envía por el socket `s` el mensaje `msg` de longitud `len` al host con dirección `to`. Donde `tolen` es el tamaño de la estructura de la dirección. El parámetro `flags` indica el compartamiento del envío.

Enviar y recibir por un socket



```
#include <sys/types.h>
#include <sys/socket.h>

int recvfrom(int s, void *buf, int lon, unsigned int flags, struct sockaddr *desde, int *londesde);
```

Recibe en el espacio `buf` con tamaño máximo `lon`. Al término de la llamada se guardará en `desde` la dirección del host que envió el mensaje y en `londesde` el tamaño de la dirección. El parámetro `flags` indica cómo se debe comportar esta función.

Funciones auxiliares

```
#include <sys/types.h>
#include <sys/socket.h>

unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);

unsigned long int inet_addr(const char *cp);
```

El host y el protocolo de red pueden tener formatos diferentes para representar enteros. Por lo tanto, debemos convertir al formato correcto las direcciones de las computadoras y los puertos.

Máquina

0000 **0010**

Red


0010 0000

`inet_addr` convierte una dirección URL o IP al formato correcto que utiliza el protocolo de red.

Ejemplo básico

Servidor del clima

Servidor: crear y configurar el socket



```
struct sockaddr_in serv;

int main ()
{
    char request[ETHSIZE];
    char *datos_para_el_cliente= "Cloudy with a chance of meatballs";

    puts("Se crea el socket");
    fd = socket(AF_INET, SOCK_DGRAM, 0); // Comunicar por internet, usando datagramas y el protocolo por omisión

    puts("Se asignan atributos al socket");
    memset(&serv, sizeof (serv), 0); // Reservar memoria
    serv.sin_family = AF_INET;
    serv.sin_addr.s_addr = htonl(INADDR_ANY); // Convertir mi dirección al formato de red.
    serv.sin_port = htons(7779); // Convertir el puerto al formato de red.

    idb = bind(fd, (struct sockaddr *) &serv, sizeof (serv));

    ...
}
```

Servidor: recibir una petición y responder

```
...
/* Aquí esperamos la petición del cliente */

cli_len = ETHSIZE;
size_rcv = recvfrom(fd, (void *)request, ETHSIZE, 0,
                    (struct sockaddr *) &cli, (socklen_t *) &cli_len);

/* Aquí enviamos los datos al cliente */
sendto(fd, datos_para_el_cliente, strlen(datos_para_el_cliente), 0,
        (struct sockaddr *)&cli, (socklen_t ) cli_len);
}
```

Cliente: crear el socket

/* sintaxis: \$> cliente <direccion IP> <# puerto> */

struct sockaddr_in serv;

int main (int argc, char *argv[])
{

char *request = "Dame el clima para hoy";

char buffer [ETHSIZE];

puts("abriendo el socket");

fd = socket(AF_INET, SOCK_DGRAM, 0);

...

Cliente: enviar petición y recibir respuesta

```
...
memset(&serv, sizeof (serv), 0);
serv.sin_family = AF_INET;
serv.sin_addr.s_addr = inet_addr(argv[1]);
serv.sin_port = htons(atoi(argv[2]));

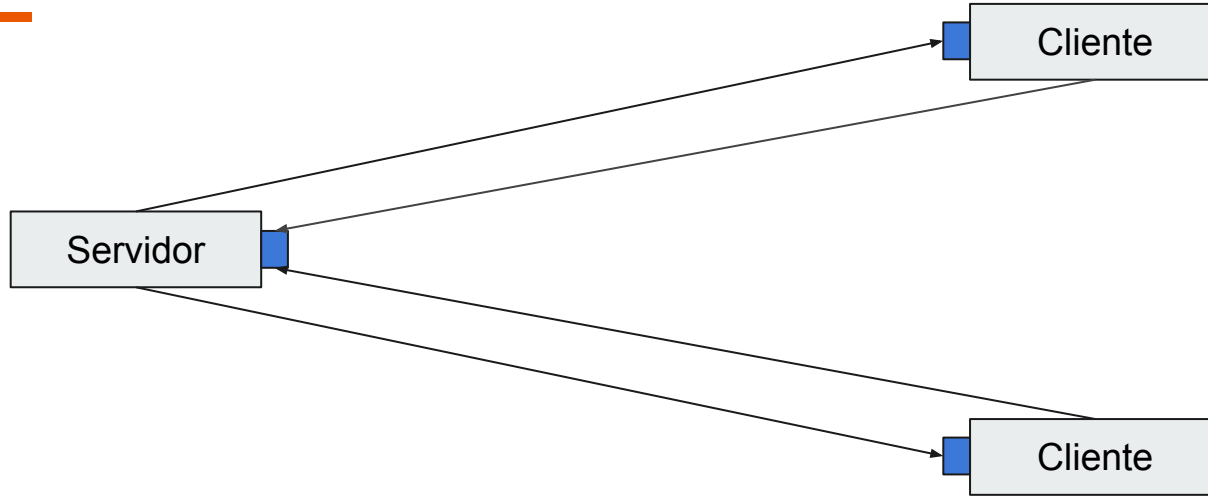
/* Para enviar datos usar sendto() */
result_sendto = sendto(fd, request, 200, 0, (const struct sockaddr *)&serv, sizeof (serv));

/* para recibir datos usar recvfrom */
recvfrom(fd, (void *) buffer, ETHSIZE, 0, (struct sockaddr *) NULL, NULL);

puts("Estos son los datos enviados por el servidor:");
puts(buffer);
}
```


Esquema cliente/servidor

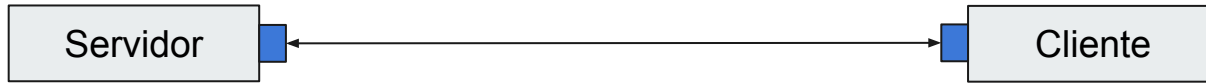
Protocolo con datagramas (sin mantener conexión)



En el protocolo orientado a datagramas o también llamado sin conexión (*datagram-oriented, connectionless*), el servidor tiene un puerto bien conocido para las solicitudes de los clientes y la respuesta se regresa a los puertos específicos para cada cliente.

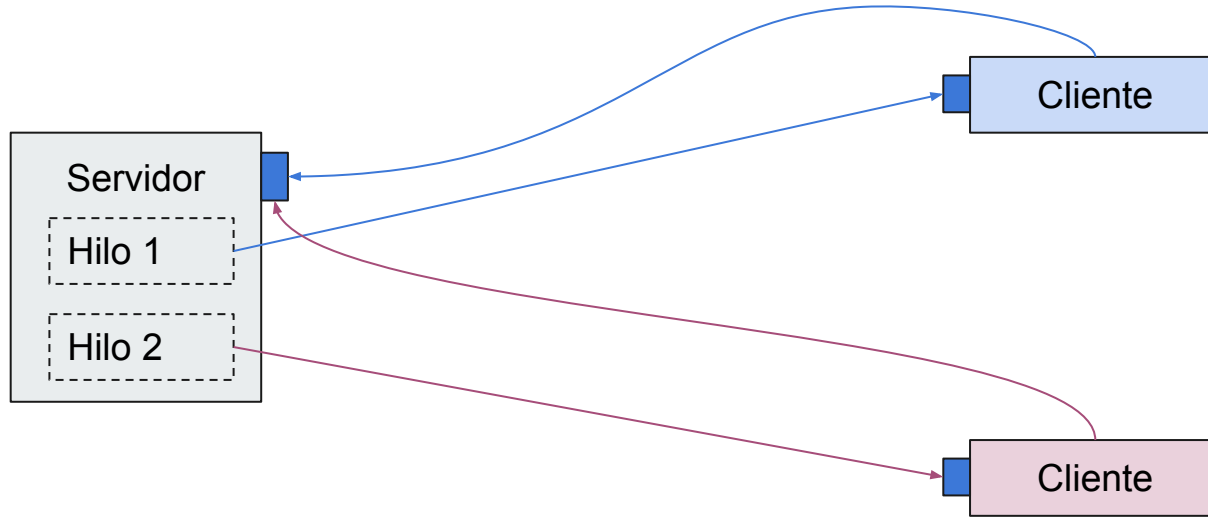
En cada datagrama debemos usar la dirección y puerto.

Protocolo con conexión



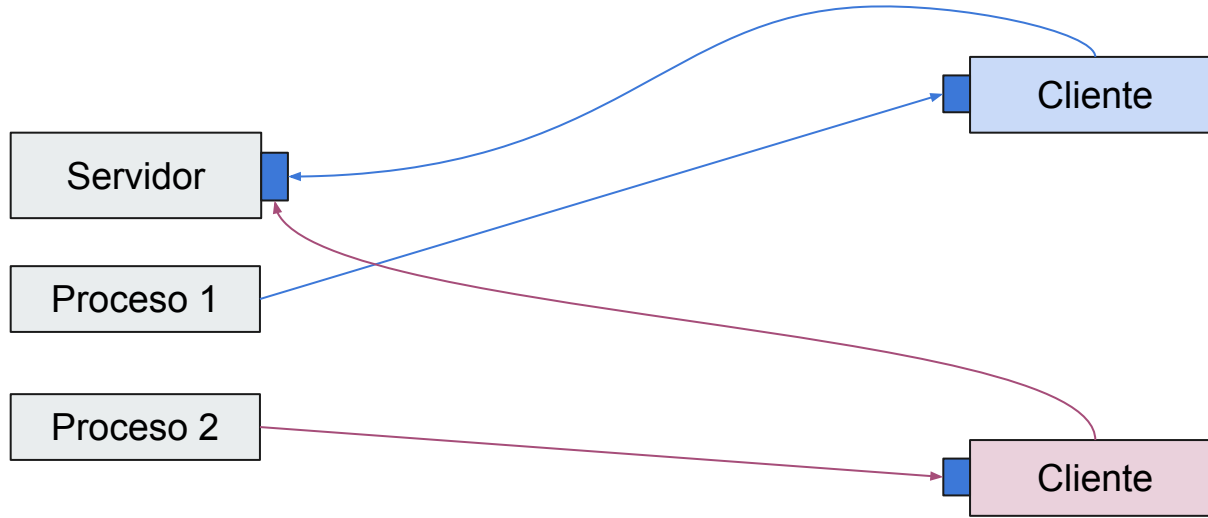
En un protocolo con conexión (*connection-oriented*), el cliente solicita un servicio a un servidor accediendo a un puerto bien conocido. El servidor responde proporcionando un *canal bidireccional privado*. Se usa la misma conexión para cada mensaje.

Procesamiento concurrente de las peticiones



El servidor recibe la petición, pero la procesa y regresa el resultado un **hilo** diferente. Hay variables compartidas.

Procesamiento concurrente de las peticiones



El servidor recibe la petición, pero la procesa y regresa el resultado un **proceso** diferente (usando fork). No hay variables compartidas.