URJC Escuela de Ingeniería de Fuenlabrada

RoboticsLabURJC

Programming Robot Intelligence

Sistemas Distribuidos y Concurrentes

Sockets

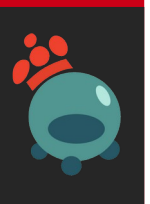
Grado en Ingeniería de Robótica Software

Teoría de la Señal y las Comunicaciones y Sistemas Telemáticos y Computación

Roberto Calvo Palomino
roberto.calvo@urjc.es

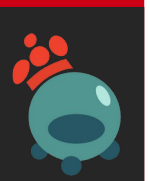
Introducción a Sockets

- Los sockets son el **mecanismo** que nos permite realizar comunicación entre procesos a través de una red.
- Proporciona una **interfaz** entre aplicación y red.
  - Las aplicaciones envían o reciben datos a través de la red usando los sockets.
- En terminología **UNIX**, un socket puede ser entendido como un descriptor de fichero.
- Sockets siempre se comunican en pares y son usados para comunicar procesos entre si (local o remotamente).



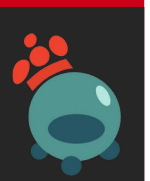
API Socket

- Usaremos el **API** de sockets, programando en **C** para entornos **Linux**.
- El API contiene **estructuras de datos**, **funciones** y **llamadas al sistema** que ayudan a escribir programas eficientes.
- **Requerimientos**: Sistema con Linux y gcc
- Existen **multitud** de APIs de comunicación para distintos sistemas operativos y diferentes lenguajes de programación.



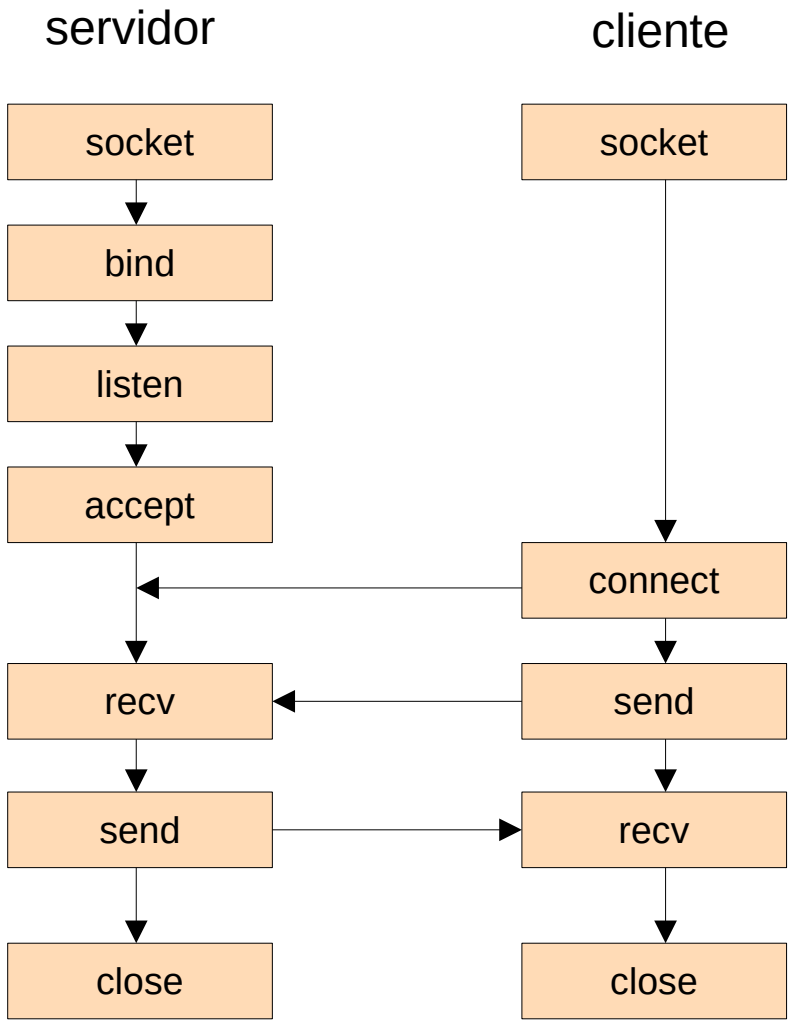
Tipos de sockets

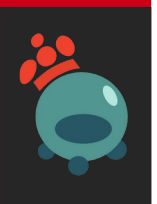
- **SOCK_STREAM**
  - TCP
  - Entrega confiable
  - Orden garantizado
  - Orientado a conexión
  - Bidireccional
- **SOCK_DGRAM**
  - UDP
  - Entrega no segura
  - No se garantiza el orden
  - No es orientado a conexión
  - Bidireccional



SOCK_STREAM (TCP)

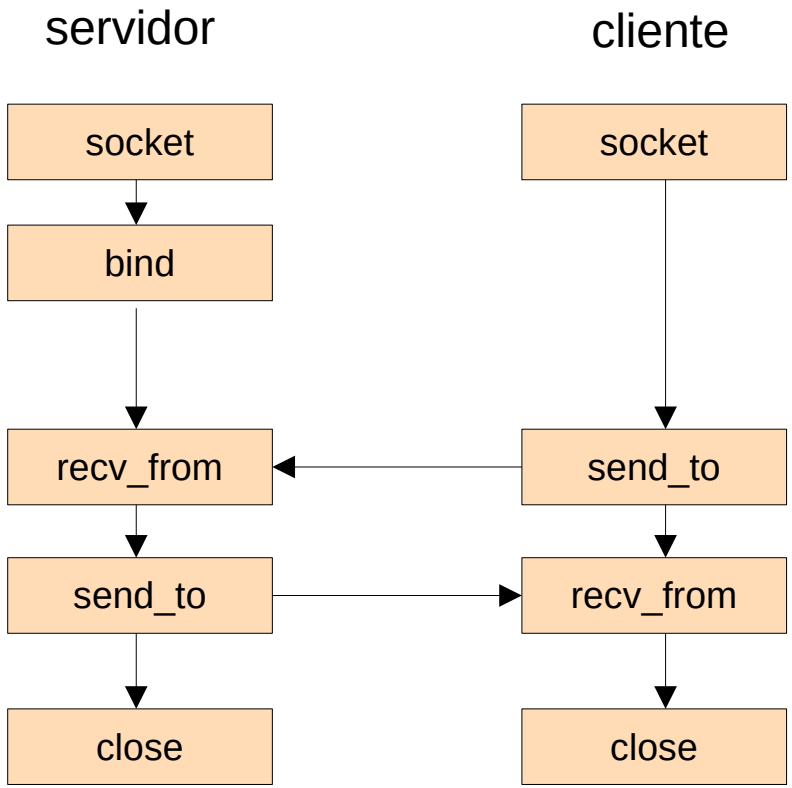
- Se crean sockets en ambos lados de la comunicación
- El servidor prepara el socket y se pone a escuchar (*listen*)
- Cliente genera su socket, y conecta.
- El servidor acepta la conexión y ambos intercambian mensajes.
- Ambos cierran los sockets.

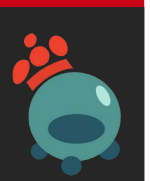




SOCK_DGRAM (UDP)

- Se crean sockets en ambos lados de la comunicación
- El servidor prepara el socket
- Ambos intercambian mensajes.
- Ambos lados cierran los sockets.



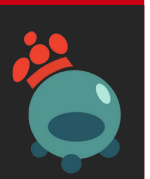


Estructura de datos

- La estructura ***in_addr*** define (en IPv4) la dirección IP representado en 4 bytes.

```
struct in_addr {
    unsigned long s_addr;
};
```

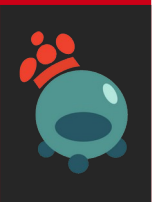
- Big-endian para su representación en la red
- Little-endian para su representación en el host
- htonl, htons, ntohl, ntohs (man)



Estructura de datos

- La estructura ***sockaddr_in*** define detalles de la conexión
  - sin_family: Tipo de familia de socket
  - sin_port: Puerto de la conexión
  - sin_addr: Dirección IP

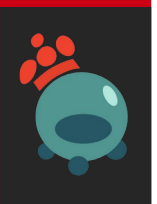
```
struct sockaddr_in {
    short           sin_family;
    unsigned short  sin_port;
    struct in_addr  sin_addr;
    char            sin_zero[8];
};
```



Estructura de datos

- *INADDR_ANY* es usado para escuchar en cualquier interfaz del servidor (lo, ethX, wlanX, ...)

```
#define PORT 8080
struct sockaddr_in servaddr;

servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(PORT);
```



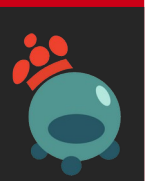
Crear un socket

- Primitiva para crear un canal de comunicación (man socket)

```
int socket(int domain, int type, int protocol);
```

  - domain: especifica el dominio de la comunicación (man socket)
  - type: especifica la semántica de la comunicación (TCP, UDP, ...)
  - protocolo: tipo de protocolo dentro del tipo (normalmente 0)

```
tcp_socket = socket(AF_INET, SOCK_STREAM, 0);
udp_socket = socket(AF_INET, SOCK_DGRAM, 0);
```



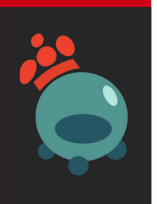
bind

- Una vez creado el socket, la llamaba ***bind*** asigna un dirección especifica (sockaddr_in) al socket.

```
int bind(int sockfd, const struct sockaddr *addr,
         socklen_t addrlen);
```

  - sockfd: descriptor asignado al socket
  - addr: estructura con la información de la IP y puerto
  - addrlen: longitud de la estructura anterior

```
res = bind( tcp_socket,
            (struct sockaddr *) &server_addr,
            sizeof(server_addr)) == -1 );
```



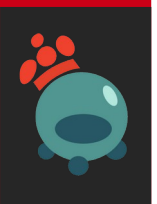
listen

- La función ***listen*** convierte un socket defindo por un descriptor, en un *socket pasivo*, y puede ser utilizado para aceptar conexiones entrantes.

```
int listen(int sockfd, int backlog);
```

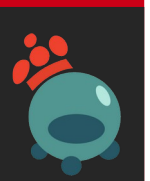
  - sockfd: descriptor asignado al socket
  - backlog: máximo tamaño de conexiones encoladas por el kernel.

```
int res = listen(sockfd, 5)
```



listen

- diagrama



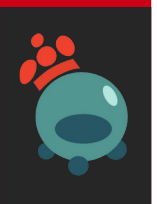
accept

- La función ***accept*** es usada con sockets orientados a conexión. Extrae la primera petición de conexión de la cola de conexiones pendientes, y crea un nuevo socket para el cliente.

```
int accept(int sockfd, struct sockaddr *addr,
           socklen_t *addrlen);
```

  - sockfd: descriptor asignado al socket
  - addr: estructura con la información de la IP y puerto

```
struct sockaddr_in sock_cli;
socket_t len = sizeof(sock_cli);
conn_fd = accept(sockfd,(struct sockaddr *)&sock_cli, &len);
```



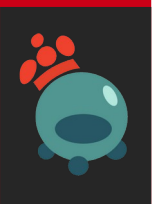
close

- La función ***close*** marca el socket como cerrado. El descriptor no puede ser usado por el proceso.

```
int close (int sockfd)
```

  - sockfd: descriptor asignado al socket

```
int r = close (sockdf);
```



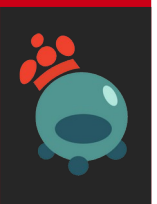
recv

- La función ***recv*** recibe datos de un socket.

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

  - sockfd: descriptor asignado al socket
  - buf: buffer donde se guarda la información que se recibe.
  - len: longitud (máxima) de la estructura anterior
  - flags: extiende funcionalidad (ej. bloqueante, no-bloqueante)

```
int r = recv(sockfd, (void*) buff, sizeof(buff), 0);
```

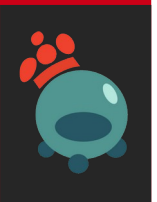


recvfrom

- La función ***recv*** recibe datos de un socket con una dirección en concreto.

```
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
```

  - sockfd: descriptor asignado al socket
  - buf: buffer donde se guarda la información que se recive.
  - len: longitud (máxima) de la estructura anterior
  - flags: extiende funcionalidad (ej. bloqueante, no-bloqueante)
  - src_addr: estructura con la información de la dirección origen
  - addrlen: longitud de la estructura anterior



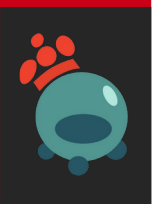
send

- La función ***send*** envía datos por un socket

```
ssize_t send(int sockfd, void *buf, size_t len, int flags);
```

  - sockfd: descriptor asignado al socket
  - buf: buffer donde se guarda la información que se envía
  - len: longitud (máxima) de la estructura anterior
  - flags: extiende funcionalidad (ej. bloqueante, no-bloqueante)

```
int r = send(sockfd, buff, sizeof(buff), 0);
```

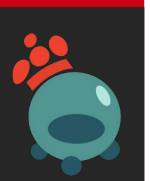


sendto

- La función ***sendto*** manda datos por un socket a un dirección en concreto.

```
ssize_t sendto(int sockfd, void *buf, size_t len, int flags,
                    struct sockaddr *dest_addr, socklen_t *addrlen);
```

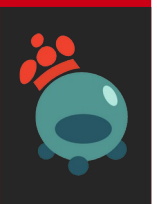
  - sockfd: descriptor asignado al socket
  - buf: buffer donde se guarda la información que se envía.
  - len: longitud (máxima) de la estructura anterior
  - flags: extiende funcionalidad (ej. bloqueante, no-bloqueante)
  - dest_addr: estructura con la información de la dirección destino
  - addrlen: longitud de la estructura anterior



Bloqueante / No Bloqueante

- Las llamadas *recv()* es bloqueante. El proceso se queda esperando indefinidamente hasta que el socket tiene información que leer.
- Una posible solución es utilizar ***MSG_DONTWAIT*** como flag
  - Si no hay datos disponibles, no bloquea retornando el error EAGAIN

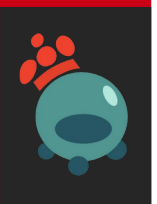
```
int r = recv(sockfd, (void*) buff, sizeof(buff), MSG_DONTWAIT);
```



Bloqueante / No Bloqueante

- La llamada ***select()*** monitoriza descriptores de ficheros hasta que están listos para realizar ciertas operaciones de I/O

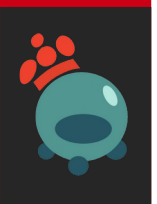
```
#include <sys/select.h>
#include <unistd.h>

fd_set readmask;
struct timeval timeout;

FD_ZERO(&readmask);                          // Reset la mascara
FD_SET(sockfd, &readmask);                 // Asignamos el nuevo descriptor
FD_SET(STDIN_FILENO, &readmask);    // Entrada
timeout.tv_sec=0; timeout.tv_usec=100000; // Timeout de 0.1 seg.
if (select(sockfd+1, &readmask, NULL, NULL, &timeout)==-1)
    exit(-1);

if (FD_ISSET(sockfd, &readmask)){
   // Hay datos que leer del descriptor
}
```



Manejando múltiples conexiones

- El servidor puede necesitar procesar múltiples conexiones a la vez.
- Inicializamos el servidor normalmente
  - *bind(), listen(), accept()*
- Una vez aceptada la conexión crearemos un thread para atender a cada cliente.
- Cada nuevo thread creado será el responsable de interactuar y comunicarse con ese cliente.

```
conn_fd = accept(sockfd,(struct sockaddr *)&sock_cli, &len);

pthread_t thread;
pthread_create(&thread, NULL, &thread_client, &conn_fd);
```



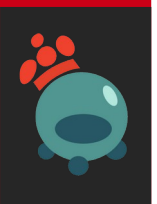
Reusando puertos

- Cuando cerráis un socket, normalmente el sistema operativo tarde un tiempo en dejarlo libre para reutilizarlo.

```
tcp        0      0 10.1.148.190:44520      142.250.185.10:443      ESTABLISHED 5283/firefox
tcp        0      0 10.1.148.190:52262      37.139.1.159:443        TIME_WAIT   -
```

- Podéis forzar la reutilización casi instantánea con la siguiente confirmación del socket.
- Debe realizarse antes del bind().

```
const int enable = 1;

if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &enable, sizeof(int)) < 0)
    error("setsockopt(SO_REUSEADDR) failed");
```

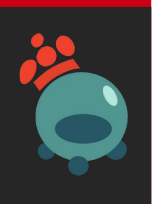


Eliminar buffering

- Para deshabilitar el buffering a la hora de imprimir mensajes, asegurate que SIEMPRE llamar a la siguiente función en las primeras líneas de tu main()

```
setbuf(stdout, NULL);
```

- ¡Añádelo siempre en tu código de prácticas!



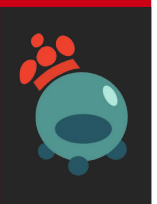
Pthread: librería hilos POSIX Linux

- Para escribir programas multihilo en C podemos utilizar la libreriía pthread que implementa el standard POSIX (Portable Operating System Interface)

```
#include <pthread.h>
```

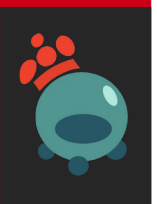
```
int pthread_create(pthread_t *restrict thread,
                   const pthread_attr_t *restrict attr,
                   void *(*start_routine)(void *),
                   void *restrict arg);
```

https://man7.org/linux/man-pages/man3/pthread_create.3.html



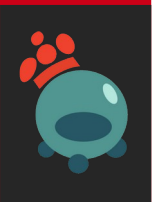
Pthread: librería hilos POSIX Linux

- **pthread_create**:
  - Crea el thread y lo pone en ejecución.
- **pthread_exit**:
  - Finaliza el thread, debe ejecutarse dentro de propio thread.
  - Puede pasar parámetros opcionales que se recogen en pthread_join()
- **pthread_join**:
  - Realiza una espera hasta que un thread determinado ha finalizado.
- Esta librería no pertenece a la biblioteca estándar de C, por lo que para compilar habrá que añadir siempre la referencia a la librería con “-lpthread”



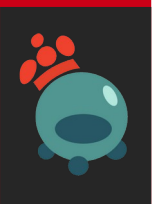
Ejemplo de Pthreads

```c
#include <stdio.h>
#include <string.h>
#include <pthread.h>

void *thread_function (void *arg) {
    int var = *(int *)(arg);
    printf( format: "Ejecutando thread_function() con valor %d\n", var);
}

int main (int argc, char* argv[]) {
    pthread_t h1;
    pthread_t h2;

    int param = 10;
    pthread_create (&h1, attr: NULL, thread_function, (void *) &param);

    int param2 = 20;
    pthread_create (&h2, attr: NULL, thread_function, (void *) &param2);

    pthread_join(h1, thread_return: NULL);
    pthread_join(h2, thread_return: NULL);

    printf( format: "End\n");
}
```



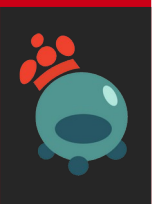
Pthreads

- El uso de threads implica en la mayoría de las situaciones tener condiciones de carrera.
- El paso de parámetros a threads:
  - Todos los argumentos deben ser pasados por referencia y realizado el cast a (void*)
  - Para múltiples parámetros:
    - Crear una estructura que contenga todos los parámetros
    - Pasar la referencia de la estructura al thread.



Bibliografía

- Hands-On Network Programming with C (Lewis Van Winkle)
- Advanced Programming in the UNIX Environment, 3rd Edition 3rd Edition (W. Stevens, Stephen Rago)
- Unix Network Programming, Volume 1: The Sockets Networking API (W. Stevens, Bill Fenner, Andrew Rudoff)
- Beej's Guide to Network Programming http://beej.us/guide/bgnet/



URJC Escuela de Ingeniería de Fuenlabrada

RoboticsLabURJC

Programming Robot Intelligence