# Práctica 2.5: Sockets

### Objetivos

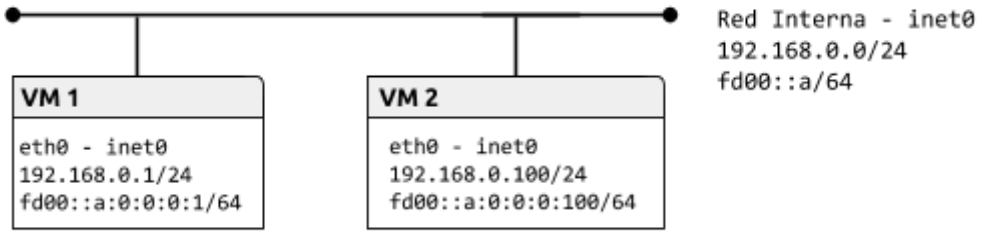
En esta práctica, nos familiarizaremos con la interfaz de programación de sockets como base para la programación de aplicaciones basadas en red, poniendo de manifiesto las diferencias de programación entre los protocolos UDP y TCP. Además, aprenderemos a programar aplicaciones independientes de la familia de protocolos de red (IPv4 o IPv6) utilizados.

### Contenidos

Preparación del entorno de la práctica
Gestión de direcciones
Protocolo UDP - Servidor de hora
Protocolo TCP - Servidor de eco

## Preparación del entorno de la práctica

Configuraremos la topología de red que se muestra en la figura. Como en prácticas anteriores construiremos la topología con la herramienta `vtopol`. Antes de comenzar la práctica, configurar los interfaces de red como se indica en la figura y comprobar la conectividad entre las máquinas.




***Nota:*** Observar que las VMs tienen un interfaz de red con pila dual IPv6 - IPv4.

## Gestión de direcciones

El uso del API BSD requiere la manipulación de direcciones de red, y traducción de estas entre las tres representaciones básicas: nombre de dominio, dirección IP (versión 4 y 6) y binario (para incluirla en la cabecera del datagrama IP).

***Ejercicio 1.*** Escribir un programa que obtenga todas las posibles direcciones con las que se podría crear un socket asociado a un host dado como primer argumento del programa. Para cada dirección, mostrar la IP numérica, la familia de protocolos y tipo de socket. Comprobar el resultado para:

- Una dirección IPv4 válida (ej. "`147.96.1.9`").
- Una dirección IPv6 válida (ej. "`fd00::a:0:0:0:1`").
- Un nombre de dominio válido (ej. "`www.google.com`").
- Un nombre en `/etc/hosts` válido (ej. "`localhost`").
- Una dirección o nombre incorrectos en cualquiera de los casos anteriores.

El programa se implementará usando `getaddrinfo(3)` para obtener la lista de posibles direcciones de socket (**`struct`** `sockaddr`). Cada dirección se imprimirá en su valor numérico, usando `getnameinfo(3)` con el *flag* `NI_NUMERICHOST`, así como la familia de direcciones y el tipo de socket.

***Nota:*** Para probar el comportamiento con DNS, realizar este ejercicio en la máquina física.

Ejemplos:

```
# Las familias 2 y 10 son AF_INET y AF_INET6, respectivamente (ver socket.h)
# Los tipos 1, 2, 3 son SOCK_STREAM, SOCK_DGRAM y SOCK_RAW, respectivamente
> ./gai www.google.com
66.102.1.147 2      1
66.102.1.147 2      2
66.102.1.147 2      3
2a00:1450:400c:c06::67    10     1
2a00:1450:400c:c06::67    10     2
2a00:1450:400c:c06::67    10     3
> ./gai localhost
::1    10     1
::1    10     2
::1    10     3
127.0.0.1    2      1
127.0.0.1    2      2
127.0.0.1    2      3
> ./gai ::1
::1    10     1
::1    10     2
::1    10     3
> ./gai 1::3::4
Error getaddrinfo(): Name or service not known
> ./gai noexiste.ucm.es
Error getaddrinfo(): Name or service not known
```

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv){
        if(argc < 2){
                printf("Faltan argumentos\n");
                exit(1);
        }
        struct addrinfo hints;
        struct addrinfo* res;
        memset(&hints, 0, sizeof(struct addrinfo));
        hints.ai_family = AF_UNSPEC;
        if(getaddrinfo(argv[1], NULL, &hints, &res) != 0){
                perror("getaddrinfo");
                exit(1);
        }
        char host[NI_MAXHOST];
        char serv[NI_MAXSERV];
        for (struct addrinfo* ptr = res; ptr != NULL; ptr = ptr->ai_next){
                if(getnameinfo(ptr->ai_addr, ptr->ai_addrlen, host, NI_MAXHOST, NULL,
```

```
0, NI_NUMERICHOST) != 0){
                        perror("getnameinfo");
                        exit(1);
                }
                printf("%s %i %i \n", host, ptr->ai_family, ptr->ai_socktype);
        }
        return 0;
}

Salida para  ./sockets ::1
::1 10 1
::1 10 2
::1 10 3
```

## Protocolo UDP - Servidor de hora

***Ejercicio 2.*** Escribir un servidor UDP de hora de forma que:

- La dirección y el puerto son el primer y segundo argumento del programa. Las direcciones pueden expresarse en cualquier formato (nombre de host, notación de punto…). Además, el servidor debe funcionar con direcciones IPv4 e IPv6 .
- El servidor recibirá un comando (codificado en un carácter), de forma que `'t'` devuelva la hora, `'d'` devuelve la fecha y `'q'` termina el proceso servidor.
- En cada mensaje el servidor debe imprimir el nombre y puerto del cliente, usar `getnameinfo(3)`.

Probar el funcionamiento del servidor con la herramienta Netcat (comando `nc` o `ncat`) como cliente.

***Nota:*** Dado que el servidor puede funcionar con direcciones IPv4 e IPv6, hay que usar **`struct`** `sockaddr_storage` para acomodar cualquiera de ellas, por ejemplo, en `recvfrom(2)`.

Ejemplo:

| **Servidor** | **Cliente** |
|---|---|
| `$ ./time_server :: 3000`<br>`2 bytes de ::FFFF:192.168.0.100:58772`<br>`2 bytes de ::FFFF:192.168.0.100:58772`<br>`2 bytes de ::FFFF:192.168.0.100:58772`<br>`Comando no soportado X`<br>`2 bytes de ::FFFF:192.168.0.100:58772`<br>`Saliendo...` | `$ nc -u 192.168.0.1 3000`<br>`t`<br>`10:30:08 PMd`<br>`2014-01-14X`<br>`q`<br>`^C`<br>`$` |

***Nota:*** El servidor no envía `'\n'`, por lo que se muestra la respuesta y el siguiente comando (en negrita en el ejemplo) en la misma línea.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#include <time.h>
#include <unistd.h>

int main(int argc, char** argv){
	if(argc < 3){
		printf("Faltan argumentos\n");
		exit(1);
	}
	struct addrinfo hints;
	struct addrinfo* res;
	memset(&hints, 0, sizeof(struct addrinfo));
	hints.ai_family = AF_UNSPEC;
	hints.ai_flags = AI_PASSIVE;
	hints.ai_socktype = SOCK_DGRAM;
	if(getaddrinfo(argv[1], argv[2], &hints, &res) != 0){
		perror("getaddrinfo");
		exit(1);
	}
	int sd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

	if(bind(sd, res->ai_addr, res->ai_addrlen) == -1){
		perror("bind");
		exit(1);
	}
	char car [2];
	struct sockaddr cliente;
	socklen_t longitudCliente;
	char s[256];
	char host[NI_MAXHOST];
	char serv[NI_MAXSERV];
	while(1){
		recvfrom(sd, &car, 2, 0, &cliente, &longitudCliente);
		car[1] = '\0';
		getnameinfo(&cliente, longitudCliente, host, NI_MAXHOST, serv, NI_MAXSERV,
NI_NUMERICHOST | NI_NUMERICSERV);
		printf("%li bytes de %s:%s\n", strlen(car), host, serv);
		if(strcmp(car, "t") == 0){
			time_t tiempo = time(NULL);
			struct tm * hora = localtime(&tiempo);
			strftime(s, 256, "%T", hora);
			sendto(sd, s, strlen(s) + 1, 0, &cliente, longitudCliente);
		}
		else if(strcmp(car, "d") == 0){
			time_t tiempo = time(NULL);
			struct tm * hora = localtime(&tiempo);
			strftime(s, 256, "%F", hora);
			sendto(sd, s, strlen(s) + 1, 0, &cliente, longitudCliente);
		}
		else if(strcmp(car, "q") == 0) break;
	}
	close(sd);
	return 0;
}
```

Servidor:

```
./sockets :: 3000
```

```
1 bytes de ::ffff:192.168.0.100:59488
1 bytes de ::ffff:192.168.0.100:59488
1 bytes de ::ffff:192.168.0.100:59488
1 bytes de ::ffff:192.168.0.100:59488
1 bytes de ::ffff:192.168.0.100:59488
1 bytes de ::ffff:192.168.0.100:59488


Cliente:

nc -u 192.168.0.1 3000
t
22:43:54t
22:43:57t
22:43:58d
2021-01-19d
2021-01-19q
```

***Ejercicio 3***. Escribir el cliente para el servidor de hora. El cliente recibirá como argumentos la dirección del servidor, el puerto del servidor y el comando. Por ejemplo, para solicitar la hora, `./time_client 192.168.0.1 3000 t`.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

int main(int argc, char** argv){
	if(argc < 4){
		printf("Faltan argumentos\n");
		exit(1);
	}
	struct addrinfo hints;
	struct addrinfo* res;
	memset(&hints, 0, sizeof(struct addrinfo));
	hints.ai_family = AF_UNSPEC;
	hints.ai_socktype = SOCK_DGRAM;
	if(getaddrinfo(argv[1], argv[2], &hints, &res) != 0){
		perror("getaddrinfo");
		exit(1);
	}
	int sd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
	char buff[256];

	sendto(sd, argv[3], strlen(argv[3]), 0, res->ai_addr, res->ai_addrlen);
	recvfrom(sd, buff, 256, 0, NULL, NULL);
	printf("%s", buff);
	close(sd);
	return 0;
```

```
}
Cliente:
gcc sockets.c -o sockets
usuarioso@ssoo:~$ ./sockets 192.168.0.1 3000 t
23:01:25usuarioso@./sockets 192.168.0.1 3000 q

Servidor:

./sockets :: 3000
1 bytes de ::ffff:192.168.0.100:40545
1 bytes de ::ffff:192.168.0.100:48964
```

***Ejercicio 4.*** Modificar el servidor para que, además de poder recibir comandos por red, los pueda recibir directamente por el terminal, leyendo dos caracteres (el comando y `'\n'`) de la entrada estándar. Multiplexar el uso de ambos canales usando `select(2)`.

```
 #include <sys/types.h>
 #include <sys/socket.h>
 #include <netdb.h>
 #include <stdio.h>
 #include <stdlib.h>
 #include <unistd.h>
 #include <string.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#include <time.h>


 #define BUF_SIZE 500
#define MAX_OUTPUT_SIZE 50

 int main(int argc, char *argv[])
 {
     struct addrinfo hints;
   struct addrinfo *result, *rp;
   int sfd, s;
   struct sockaddr_storage peer_addr;
   socklen_t peer_addr_len;
   ssize_t nread;
   char buf[2];

    if (argc < 3) {
       fprintf(stderr, "Error. Faltan argumentos\n");
       exit(EXIT_FAILURE);
    }

    /* Obtain address(es) matching host/port */

    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_UNSPEC;    /* Allow IPv4 or IPv6 */
    hints.ai_socktype = SOCK_DGRAM; /* Datagram socket */
```

```
    hints.ai_flags = 0;
    hints.ai_protocol = 0;          /* Any protocol */

     s = getaddrinfo(argv[1], argv[2], &hints, &result);
    if (s != 0) {
       fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
       exit(EXIT_FAILURE);
    }

    /* getaddrinfo() returns a list of address structures.
      Try each address until we successfully connect(2).
      If socket(2) (or connect(2)) fails, we (close the socket
      and) try the next address. */

    for (rp = result; rp != NULL; rp = rp->ai_next) {
       sfd = socket(rp->ai_family, rp->ai_socktype,
                rp->ai_protocol);
       if (sfd == -1)
          continue;

       if (bind(sfd, rp->ai_addr, rp->ai_addrlen) != -1)
          break;                  /* Success */

       close(sfd);
    }

    if (rp == NULL) {               /* No address succeeded */
       fprintf(stderr, "Could not connect\n");
       exit(EXIT_FAILURE);
    }

    freeaddrinfo(result);           /* No longer needed */

    /* Send remaining command-line arguments as separate
      datagrams, and read responses from server */
/* Send remaining command-line arguments as separate
      datagrams, and read responses from server */
     char car [2];
     struct sockaddr_storage cliente;
     socklen_t longitudCliente = = sizeof(struct sockaddr_storage);
     char ss[256];
     char host[NI_MAXHOST];
     char serv[NI_MAXSERV];
/*
    while(1){
      fd_set set;
      FD_ZERO(&set);
      FD_SET(sfd, &set);
      FD_SET(0, &set);
      select(sfd+1, &set, NULL, NULL, NULL);

      if(FD_ISSET(sfd, &set)){


        recvfrom(sfd, &car, 2, 0, (struct sockaddr *)& cliente, &longitudCliente);
        car[1] = '\0';
        getnameinfo((struct sockaddr *)&cliente, longitudCliente, host, NI_MAXHOST, serv,
```

```
NI_MAXSERV, NI_NUMERICHOST | NI_NUMERICSERV);
        printf("%li bytes de %s:%s\n", strlen(car), host, serv);
        if(strcmp(car, "t") == 0){
          time_t tiempo = time(NULL);
          struct tm * hora = localtime(&tiempo);
          strftime(ss, 256, "%T", hora);
          sendto(sfd, ss, strlen(ss) + 1, 0, (struct sockaddr *)&cliente, longitudCliente);
        }
        else if(strcmp(car, "d") == 0){
          time_t tiempo = time(NULL);
          struct tm * hora = localtime(&tiempo);
          strftime(ss, 256, "%F", hora);
          sendto(sfd, ss, strlen(ss) + 1, 0, (struct sockaddr *)&cliente, longitudCliente);
        }
        else if(strcmp(car, "q") == 0) break;

       }

       else if(FD_ISSET(0, &set)){
         read(0, car, 2);
         car[1] = '\0';
         if(strcmp(car, "t") == 0){
           time_t tiempo = time(NULL);
           struct tm * hora = localtime(&tiempo);
           strftime(ss, 256, "%T", hora);
           printf("%s", ss);
         }
         else if(strcmp(car, "d") == 0){
           time_t tiempo = time(NULL);
           struct tm * hora = localtime(&tiempo);
           strftime(ss, 256, "%F", hora);
           printf("%s", ss);
         }
         else if(strcmp(car, "q") == 0) break;

       }
    }
  */


  while(1){

    fd_set fset;
    FD_ZERO(&fset);
    FD_SET(0,&fset);
    FD_SET(sfd, &fset);

    s = select(sfd+1, &fset, NULL, NULL, NULL);

     if(FD_ISSET(sfd, &fset)){
        recvfrom(sfd, &car, 2, 0, (struct sockaddr *)& cliente, &longitudCliente);
        car[1] = '\0';
        getnameinfo((struct sockaddr *)&cliente, longitudCliente, host, NI_MAXHOST, serv,
NI_MAXSERV, NI_NUMERICHOST | NI_NUMERICSERV);
        printf("%li bytes de %s:%s\n", strlen(car), host, serv);
        if(strcmp(car, "t") == 0){
          time_t tiempo = time(NULL);
```

```
          struct tm * hora = localtime(&tiempo);
          strftime(ss, 256, "%T", hora);
          sendto(sfd, ss, strlen(ss) + 1, 0, (struct sockaddr *)&cliente, longitudCliente);
        }
        else if(strcmp(car, "d") == 0){
          time_t tiempo = time(NULL);
          struct tm * hora = localtime(&tiempo);
          strftime(ss, 256, "%F", hora);
          sendto(sfd, ss, strlen(ss) + 1, 0, (struct sockaddr *)&cliente, longitudCliente);
        }
        else if(strcmp(car, "q") == 0) break;
    }

    if(FD_ISSET(0,&fset)){
        s = read(0, buf, 2);
        if(s < 0){
          perror("Fallo al leer comando de stdin");
        }
        char now[MAX_OUTPUT_SIZE];
        if(buf[0] == 't'){
            time_t t;
            char now[MAX_OUTPUT_SIZE];
            time(&t);
            struct tm* timedata = localtime(&t);
            strftime(now, MAX_OUTPUT_SIZE, "%T", timedata);
            printf("%s\n",now);
        }
        else if(buf[0] == 'd'){
            time_t t;
            char now[MAX_OUTPUT_SIZE];
            time(&t);
            struct tm* timedata = localtime(&t);
            strftime(now, MAX_OUTPUT_SIZE, "%F", timedata);
            printf("%s\n",now);
        }
        else if(buf[0] == 'q') break;

    }

 }
   exit(EXIT_SUCCESS);
}
```

Salida servidor:

```
./prueba :: 3000
t
13:21:21
d
2021-01-20
d
2021-01-20
1 bytes de ::ffff:192.168.0.100:53350
1 bytes de ::ffff:192.168.0.100:57742


            return 0;
```

```
}
Salida cliente:
./sockets 192.168.0.1 3000 t
13:21:53usuarioso@ssoo:~$ ./sockets 192.168.0.1 3000 t
13:21:56usuarioso@ssoo:~$
```

***Ejercicio 5.*** Convertir el servidor UDP en multi-proceso siguiendo el patrón *pre-fork*. Una vez asociado el socket a la dirección local con `bind(2)`, crear varios procesos que llamen a `recvfrom(2)` de forma que cada uno atenderá un mensaje de forma concurrente. Imprimir el PID del proceso servidor para comprobarlo.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#include <time.h>
#include <sys/wait.h>

#define BUF_SIZE 500
#define MAX_OUTPUT_SIZE 50
#define NUMPROCESOS 20

int main(int argc, char *argv[])
{
    struct addrinfo hints;
  struct addrinfo *result, *rp;
  int sfd, s;
  struct sockaddr_storage peer_addr;
  socklen_t peer_addr_len;
  ssize_t nread;
  char buf[2];

   if (argc < 3) {
      fprintf(stderr, "Error. Faltan argumentos\n");
      exit(EXIT_FAILURE);
   }

   /* Obtain address(es) matching host/port */

   memset(&hints, 0, sizeof(struct addrinfo));
   hints.ai_family = AF_UNSPEC;    /* Allow IPv4 or IPv6 */
   hints.ai_socktype = SOCK_DGRAM; /* Datagram socket */
   hints.ai_flags = 0;
   hints.ai_protocol = 0;          /* Any protocol */
```

```
    s = getaddrinfo(argv[1], argv[2], &hints, &result);
   if (s != 0) {
      fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
      exit(EXIT_FAILURE);
   }

   /* getaddrinfo() returns a list of address structures.
     Try each address until we successfully connect(2).
     If socket(2) (or connect(2)) fails, we (close the socket
     and) try the next address. */

   for (rp = result; rp != NULL; rp = rp->ai_next) {
      sfd = socket(rp->ai_family, rp->ai_socktype,
                rp->ai_protocol);
      if (sfd == -1)
         continue;

      if (bind(sfd, rp->ai_addr, rp->ai_addrlen) != -1)
         break;               /* Success */

      close(sfd);
   }

   if (rp == NULL) {            /* No address succeeded */
      fprintf(stderr, "Could not connect\n");
      exit(EXIT_FAILURE);
   }

   freeaddrinfo(result);         /* No longer needed */

   /* Send remaining command-line arguments as separate
     datagrams, and read responses from server */
/* Send remaining command-line arguments as separate
     datagrams, and read responses from server */
    char car [2];
    struct sockaddr_storage cliente;
    socklen_t longitudCliente = sizeof(struct sockaddr_storage);
    char ss[256];
    char host[NI_MAXHOST];
    char serv[NI_MAXSERV];
/*
   while(1){
     fd_set set;
     FD_ZERO(&set);
     FD_SET(sfd, &set);
     FD_SET(0, &set);
     select(sfd+1, &set, NULL, NULL, NULL);

     if(FD_ISSET(sfd, &set)){


       recvfrom(sfd, &car, 2, 0, (struct sockaddr *)& cliente, &longitudCliente);
       car[1] = '\0';
       getnameinfo((struct sockaddr *)&cliente, longitudCliente, host, NI_MAXHOST, serv,
NI_MAXSERV, NI_NUMERICHOST | NI_NUMERICSERV);
       printf("%li bytes de %s:%s\n", strlen(car), host, serv);
       if(strcmp(car, "t") == 0){
```

```
          time_t tiempo = time(NULL);
          struct tm * hora = localtime(&tiempo);
          strftime(ss, 256, "%T", hora);
          sendto(sfd, ss, strlen(ss) + 1, 0, (struct sockaddr *)&cliente, longitudCliente);
        }
        else if(strcmp(car, "d") == 0){
          time_t tiempo = time(NULL);
          struct tm * hora = localtime(&tiempo);
          strftime(ss, 256, "%F", hora);
          sendto(sfd, ss, strlen(ss) + 1, 0, (struct sockaddr *)&cliente, longitudCliente);
        }
        else if(strcmp(car, "q") == 0) break;

      }

      else if(FD_ISSET(0, &set)){
        read(0, car, 2);
        car[1] = '\0';
        if(strcmp(car, "t") == 0){
          time_t tiempo = time(NULL);
          struct tm * hora = localtime(&tiempo);
          strftime(ss, 256, "%T", hora);
          printf("%s", ss);
        }
        else if(strcmp(car, "d") == 0){
          time_t tiempo = time(NULL);
          struct tm * hora = localtime(&tiempo);
          strftime(ss, 256, "%F", hora);
          printf("%s", ss);
        }
        else if(strcmp(car, "q") == 0) break;

      }
    }
  */
int pid;
  while(1){
        for (int i = 0; i < NUMPROCESOS; ++i){
           pid = fork();
           if(pid == 0) break;
        }
        if(pid == 0){
          recvfrom(sfd, &car, 2, 0, (struct sockaddr *)& cliente, &longitudCliente);
          printf("Atendido por: %i\n", getpid());
          car[1] = '\0';
          getnameinfo((struct sockaddr *)&cliente, longitudCliente, host, NI_MAXHOST, serv,
NI_MAXSERV, NI_NUMERICHOST | NI_NUMERICSERV);
          printf("%li bytes de %s:%s\n", strlen(car), host, serv);
          if(strcmp(car, "t") == 0){
            time_t tiempo = time(NULL);
            struct tm * hora = localtime(&tiempo);
            strftime(ss, 256, "%T", hora);
            sendto(sfd, ss, strlen(ss) + 1, 0, (struct sockaddr *)&cliente, longitudCliente);
          }
          else if(strcmp(car, "d") == 0){
            time_t tiempo = time(NULL);
            struct tm * hora = localtime(&tiempo);
```

```
          strftime(ss, 256, "%F", hora);
          sendto(sfd, ss, strlen(ss) + 1, 0, (struct sockaddr *)&cliente, longitudCliente);
        }
        else if(strcmp(car, "q") == 0) break;
      }
      else wait(NULL);
}

    exit(EXIT_SUCCESS);
 }
```

Salida servidor:

```
./prueba :: 3000
Atendido por: 7079
1 bytes de ::ffff:192.168.0.100:47497
Atendido por: 7078
1 bytes de ::ffff:192.168.0.100:55847
Atendido por: 7080
1 bytes de ::ffff:192.168.0.100:43393
```

Salida cliente:

```
./sockets 192.168.0.1 3000 t
13:43:39usuarioso@ssoo:~$ ./sockets 192.168.0.1 3000 t
13:44:48usuarioso@ssoo:~$ ./sockets 192.168.0.1 3000 t
13:44:49usuarioso@ssoo:~$
```

## Protocolo TCP - Servidor de eco

TCP nos ofrece un servicio orientado a conexión y fiable. Una vez creado el socket, debe ponerse en estado LISTEN (apertura pasiva, `listen(2)` ) y a continuación quedarse a la espera de conexiones entrantes mediante una llamada `accept(2)`.

***Ejercicio 6.*** Crear un servidor TCP de eco que escuche por conexiones entrantes en una dirección (IPv4 o IPv6) y puerto dados. Cuando reciba una conexión entrante, debe mostrar la dirección y número de puerto del cliente. A partir de ese momento, enviará al cliente todo lo que reciba desde el mismo (eco). Comprobar su funcionamiento empleando la herramienta Netcat como cliente. Comprobar qué sucede si varios clientes intentan conectar al mismo tiempo.

Ejemplo:

| **Servidor** | **Cliente** |
|---|---|
| `$ ./echo_server :: 2222`<br>`Conexión desde fd00::a:0:0:0:1 53456`<br>`Conexión terminada` | `$ nc -6 fd00::a:0:0:0:1 2222`<br>**`Hola`**<br>`Hola`<br>**`Qué tal`**<br>`Qué tal`<br>**`^C`**<br>`$` |

```c
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <netdb.h>


#define BUF_SIZE 500

int main(int argc, char *argv[])
{
struct addrinfo hints;
struct addrinfo *result, *rp;
int sfd, s;
struct sockaddr_storage peer_addr;
socklen_t peer_addr_len;
ssize_t nread;
char buf[BUF_SIZE];

if (argc < 3) {
    fprintf(stderr, "Faltan argumentos\n");
    exit(EXIT_FAILURE);
}

memset(&hints, 0, sizeof(struct addrinfo));
hints.ai_family = AF_UNSPEC;    /* Allow IPv4 or IPv6 */
hints.ai_socktype = SOCK_STREAM; /* Datagram socket */
hints.ai_flags = AI_PASSIVE;    /* For wildcard IP address */
hints.ai_protocol = 0;          /* Any protocol */
hints.ai_canonname = NULL;
hints.ai_addr = NULL;
hints.ai_next = NULL;

s = getaddrinfo(argv[1], argv[2], &hints, &result);
if (s != 0) {
```

```
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
    exit(EXIT_FAILURE);
}

/* getaddrinfo() returns a list of address structures.
   Try each address until we successfully bind(2).
   If socket(2) (or bind(2)) fails, we (close the socket
   and) try the next address. */

for (rp = result; rp != NULL; rp = rp->ai_next) {
    sfd = socket(rp->ai_family, rp->ai_socktype,
            rp->ai_protocol);
    if (sfd == -1)
        continue;

    if (bind(sfd, rp->ai_addr, rp->ai_addrlen) == 0)
        break;                  /* Success */

    close(sfd);
}

if (rp == NULL) {               /* No address succeeded */
    fprintf(stderr, "Could not bind\n");
    exit(EXIT_FAILURE);
}

freeaddrinfo(result);           /* No longer needed */

/* Read datagrams and echo them back to sender */

listen(sfd, 5);

while(1){
   int sockfd = accept(sfd, (struct sockaddr *) &peer_addr,
&peer_addr_len);

   char host[NI_MAXHOST], service[NI_MAXSERV];

   s = getnameinfo((struct sockaddr *) &peer_addr, peer_addr_len,
```

```
host, NI_MAXHOST, service, NI_MAXSERV, NI_NUMERICSERV);
   printf("Conexión desde %s:%s\n", host, service);

   while(1){
   int leido = recv(sockfd, buf, BUF_SIZE, 0);
   buf[leido] = '\0';
   if(strcmp(buf, "Q\n") == 0) {
           printf("Finalizando conexion\n");
           break;
       }
   printf("Recibido: %s", buf);
   send(sockfd, buf, strlen(buf), 0);
   }
}


}

Salida servidor:
./programa :: 2224
Conexión desde fd00:0:0:a::100:49742
Recibido: hola
Recibido: que
Recibido: tal
Recibido: va
Recibido: todo
Recibido: por
Recibido: ahi
Finalizando conexion

Salida cliente:
nc -6 fd00::a:0:0:0:1 2224
hola
hola
que
que
tal
tal
va
```

```
va
todo
todo
por
por
ahi
ahi
Q
```

***Ejercicio 7.*** Escribir el cliente para conectarse con el servidor del ejercicio anterior. El cliente recibirá la dirección y el puerto del servidor como argumentos y, una vez establecida la conexión con el servidor, le enviará lo que el usuario escriba por teclado. Mostrará en la consola la respuesta recibida desde el servidor. Cuando el usuario escriba el carácter 'Q' como único carácter de una línea, el cliente cerrará la conexión con el servidor.

Ejemplo:

| Servidor | Cliente |
|---|---|
| `$ ./echo_server :: 2222`<br>`Conexión desde fd00::a:0:0:0:1 53445`<br>`Conexión terminada`<br>`$` | `$ ./echo_client fd00::a:0:0:0:1 2222`<br>**`Hola`**<br>`Hola`<br>**`Q`**<br>`$` |

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define BUF_SIZE 500

int main(int argc, char *argv[])
{
struct addrinfo hints;
struct addrinfo *result, *rp;
int sfd, s, j;
size_t len;
```

```
ssize_t nread;
char buf[BUF_SIZE];
char buf2[BUF_SIZE];
if (argc < 3) {
fprintf(stderr, "Usage: %s host port msg...\n", argv[0]);
exit(EXIT_FAILURE);
}

/* Obtain address(es) matching host/port */

memset(&hints, 0, sizeof(struct addrinfo));
hints.ai_family = AF_UNSPEC;    /* Allow IPv4 or IPv6 */
hints.ai_socktype = SOCK_STREAM; /* Datagram socket */
hints.ai_flags = 0;
hints.ai_protocol = 0;          /* Any protocol */

s = getaddrinfo(argv[1], argv[2], &hints, &result);
if (s != 0) {
   fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
   exit(EXIT_FAILURE);
}
for (rp = result; rp != NULL; rp = rp->ai_next) {
   sfd = socket(rp->ai_family, rp->ai_socktype, rp->ai_protocol);
if (sfd == -1)
   continue;

if (connect(sfd, rp->ai_addr, rp->ai_addrlen) != -1)
   break;                  /* Success */

close(sfd);
}

if (rp == NULL) {               /* No address succeeded */
fprintf(stderr, "Could not connect\n");
exit(EXIT_FAILURE);
}
freeaddrinfo(result);           /* No longer needed */
```

```
while(1){
   int leido = read(0, buf, BUF_SIZE);

   buf[leido] = '\0';
   //printf("%s", buf);

   send(sfd, buf, strlen(buf), 0);
   if(strcmp(buf, "Q\n") == 0) {
       close(sfd);
       exit(0);
   }
   int recibido = recv(sfd, buf2,  BUF_SIZE, 0);
   buf2[recibido] = '\0';
   printf("%s", buf2);
}




exit(EXIT_SUCCESS);
}

Servidor:
Conexión desde fd00:0:0:a::100:56584
Recibido: ola
Recibido: que
Recibido: tal
Recibido: va
Recibido: todo
Finalizando conexion

Cliente:
./programa fd00::a:0:0:0:1 2222
ola
ola
que
que
tal
tal
```

```
va
va
todo
todo
Q
```

***Ejercicio 8***. Modificar el código del servidor para que acepte varias conexiones simultáneas. Cada petición debe gestionarse en un proceso diferente, siguiendo el patrón *accept-and-fork*. El proceso padre debe cerrar el socket devuelto por `accept(2)`.

```c
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <netdb.h>


#define BUF_SIZE 500

int main(int argc, char *argv[])
{
struct addrinfo hints;
struct addrinfo *result, *rp;
int sfd, s;
struct sockaddr_storage peer_addr;
socklen_t peer_addr_len;
ssize_t nread;
char buf[BUF_SIZE];

if (argc < 3) {
    fprintf(stderr, "Faltan argumentos\n");
    exit(EXIT_FAILURE);
}

memset(&hints, 0, sizeof(struct addrinfo));
```

```
hints.ai_family = AF_UNSPEC;    /* Allow IPv4 or IPv6 */
hints.ai_socktype = SOCK_STREAM; /* Datagram socket */
hints.ai_flags = AI_PASSIVE;    /* For wildcard IP address */
hints.ai_protocol = 0;          /* Any protocol */
hints.ai_canonname = NULL;
hints.ai_addr = NULL;
hints.ai_next = NULL;

s = getaddrinfo(argv[1], argv[2], &hints, &result);
if (s != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
    exit(EXIT_FAILURE);
}

/* getaddrinfo() returns a list of address structures.
   Try each address until we successfully bind(2).
   If socket(2) (or bind(2)) fails, we (close the socket
   and) try the next address. */

for (rp = result; rp != NULL; rp = rp->ai_next) {
    sfd = socket(rp->ai_family, rp->ai_socktype,
            rp->ai_protocol);
    if (sfd == -1)
        continue;

    if (bind(sfd, rp->ai_addr, rp->ai_addrlen) == 0)
        break;                  /* Success */

    close(sfd);
}

if (rp == NULL) {               /* No address succeeded */
    fprintf(stderr, "Could not bind\n");
    exit(EXIT_FAILURE);
}

freeaddrinfo(result);           /* No longer needed */

/* Read datagrams and echo them back to sender */
```

```
listen(sfd, 5);

while(1){
   int sockfd = accept(sfd, (struct sockaddr *) &peer_addr,
&peer_addr_len);
   int pid = fork();
   if(pid == 0){
       char host[NI_MAXHOST], service[NI_MAXSERV];

       s = getnameinfo((struct sockaddr *) &peer_addr, peer_addr_len,
host, NI_MAXHOST, service, NI_MAXSERV, NI_NUMERICSERV);
       printf("Conexión desde %s:%s\n", host, service);

       while(1){
       int leido = recv(sockfd, buf, BUF_SIZE, 0);
       buf[leido] = '\0';
       if(strcmp(buf, "Q\n") == 0) {
               printf("Finalizando conexion\n");
               close(sockfd);
               break;
           }
       printf("Recibido: %s", buf);
       send(sockfd, buf, strlen(buf), 0);
       }
   }
   close(sockfd);
}


}
Servidor:
./programa :: 2223
Conexión desde fd00:0:0:a::100:52120
Recibido: hola
Conexión desde fd00:0:0:a::100:52122
Recibido: que
Recibido: tal
Finalizando conexion
```

```
Finalizando conexion

Cliente simultáneos:
Cliente 1:
/programa fd00::a:0:0:0:1 2223
hola
hola
tal
tal
Q

Cliente 2:
./programa fd00::a:0:0:0:1 2223
que
que
Q
```

***Ejercicio 9***. Añadir la lógica necesaria en el servidor para que no quede ningún proceso en estado *zombie*. Para ello, se deberá capturar la señal `SIGCHLD` y obtener la información de estado de los procesos hijos finalizados.

```c
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <netdb.h>


#define BUF_SIZE 500

int main(int argc, char *argv[])
{
struct addrinfo hints;
struct addrinfo *result, *rp;
int sfd, s;
```

```c
struct sockaddr_storage peer_addr;
socklen_t peer_addr_len;
ssize_t nread;
char buf[BUF_SIZE];

if (argc < 3) {
    fprintf(stderr, "Faltan argumentos\n");
    exit(EXIT_FAILURE);
}

memset(&hints, 0, sizeof(struct addrinfo));
hints.ai_family = AF_UNSPEC;    /* Allow IPv4 or IPv6 */
hints.ai_socktype = SOCK_STREAM; /* Datagram socket */
hints.ai_flags = AI_PASSIVE;    /* For wildcard IP address */
hints.ai_protocol = 0;          /* Any protocol */
hints.ai_canonname = NULL;
hints.ai_addr = NULL;
hints.ai_next = NULL;

s = getaddrinfo(argv[1], argv[2], &hints, &result);
if (s != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
    exit(EXIT_FAILURE);
}

/* getaddrinfo() returns a list of address structures.
   Try each address until we successfully bind(2).
   If socket(2) (or bind(2)) fails, we (close the socket
   and) try the next address. */

for (rp = result; rp != NULL; rp = rp->ai_next) {
    sfd = socket(rp->ai_family, rp->ai_socktype,
            rp->ai_protocol);
    if (sfd == -1)
        continue;

    if (bind(sfd, rp->ai_addr, rp->ai_addrlen) == 0)
        break;                  /* Success */
```

```
    close(sfd);
}

if (rp == NULL) {               /* No address succeeded */
    fprintf(stderr, "Could not bind\n");
    exit(EXIT_FAILURE);
}

freeaddrinfo(result);           /* No longer needed */

/* Read datagrams and echo them back to sender */

listen(sfd, 5);

while(1){
   int sockfd = accept(sfd, (struct sockaddr *) &peer_addr,
&peer_addr_len);
   int pid = fork();
   if(pid == 0){
       char host[NI_MAXHOST], service[NI_MAXSERV];

       s = getnameinfo((struct sockaddr *) &peer_addr, peer_addr_len,
host, NI_MAXHOST, service, NI_MAXSERV, NI_NUMERICSERV);
       printf("Conexión desde %s:%s\n", host, service);

       while(1){
       int leido = recv(sockfd, buf, BUF_SIZE, 0);
       buf[leido] = '\0';
       if(strcmp(buf, "Q\n") == 0) {
               printf("Finalizando conexion\n");
               close(sockfd);
               break;
           }
       printf("Recibido: %s", buf);
       send(sockfd, buf, strlen(buf), 0);
       }
   }
   else{
       hijosCreados++;
```

```
        close(sockfd);
    }
}
int status;
while (wait(&status) != -1){
    printf("Proceso finalizado con status %i", status);
}

}
```