Práctica 2.5. Sockets

Objetivos

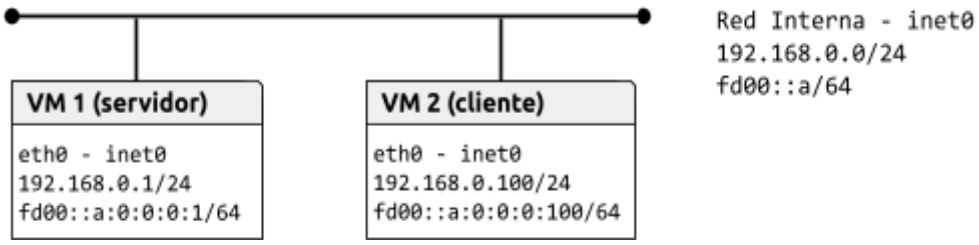
En esta práctica, nos familiarizaremos con la interfaz de programación de sockets como base para la programación de aplicaciones basadas en red, poniendo de manifiesto las diferencias de programación entre los protocolos UDP y TCP. Además, aprenderemos a programar aplicaciones independientes de la familia 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 la 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 <stdio.h>
#include <errno.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdlib.h>
#include <string.h>

int main (int argc, char**argv) {
   if (argc < 2) {
      printf("ERORR: Introduce el host por parámetros!\n");
      return -1;
   }

   struct addrinfo hints, *res, *it;
   memset(&hints, 0, sizeof(struct addrinfo));

   hints.ai_flags = AI_PASSIVE;
   hints.ai_family = AF_UNSPEC;
   hints.ai_socktype = 0;
   hints.ai_protocol = 0;
```

```
   hints.ai_addr = NULL;
   hints.ai_canonname = NULL;
   hints.ai_next = NULL;

   int sol = getaddrinfo(argv[1], NULL, &hints, &res);
   if (sol != 0) {
      printf("ERROR en el getaddrinfo\n");
      exit(EXIT_FAILURE);
   }

   char host[500];

   for(it = res ; it != NULL; it = it->ai_next){
      sol = getnameinfo(it->ai_addr, it-> ai_addrlen, host, NI_MAXHOST, NULL, 0,
NI_NUMERICHOST);
      printf("%s %i %i\n", host, it->ai_family, it->ai_socktype);
   }

   freeaddrinfo(res);
   return 0;
}
```

$ ./ej1 www.google.com
142.250.178.164 2 1
142.250.178.164 2 2
142.250.178.164 2 3
2a00:1450:4003:807::2004 10 1
2a00:1450:4003:807::2004 10 2
2a00:1450:4003:807::2004 10 3
$ ./ej1 147.96.1.9
147.96.1.9 2 1
147.96.1.9 2 2
147.96.1.9 2 3
$ ./ej1 fd00::a:0:0:0:1
fd00:0:0:a::1 10 1
fd00:0:0:a::1 10 2
fd00:0:0:a::1 10 3
$ ./ej1 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
$ ./ej1 noexiste.ucm.es
ERROR en el getaddrinfo

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:

```
$ ./time_server :: 3000
2 bytes de ::FFFF:192.168.0.100:58772
2 bytes de ::FFFF:192.168.0.100:58772
2 bytes de ::FFFF:192.168.0.100:58772
Comando X no soportado
2 bytes de ::FFFF:192.168.0.100:58772
Saliendo...
$
```

```
$ nc -u 192.168.0.1 3000
t
10:30:08 PMd
2014-01-14X
q
^C
$
```

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 <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netdb.h>
#include <time.h>


int main (int argc, char**argv) {
   if (argc < 3) {
      printf("Introduce dirección y puerto por parámetros\n");
      return -1;
   }

   struct addrinfo hints, *res, *it;

   //Rellenamos los hints de búsqueda
   memset(&hints, 0, sizeof(struct addrinfo));
   hints.ai_flags = AI_PASSIVE;
```

```
   hints.ai_family = AF_UNSPEC;
   hints.ai_socktype = SOCK_DGRAM; //PARA SERVIDORES UDP

   if (getaddrinfo(argv[1], argv[2], &hints, &res) != 0) {
      printf("ERROR al ejecutar el getaddrinfo\n");
      exit(EXIT_FAILURE);
   }

   int socketUDP = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

   if (bind(socketUDP, res->ai_addr, res->ai_addrlen) != 0) {
      printf("ERROR al ejecutar el bind\n");
      exit(EXIT_FAILURE);
   }

   freeaddrinfo(res);

   char buffer[2];
   char host[NI_MAXHOST];
   char serv[NI_MAXSERV];

   struct sockaddr_storage cliente_addr;
   socklen_t cliente_addrlen = sizeof(cliente_addr);

   while(1){
      ssize_t bytes = recvfrom(socketUDP, buffer, 2, 0, (struct sockaddr *) &cliente_addr,
&cliente_addrlen);  //Recibe un comando (de 1 byte, se pone dos por ser el tam del
buffer) y lo almacena en buffer
      buffer[1] = '\0';
      getnameinfo((struct sockaddr *) &cliente_addr, cliente_addrlen, host, NI_MAXHOST,
serv, NI_MAXSERV, NI_NUMERICHOST|NI_NUMERICSERV); //para obtener host y serv
      printf("%i byte(s) de %s:%s\n", bytes, host, serv);

      time_t tiempo = time(NULL);
      struct tm *tm = localtime(&tiempo);
      size_t max;
      char s[50];

      if (buffer[0] == 't'){ //hora
         size_t tam = strftime(s, max, "%I:%M:%S %p", tm);
         s[tam] = '\0';
         sendto(socketUDP, s, tam, 0, (struct sockaddr *) &cliente_addr, cliente_addrlen);
      }
      else if (buffer[0] =='d'){ //fecha
         size_t tam = strftime(s, max, "%Y-%m-%d", tm);
         s[tam] = '\0';
         sendto(socketUDP, s, tam, 0, (struct sockaddr *) &cliente_addr, cliente_addrlen);
      }
      else if (buffer[0] =='q'){
         printf("SALIENDO...\n");
```

```
        exit(0);
      }
      else{
        printf("Comando %d no soportado\n", buffer[0]);
      }
   }

   return 0;
}
```

VM1:	**VM2:**
$ ./ej2 :: 3000	**$ nc -u 192.168.0.1 3000**
2 byte(s) de ::ffff:192.168.0.100:52379	**t**
2 byte(s) de ::ffff:192.168.0.100:52379	04:53:28 PM**d**
2 byte(s) de ::ffff:192.168.0.100:52379	2021-12-13**X**
Comando 88 no soportado	**q**
2 byte(s) de ::ffff:192.168.0.100:52379	**^C**
SALIENDO...	

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 192.128.0.1 3000 t`.

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netdb.h>
#include <time.h>


int main (int argc, char**argv) {
   if (argc != 4) {
      printf("Introduce dirección y puerto por parámetros\n");
      return -1;
   }

   struct addrinfo hints, *res, *it;

   //Rellenamos los hints de búsqueda
   memset(&hints, 0, sizeof(struct addrinfo));
   hints.ai_flags = AI_PASSIVE;
   hints.ai_family = AF_UNSPEC;
   hints.ai_socktype = SOCK_DGRAM;
```

```
  if (getaddrinfo(argv[1], argv[2], &hints, &res) != 0) {
    printf("ERROR al ejecutar el getaddrinfo\n");
    exit(EXIT_FAILURE);
  }

  int socketUDP = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

  struct sockaddr_storage cliente_addr;
  socklen_t cliente_addrlen = sizeof(cliente_addr);

  sendto(socketUDP, argv[3], 2, 0, res->ai_addr, res->ai_addrlen);

  if (*argv[3] == 'd' || *argv[3] == 't'){
    char buffer[256];
    ssize_t size = recvfrom(socketUDP, buffer, 256, 0, (struct sockaddr *) &cliente_addr,
&cliente_addrlen);
    buffer[size] = '\0';
    printf("%s\n", buffer);
  }

  return 0;
}
```

VM1:	***VM2***:
$ ./ej2 :: 3000 2 byte(s) de ::ffff:192.168.0.100:41329 2 byte(s) de ::ffff:192.168.0.100:43456	**$ ./ej3 192.168.0.1 3000 t** 11:31:53 AM **$ ./ej3 192.168.0.1 3000 d** 2021-12-15

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 <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netdb.h>
#include <time.h>
#include <unistd.h>
```

```
int main (int argc, char**argv) {
   if (argc < 3) {
      printf("Introduce dirección y puerto por parámetros\n");
      return -1;
   }

   struct addrinfo hints, *res, *it;

   //Rellenamos los hints de búsqueda
   memset(&hints, 0, sizeof(struct addrinfo));
   hints.ai_flags = AI_PASSIVE;
   hints.ai_family = AF_UNSPEC;
   hints.ai_socktype = SOCK_DGRAM;

   if (getaddrinfo(argv[1], argv[2], &hints, &res) != 0) {
      printf("ERROR al ejecutar el getaddrinfo\n");
      exit(EXIT_FAILURE);
   }

   int socketUDP = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

   if (bind(socketUDP, res->ai_addr, res->ai_addrlen) != 0) {
      printf("ERROR al ejecutar el bind\n");
      exit(EXIT_FAILURE);
   }

   freeaddrinfo(res);

   char buffer[2];
   char host[NI_MAXHOST];
   char serv[NI_MAXSERV];

   time_t tiempo = time(NULL);
   struct tm *tm = localtime(&tiempo);
   char bufferSalida[100];

   struct sockaddr_storage cliente_addr;
   socklen_t cliente_addrlen = sizeof(cliente_addr);

   fd_set setLectura; //Creamos descriptor de lectura
   int df = -1;

   while(1){
      FD_ZERO(&setLectura);          //Vaciamos el puntero
      FD_SET(socketUDP, &setLectura);  //Metemos el descriptor del socket
      FD_SET(0, &setLectura);        //Metemos el descriptor de la entrada estándar
      df = select(socketUDP+1, &setLectura, NULL, NULL, NULL);
```

```
      if(FD_ISSET(socketUDP, &setLectura)){ //si ha llegado por red el comando
         ssize_t bytes = recvfrom(socketUDP, buffer, 2, 0, (struct sockaddr *) &cliente_addr,
&cliente_addrlen); //Recibe un comando (de 1 byte, se pone dos por ser el tam del
buffer) y lo almacena en buffer
         buffer[1] = '\0';
         getnameinfo((struct sockaddr *) &cliente_addr, cliente_addrlen, host,
NI_MAXHOST, serv, NI_MAXSERV, NI_NUMERICHOST|NI_NUMERICSERV);
         printf("[RED] %i byte(s) de %s:%s\n", bytes, host, serv);

         if (buffer[0] == 't'){ //hora
            size_t tam = strftime(bufferSalida, 100, "%I:%M:%S %p", tm);
            bufferSalida[tam] = '\0';
            sendto(socketUDP, bufferSalida, tam, 0, (struct sockaddr *) &cliente_addr,
cliente_addrlen);
         }
         else if (buffer[0] =='d'){ //fecha
            size_t tam = strftime(bufferSalida, 100, "%Y-%m-%d", tm);
            bufferSalida[tam] = '\0';
            sendto(socketUDP, bufferSalida, tam, 0, (struct sockaddr *) &cliente_addr,
cliente_addrlen);
         }
         else if (buffer[0] =='q'){
            printf("SALIENDO...\n");
            exit(0);
         }
         else{
            printf("Comando erroneo %d...\n", buffer[0]);
         }
      }
      else{ //Si llega por consola
         read(0, buffer, 2); //Leer de la entrada estandar(terminal) dos bytes y almacenarlos
en buffer
         buffer[1] = '\0';
         printf("[TERMINAL] %i byte(s)\n", 2);

         if (buffer[0] == 't'){ //hora
            size_t tam = strftime(bufferSalida, 100, "%I:%M:%S %p", tm);
            bufferSalida[tam] = '\0';
            printf("%s\n", bufferSalida);
         }
         else if (buffer[0] =='d'){ //fecha
            size_t tam = strftime(bufferSalida, 100, "%Y-%m-%d", tm);
            bufferSalida[tam] = '\0';
            printf("%s\n", bufferSalida);
         }
         else if (buffer[0] =='q'){
            printf("SALIENDO...\n");
            exit(0);
         }
         else{
```

```
            printf("Comando erroneo %d...\n", buffer[0]);
        }
      }
   }

   return 0;
}
```

***VM1*:** **$ ./ej4 :: 3000** **t** [TERMINAL] 2 byte(s) 11:47:44 AM **d** [TERMINAL] 2 byte(s) 2021-12-15 [RED] 2 byte(s) de ::ffff:192.168.0.100:58227 [RED] 2 byte(s) de ::ffff:192.168.0.100:58227 [RED] 2 byte(s) de ::ffff:192.168.0.100:58227 Comando erroneo 88...	***VM2:*** **$ nc -u 192.168.0.1 3000** **t** 11:48:01 AM**d** 2021-12-15**X**

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. Para terminar el servidor, enviar la señal `SIGTERM` al grupo de procesos.

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netdb.h>
#include <time.h>
#include <signal.h>
#include <sys/wait.h>

#define NPROCESOS 2

int main (int argc, char**argv) {
   if (argc < 3) {
      printf("Introduce dirección y puerto por parámetros\n");
      return -1;
   }
```

```
  struct addrinfo hints, *res, *it;

  //Rellenamos los hints de búsqueda
  memset(&hints, 0, sizeof(struct addrinfo));
  hints.ai_flags = AI_PASSIVE;
  hints.ai_family = AF_UNSPEC;
  hints.ai_socktype = SOCK_DGRAM;

  if (getaddrinfo(argv[1], argv[2], &hints, &res) != 0) {
     printf("ERROR al ejecutar el getaddrinfo\n");
     exit(EXIT_FAILURE);
  }

  int socketUDP = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

  if (bind(socketUDP, res->ai_addr, res->ai_addrlen) != 0) {
     printf("ERROR al ejecutar el bind\n");
     exit(EXIT_FAILURE);
  }

  freeaddrinfo(res);
  signal(SIGCHLD, NULL);

  char buffer[2];
  char host[NI_MAXHOST];
  char serv[NI_MAXSERV];

  time_t tiempo = time(NULL);
  struct tm *tm = localtime(&tiempo);
  char bufferSalida[100];

  struct sockaddr_storage cliente_addr;
  socklen_t cliente_addrlen = sizeof(cliente_addr);

  int status;
  for(int i = 0; i < NPROCESOS; ++i){
     pid_t pid = fork();

     if (pid == 0){
        while(1){
           ssize_t bytes = recvfrom(socketUDP, buffer, 2, 0, (struct sockaddr *)
&cliente_addr, &cliente_addrlen);
           buffer[1] = '\0';
           getnameinfo((struct sockaddr *) &cliente_addr, cliente_addrlen, host,
NI_MAXHOST, serv, NI_MAXSERV, NI_NUMERICHOST|NI_NUMERICSERV);
           printf("%i byte(s) de %s:%s\n", bytes, host, serv);

           if (buffer[0] == 't'){ //hora
              size_t bytesT = strftime(bufferSalida, 100, "%I:%M:%S %p", tm);
```

```
            bufferSalida[bytesT] = '\0';
            sendto(socketUDP, bufferSalida, bytesT, 0, (struct sockaddr *) &cliente_addr,
cliente_addrlen);
          }
          else if (buffer[0] =='d'){ //fecha
            size_t bytesT = strftime(bufferSalida, 100, "%Y-%m-%d", tm);
            bufferSalida[bytesT] = '\0';
            sendto(socketUDP, bufferSalida, bytesT, 0, (struct sockaddr *) &cliente_addr,
cliente_addrlen);
          }
          else if (buffer[0] =='q'){
            printf("SALIENDO...\n");
            exit(0);
          }
          else{
            printf("Comando erroneo %d...\n", buffer[0]);
          }
        }
      }
      else{
        wait(&status);
      }
    }

    kill(0, SIGTERM);
    return 0;
}
```

VM1:	***VM2:***
$ ./ej5 :: 3000	***Terminal 1:***
2 byte(s) de ::ffff:192.168.0.100:50731	**$ nc -u 192.168.0.1 3000**
2 byte(s) de ::ffff:192.168.0.100:50731	**t**
2 byte(s) de ::ffff:192.168.0.100:50731	12:02:07 PM**d**
Comando erroneo 88...	2021-12-15**X**
2 byte(s) de ::ffff:192.168.0.100:44094	
2 byte(s) de ::ffff:192.168.0.100:44094	***Terminal 2***
2 byte(s) de ::ffff:192.168.0.100:44094	**$ nc -u 192.168.0.1 3000**
SALIENDO...	**t**
Pid exited.	12:02:07 PM**d**
	2021-12-15**q**

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:

```
$ ./echo_server :: 2222
Conexión desde fd00::a:0:0:0:100 53456
Conexión terminada
```

```
$ nc -6 fd00::a:0:0:0:1 2222
Hola
Hola
Qué tal
Qué tal
^C
$
```

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

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

  if (argc < 2) {
    printf("Introduce los parámetros.\n");
    return -1;
  }

  struct addrinfo hints, *res;

  memset(&hints,0,sizeof(struct addrinfo));
  hints.ai_flags = AI_PASSIVE;
  hints.ai_family = AF_UNSPEC;
  hints.ai_socktype = SOCK_STREAM;  //PARA SERVIDORES TCP


  if (getaddrinfo(argv[1], argv[2], &hints, &res) != 0) {
    printf("ERROR al ejecutar el getaddrinfo");
    exit(EXIT_FAILURE);
  }
```

```
   int socketTCP = socket(res->ai_family, res->ai_socktype, 0);

   if (bind(socketTCP, res->ai_addr, res->ai_addrlen) != 0) {
      printf("ERROR al ejecutar el bind\n");
      exit(EXIT_FAILURE);
   }

   freeaddrinfo(res);
   listen(socketTCP, 5);

   struct sockaddr_storage cliente_addr;
   socklen_t cliente_addrlen = sizeof(cliente_addr);

   char buffer[81];
   char host[NI_MAXHOST];
   char serv[NI_MAXSERV];
   int clisd;
   ssize_t bytes;

   while (1) {
      clisd = accept(socketTCP,(struct sockaddr *) &cliente_addr, &cliente_addrlen);

      while (1) {
         getnameinfo((struct sockaddr *)&cliente_addr, cliente_addrlen, host,
NI_MAXHOST, serv, NI_MAXSERV, NI_NUMERICHOST);
         printf("Conexión desde %s:%s\n", host, serv);

         bytes = recv(clisd, buffer, 80, 0);
         buffer[bytes] = '\0';

         if ((buffer[0] == 'Q') && (bytes == 2)) {
            printf("Conexión terminada\n");
            break;
         }
         send(clisd, buffer, bytes, 0);
      }
   }

   close(clisd);
   return 0;
}
```

VM1: **$ ./ej6 :: 2222** Conexión desde fd00:0:0:a::100:58954 Conexión desde fd00:0:0:a::100:58954 Conexión desde fd00:0:0:a::100:58954 Conexión terminada!!	*VM2:* **$ nc -6 fd00::a:0:0:0:1 2222** **Hola** Hola **Qué tal** Qué tal **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 y terminará.

Ejemplo:

`$ ./echo_server :: 2222`
`Conexión desde fd00::a:0:0:0:100 53445`
`Conexión terminada`
	`$ ./echo_client fd00::a:0:0:0:1 2222`
	`Hola`
	`Hola`
	`Q`
	`$`

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

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

  if (argc < 2) {
    printf("Introduce los parámetros.\n");
    return -1;
  }

  struct addrinfo hints, *res;

  memset(&hints,0,sizeof(struct addrinfo));
  hints.ai_flags = AI_PASSIVE;
  hints.ai_family = AF_UNSPEC;
  hints.ai_socktype = SOCK_STREAM;


  if (getaddrinfo(argv[1], argv[2], &hints, &res) != 0) {
    printf("ERROR al ejecutar el getaddrinfo");
    exit(EXIT_FAILURE);
  }
```

```
    int socketTCP = socket(res->ai_family, res->ai_socktype, 0);
    connect(socketTCP,(struct sockaddr *)res->ai_addr, res->ai_addrlen);
    freeaddrinfo(res);

    char bufferIN[256];
    char bufferOUT[256];
    char host[NI_MAXHOST];
    char serv[NI_MAXSERV];
    ssize_t bytes;

    while (1) {
        bytes = read(0, bufferOUT, 255);
        bufferOUT[bytes] = '\0';
        send(socketTCP, bufferOUT, bytes, 0);

        if ((bufferOUT[1] == 'Q') && (bytes == 2)) {
            printf("Conexión terminada!!\n");
            break;
        }

        bytes = recv(socketTCP, bufferIN, 255, 0);
        bufferIN[bytes] = '\0';
        printf("%s", bufferIN);
    }

    close(socketTCP);
    return 0;
}
```

VM1:	***VM2:***
$ ./ej7:: 2222	**$ ./ej7 fd00::a:0:0:0:1 2222**
Conexión desde fd00::a:0:0:0:100 58958	**Hola**
Conexión desde fd00:0:0:a::100:58958	Hola
Conexión terminada!!	**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)`.

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netdb.h>
#include <time.h>
#include <signal.h>
#include <sys/wait.h>

int main (int argc, char**argv) {
   if (argc < 3) {
      printf("Introduce dirección y puerto por parámetros\n");
      return -1;
   }

   struct addrinfo hints, *res, *it;

   //Rellenamos los hints de búsqueda
   memset(&hints, 0, sizeof(struct addrinfo));
   hints.ai_flags = 0;
   hints.ai_family = AF_UNSPEC;
   hints.ai_socktype = SOCK_STREAM;

   if (getaddrinfo(argv[1], argv[2], &hints, &res) != 0) {
      printf("ERROR al ejecutar el getaddrinfo\n");
      exit(EXIT_FAILURE);
   }

   int socketTCP = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

   if (bind(socketTCP, res->ai_addr, res->ai_addrlen) != 0) {
      printf("ERROR al ejecutar el bind\n");
      exit(EXIT_FAILURE);
   }

   listen(socketTCP, 5);
   freeaddrinfo(res);


   char buffer[256];
   char host[NI_MAXHOST];
   char serv[NI_MAXSERV];
```

```
   int clisd;
   ssize_t bytes;

   struct sockaddr_storage cliente_addr;
   socklen_t cliente_addrlen = sizeof(cliente_addr);


   while(1){
      clisd = accept(socketTCP,(struct sockaddr *) &cliente_addr, &cliente_addrlen);
      getnameinfo((struct sockaddr *)&cliente_addr, cliente_addrlen, host, NI_MAXHOST,
serv, NI_MAXSERV, NI_NUMERICHOST);
      printf("[PID: %i] Conexión desde %s:%s\n", getpid(), host, serv);

      pid_t pid = fork();
      if (pid == 0){
         while (1) {

            bytes = recv(clisd, buffer, 256, 0);
            buffer[bytes] = '\0';

            if ((buffer[0] == 'Q') && (bytes == 2)) {
               printf("Conexión terminada!!\n");
               break;
            } else {
               send(clisd, buffer, bytes, 0);
            }
         }
         _exit(EXIT_SUCCESS);
      }
      else{
         close(clisd);
      }
   }

   exit(EXIT_SUCCESS);
   return 0;
}
```

VM1:	***VM2:***	***VM2:***
$ ./ej8 :: 2222	***Terminal 1:***	***Terminal 2:***
[PID: 8167] Conexión desde fd00:0:0:a::100:58962	**$ nc -6 fd00::a:0:0:0:1 2222**	**$ nc -6 fd00::a:0:0:0:1 2222**
[PID: 8167] Conexión desde fd00:0:0:a::100:58964	**t**	**t**
Conexión terminada!!	t	t
Conexión terminada!!	**t**	**Hola**
	t	Hola
	Q	**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.

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netdb.h>
#include <time.h>
#include <signal.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <sys/select.h>

void handler(int signal){
   int status;
   wait(&status);
   printf("Proceso finalizado con status %i\n", status);
}

int main (int argc, char**argv) {
   if (argc < 3) {
      printf("Introduce dirección y puerto por parámetros\n");
      return -1;
   }

   struct addrinfo hints, *res;

   //Rellenamos los hints de búsqueda
   memset(&hints, 0, sizeof(struct addrinfo));
   hints.ai_flags = AI_PASSIVE;
```

```
  hints.ai_family = AF_UNSPEC;
  hints.ai_socktype = SOCK_STREAM;

  if (getaddrinfo(argv[1], argv[2], &hints, &res) != 0) {
     printf("ERROR al ejecutar el getaddrinfo\n");
     exit(EXIT_FAILURE);
  }

  int socketTCP = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

  if (bind(socketTCP, res->ai_addr, res->ai_addrlen) != 0) {
     printf("ERROR al ejecutar el bind\n");
     exit(EXIT_FAILURE);
  }

  freeaddrinfo(res);
  listen(socketTCP, 5);

  char buffer[100];
  char host[NI_MAXHOST];
  char serv[NI_MAXSERV];
  int clisd;
  ssize_t bytes;

  struct sockaddr_storage cliente_addr;
  socklen_t cliente_addrlen = sizeof(cliente_addr);
  struct sigaction act;
  act.sa_handler = handler;

  int status;
  while(1){
     clisd = accept(socketTCP,(struct sockaddr *) &cliente_addr, &cliente_addrlen);
     pid_t pid = fork();

     if (pid == 0){
        while (1) {
           getnameinfo((struct sockaddr *)&cliente_addr, cliente_addrlen, host,
NI_MAXHOST, serv, NI_MAXSERV, NI_NUMERICHOST);

           bytes = recv(clisd, buffer, 100, 0);
           buffer[bytes] = '\0';

           if ((buffer[0] == 'Q') && (bytes == 2)) {
              printf("Conexión terminada!!\n");
              exit(0);
           } else {
              printf("[PID: %i] Conexión desde %s:%s\n", getpid(), host, serv);
              send(clisd, buffer, bytes, 0);
           }
           kill(getppid(),SIGCHLD);
```

```
        }
      }
      else{
         sigaction(SIGCHLD, &act, NULL);
         close(clisd);
      }
   }

   close(clisd);
   return 0;
}
```

VM1:	*VM2:*
$ ./ej9 :: 2222 [PID: 10572] Conexión desde fd00:0:0:a::100:58980 [PID: 10572] Conexión desde fd00:0:0:a::100:58980 Conexión terminada!! Proceso finalizado con status 32765	**$ nc -6 fd00::a:0:0:0:1 2222** **Hola** Hola **Caracola** Caracola **Q**