Práctica 2.5. Sockets

Objetivos

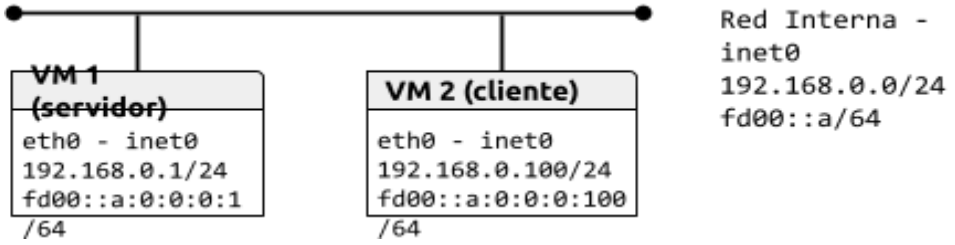
En esta práctica, nos familiarizaremos con la interfaz de programación de sockets como base para la programación de aplicaciones basadas en red, poniendo de manifiesto las diferencias de programación entre los protocolos UDP y TCP. Además, aprenderemos a programar aplicaciones independientes de la familia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

```
#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 pruebas;
        struct addrinfo *solucion, *r;
        int sfd, s, j;
        size_t l;
        size_t size_read;
        char buf[BUF_SIZE];
        if (argc < 3) {
                fprintf(stderr, "Usage: %s host port msg...\n", argv[0]);
                exit(EXIT_FAILURE);
          }
        memset(&pruebas, 0, sizeof(struct addrinfo));
        pruebas.ai_family = AF_UNSPEC;
        pruebas.ai_socktype = SOCK_DGRAM;
        pruebas.ai_flags = 0;
        pruebas.ai_protocol = 0;
        s = getaddrinfo(argv[1], argv[2], &pruebas, &solucion);
        if (s != 0) {
                fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
                exit(EXIT_FAILURE);
        }
        for (r = solucion; r != NULL; r = r->ai_next) {
                sfd = socket(r->ai_family, r->ai_socktype, r->ai_protocol);
                if (sfd == -1)
                   continue;
                if (connect(sfd, r->ai_addr, r->ai_addrlen) != -1)
                   break;
                close(sfd);
        }
        if (r == NULL) {
     fprintf(stderr, "Could not connect\n");
     exit(EXIT_FAILURE);
        }
        freeaddrinfo(solucion);
        for (j = 3; j < argc; j++) {
        l = strlen(argv[j]) + 1;
                 if (l + 1 > BUF_SIZE) {
         fprintf(stderr,"Ignoring long message in argument %d\n", j);
                   continue;
                }
                 if (write(sfd, argv[j], l) != l) {
                   fprintf(stderr, "partial/failed write\n");
```

```
            exit(EXIT_FAILURE);
        }
        size_read=(sfd,buf, BUF_SIZE);
        if (size_read == -1) {
            perror("read");
            exit(EXIT_FAILURE);
        }
        printf("Received %ld bytes: %s\n", (long) size_read, buf);
    }

    exit(EXIT_SUCCESS);
}
```

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
```

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

void main(int argc, char *argv[]) {
   struct addrinfo hints;
   struct addrinfo *res;
   struct sockaddr_storage cli;
   time_t rawtime;
   struct tm* timeinfo;
   char tbuffer[9];
   char buf[81], host[NI_MAXHOST], serv[NI_MAXSERV];

   // Rellenar info hints struct
   hints.ai_family = AF_UNSPEC;    /* Allow IPv4 or IPv6 */
   hints.ai_socktype = SOCK_DGRAM; /* Datagram socket */
   hints.ai_flags = AI_PASSIVE;    /* For wildcard IP address */
   hints.ai_protocol = 0;          /* Any protocol */

   // Llamada a getaddrinfo para obtener la info en la la struct res
   getaddrinfo(argv[1], argv[2], &hints, &res);
   // Creación del socket con la información almacenada en la struct res
   int sd = socket(res->ai_family, res->ai_socktype, 0);
   // Llamada a bind para enlazar el socket con la dirección de la struct res
   bind(sd, (struct sockaddr *)res->ai_addr, res->ai_addrlen);
   // Llamada a freeaddrinfo para liberar la memoria asignada a la struct res
```

```
  freeaddrinfo(res);

  // Bucle de escucha del servidor:
  while(1) {
    // socklen_t clen almacena el tamaño de la dirección cliente, que será modificado cuando
    // la llamada a recvfrom() regrese para indicar el tamaño real de la direccion.
    socklen_t clen = sizeof(cli);
    // guardamos en c el número de bytes recibidos, devueltos por la llamada a recvfrom(), a la cual
    // le pasamos el socket, un buffer y su tamaño, los flags y punteros a la dirección y a su tamaño.
    int c = recvfrom(sd, buf, 80, 0, (struct sockaddr*) &cli, &clen);
    buf[c] = '\0';
        // Tras llenarse la dirección cli y su tamaño traducimos la direccion al nombre con
getnameinfo(),
    // para ello le pasamos un puntero a la direccion, su tamaño, buffers y tamaños para el host y el
    // servidor, así como el flag NI_NUMERICHOST para que devuelva el host en forma numérica.
        getnameinfo((struct sockaddr*) &cli, clen, host, NI_MAXHOST, serv, NI_MAXSERV,
NI_NUMERICHOST);
    time(&rawtime);
    timeinfo = localtime(&rawtime);
    if(buf[0] == 't') { // Si obtenemos una t del cliente obtenemos y mostramos la hora
        // Imprimimos por pantalla los bytes, dirección y puerto del cliente
        printf("%ld bytes de %s:%s\n", c, host, serv);
        ssize_t chars = strftime(tbuffer, sizeof(tbuffer), "%T", timeinfo);
        sendto(sd, tbuffer, chars, 0, (struct sockaddr *)&cli, clen);
    } else if(buf[0] == 'd') { // Si obtenemos una t del cliente obtenemos y mostramos la fecha
        // Imprimimos por pantalla los bytes, dirección y puerto del cliente
        printf("%ld bytes de %s:%s\n", c, host, serv);
        ssize_t chars = strftime(tbuffer, sizeof(tbuffer), "%D", timeinfo);
        sendto(sd, tbuffer, chars, 0, (struct sockaddr *)&cli, clen);
    } else if(buf[0] == 'q') { // Si obtenemos una q del cliente terminamos el proceso servidor
        printf("Saliendo...\n");
        exit(EXIT_SUCCESS);
    } else {
        printf("Comando no soportado %s", buf);
    }
  }
}
```

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

```
int main(int argc, char *argv[]) {
   struct addrinfo hints;
   struct addrinfo *res, *resp;
   int sd, j, s;
   size_t len;
   ssize_t nread, nwrite;
   char buf[500];

   if (argc < 3) {
      fprintf(stderr, "Usage: %s host port command...\n", argv[0]);
      exit(EXIT_FAILURE);
   }

   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 */

   getaddrinfo(argv[1], argv[2], &hints, &res);

   for(resp = res; resp != NULL; resp = resp->ai_next) {
      sd = socket(resp->ai_family, resp->ai_socktype, resp->ai_protocol);
      if(sd == -1) {
         perror("socket()");
         continue;
      }

      if(connect(sd, resp->ai_addr, resp->ai_addrlen) == -1) {
         perror("socket()");
      } else {
         break;
      }

      close(sd);
   }

   freeaddrinfo(res);

   for(j = 3; j < argc; j++) {
      len = strlen(argv[j]) + 1;
      nwrite = write(sd, argv[j], len);
      if(nwrite == -1) perror("write()");
      nread = read(sd, buf, 500);
      buf[nread] = 0;
      if(nread == -1) perror("read()");
      printf("Recibidos %ld bytes: %s\n", (long) nread, buf);
   }

   exit(EXIT_SUCCESS);
}
```

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> //Memset,

#include <sys/types.h> //getaddrinfo, socket, bind
#include <sys/socket.h> //getaddrinfo, socket, bind


#include <arpa/inet.h>
#include <netinet/in.h>
#include <netdb.h> //getaddrinfo,

#include <time.h>

#include <sys/select.h>


#include <unistd.h>


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


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

 struct addrinfo hints;
 struct addrinfo *result, *iterator;

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

 //getaddrinfo
 if (getaddrinfo(argv[1], argv[2], &hints, &result) != 0) {
    printf("ERROR: No se ha podido ejecutar el getaddrinfo.");
    exit(EXIT_FAILURE);
 }

 //socket
 int socketUDP = socket(result->ai_family, result->ai_socktype, result->ai_protocol);
```

```
//bind
if (bind(socketUDP, result->ai_addr, result->ai_addrlen) != 0) {
  printf("ERROR: No se ha podido ejecutar el bind.");
  exit(EXIT_FAILURE);
}

freeaddrinfo(result);

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

struct sockaddr_storage client_addr;
socklen_t client_addrlen = sizeof(client_addr);

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

while(1){

  while(df == -1) {
    FD_ZERO(&dflectura); //Vaciamos el puntero (No nos interesa ningún descriptor de fichero).
    FD_SET(socketUDP, &dflectura); //Metemos el descriptor del socket
    FD_SET(0, &dflectura); //Metemos el descriptor de la entrada estándar
    df = select(socketUDP+1, &dflectura, NULL, NULL, NULL);
  }


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

  if (FD_ISSET(socketUDP,&dflectura)){
    ssize_t bytes = recvfrom(socketUDP, buf, 2, 0, (struct sockaddr *) &client_addr, &client_addrlen);

          getnameinfo((struct sockaddr *) &client_addr, client_addrlen, host, NI_MAXHOST, serv,
NI_MAXSERV, NI_NUMERICHOST|NI_NUMERICSERV);
    printf("[RED] %i byte(s) de %s:%s\n", bytes, host, serv);
    buf[1] = '\0';

    if (buf[0] == 't'){
      size_t bytesT = strftime(s, max, "%I:%M:%S %p", tm);
      s[bytesT] = '\0';

      sendto(socketUDP, s, bytesT, 0, (struct sockaddr *) &client_addr, client_addrlen);

    }else if (buf[0] =='d'){
      size_t bytesT = strftime(s, max, "%Y-%m-%d", tm);
      s[bytesT] = '\0';
```

```
      sendto(socketUDP, s, bytesT, 0, (struct sockaddr *) &client_addr, client_addrlen);
    }else if (buf[0] =='q'){
      printf("Saliendo...\n");
      exit(0);
    }else{
      printf("Comando no soportado: %d...\n", buf[0]);
    }

  } else {
    read(0, buf, 2);
    printf("[Consola] %i byte(s)\n",2);
    buf[1] = '\0';

    if (buf[0] == 't'){
      size_t bytesT = strftime(s, max, "%I:%M:%S %p", tm);
      s[bytesT] = '\0';

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

    }else if (buf[0] =='d'){
      size_t bytesT = strftime(s, max, "%Y-%m-%d", tm);
      s[bytesT] = '\0';

      printf("%s\n", s);
    }else if (buf[0] =='q'){
      printf("Saliendo...\n");
      exit(0);
    }else{
      printf("Comando no soportado: %d...\n", buf[0]);
    }
  }


 }

 return 0;
}
```

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 <stdlib.h>
#include <string.h> //Memset,

#include <sys/types.h> //getaddrinfo, socket, bind
#include <sys/socket.h> //getaddrinfo, socket, bind
```

```
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netdb.h> //getaddrinfo,

#include <time.h>


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


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

 struct addrinfo hints;
 struct addrinfo *result, *iterator;

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

 //getaddrinfo
 if (getaddrinfo(argv[1], argv[2], &hints, &result) != 0) {
    printf("ERROR: No se ha podido ejecutar el getaddrinfo.");
    exit(EXIT_FAILURE);
 }

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


 //bind
 if (bind(socketUDP, result->ai_addr, result->ai_addrlen) != 0) {
  printf("ERROR: No se ha podido ejecutar el bind.");
  exit(EXIT_FAILURE);
 }

 freeaddrinfo(result);

 signal(SIGCHLD, hler);

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

 struct sockaddr_storage client_addr;
 socklen_t client_addrlen = sizeof(client_addr);
 int i = 0;
```

```
 int status;
 for (i = 0; i < 2; i++){
  pid_t pid;
  pid = fork();

  if (pid == 0) {
   while(1){
    ssize_t bytes = recvfrom(socketUDP, buf, 2, 0, (struct sockaddr *) &client_addr, &client_addrlen);
    buf[1] = '\0';

          getnameinfo((struct sockaddr *) &client_addr, client_addrlen, host, NI_MAXHOST, serv,
NI_MAXSERV, NI_NUMERICHOST|NI_NUMERICSERV);

    printf("%i byte(s) de %s:%s cuyo PID: %d\n", bytes, host, serv, getpid());

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

    if (buf[0] == 't'){
     size_t bytesT = strftime(s, max, "%I:%M:%S %p", tm);
     s[bytesT] = '\0';

     sendto(socketUDP, s, bytesT, 0, (struct sockaddr *) &client_addr, client_addrlen);

    }else if (buf[0] =='d'){
     size_t bytesT = strftime(s, max, "%Y-%m-%d", tm);
     s[bytesT] = '\0';

     sendto(socketUDP, s, bytesT, 0, (struct sockaddr *) &client_addr, client_addrlen);
    }else if (buf[0] =='q'){
     printf("Saliendo...\n");
     exit(0);
    }else{
     printf("Comando no soportado: %d...\n", buf[0]);
    }
   }
  }
  else {
   pid = wait(&status);
  }
 }
 return 0;
}
```

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.

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

void main(int argc, char *argv[]) {
  struct addrinfo hints;
  struct addrinfo *res;
  struct sockaddr_storage cli;
  char buf[81];
  hints.ai_family = AF_UNSPEC;
  hints.ai_socktype = SOCK_STREAM;
  hints.ai_flags = AI_PASSIVE;
  hints.ai_protocol = 0;
  getaddrinfo(argv[1], argv[2], &hints, &res);
  int sd = socket(res->ai_family, res->ai_socktype, 0);
  bind(sd, (struct sockaddr *)res->ai_addr, res->ai_addrlen);
  freeaddrinfo(res);
  listen(sd, 5);

  while(1) {
```

```
        socklen_t clen = sizeof(cli);
        int cli_sd = accept(sd, (struct sockaddr*) &cli, &clen);
        int c = recv(cli_sd, buf, 80, 0);
        send(cli_sd, buf, c, 0);
        close(cli_sd);
    }
}
```

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`
	`$`

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.

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <errno.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <sys/select.h>
#include <sys/socket.h>
#include <netdb.h>
```

```
#include <arpa/inet.h>


int main(int argc, char *argv[]){
   char buf[81];
   char enviar[256];
   struct addrinfo hints, cliente;
   struct addrinfo *res;
   hints.ai_flags = AI_PASSIVE;
   hints.ai_family = PF_UNSPEC;
   hints.ai_socktype = SOCK_STREAM;
   getinfo(argv[1], argv[2], &hints, &res);
   int miSocket = socket(res->ai_family,res->ai_socktype,0);
   connect(miSocket, (struct sockaddr*) res->ai_addr, res->ai_addrlen);
    while(1) {
     scanf("%s",enviar);
     if(strlen(enviar)==1 && enviar[0] == 'q') {
       close(miSocket);
       break;
     }
     send(miSocket,enviar,strlen(enviar), 0);
     recv(miSocket,buf,80,0);
     printf("%s\n", buf);
          }


   return 0;
}
```

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 $

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 <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <errno.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <sys/select.h>
#include <sys/socket.h>
#include <netdb.h>
#include <arpa/inet.h>

void main(int argc, char *argv[]) {
  struct addrinfo hints;
  struct addrinfo *res;
  struct sockaddr_storage cli;
  char buf[81];
  char host[NI_MAXHOST], serv[NI_MAXSERV];
  hints.ai_family = AF_UNSPEC;
  hints.ai_socktype = SOCK_STREAM;
  hints.ai_flags = AI_PASSIVE;
  hints.ai_protocol = 0;
  getaddrinfo(argv[1], argv[2], &hints, &res);
  int sd = socket(res->ai_family, res->ai_socktype, 0);
  bind(sd, (struct sockaddr *)res->ai_addr, res->ai_addrlen);
```

```
  freeaddrinfo(res);
  listen(sd, 5);
   pid_t pid = fork();
  if(pid == -1) {
        perror("Error en el fork");
  } else if(pid == 0) {
            while(1) {
                    socklen_t clen = sizeof(cli);
                    int cli_sd = accept(sd, (struct sockaddr*) &cli, &clen);
                    int c = recv(cli_sd, buf, 80, 0);
          buf[c] = '\0';
                    getnameinfo((struct sockaddr*) &cli, clen, host, NI_MAXHOST, serv, NI_MAXSERV,
NI_NUMERICHOST);
          pid_t pid = getpid();
                    printf("%s %s %i %s\n", host, serv, pid, buf);
                    send(cli_sd, buf, c, 0);
                    close(cli_sd);
        }
  } else if(pid > 0) {
                     while(1) {
                              socklen_t clen = sizeof(cli);
                         int cli_sd = accept(sd, (struct sockaddr*) &cli, &clen);
                              int c = recv(cli_sd, buf, 80, 0);
                      buf[c] = '\0';
                                  getnameinfo((struct sockaddr*) &cli, clen, host, NI_MAXHOST, serv,
NI_MAXSERV, NI_NUMERICHOST);
                       pid_t pid = getpid();
                                 printf("%s %s %i %s\n", host, serv, pid, buf);
                                 send(cli_sd, buf, c, 0);
        close(cli_sd);
    }
```

```
  }
}
```

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 <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <signal.h>
#include <errno.h>
#include <time.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <sys/stat.h>

volatile int stop = 0;

void hler(int senial){
 pid_t pid;

 pid = wait(NULL);

  printf("Pid exited.\n");//, pid);
```

```
}

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

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

 struct addrinfo hints;
 struct addrinfo *result;

 memset(&hints,0,sizeof(struct addrinfo));
 hints.ai_family = AF_UNSPEC;
 hints.ai_socktype = SOCK_STREAM;
 hints.ai_flags = AI_PASSIVE;
 if (getaddrinfo(argv[1], argv[2], &hints, &result) != 0) {
    printf("ERROR: No se ha podido ejecutar el getaddrinfo.");
    exit(EXIT_FAILURE);
 }

 int socketTCP = socket(result->ai_family, result->ai_socktype, 0);
 bind(socketTCP, result->ai_addr, result->ai_addrlen);
 freeaddrinfo(result);
 listen(socketTCP, 5);
 struct sockaddr_storage cli;
 socklen_t clen = sizeof(cli);
 char buf[81];
 int cli_sd;
 char host[NI_MAXHOST];
```

```
 char serv[NI_MAXSERV];
 ssize_t c;
 signal(SIGCHLD, hler);
 int status;
 while (1) {
            cli_sd = accept(socketTCP,(struct sockaddr *) &cli, &clen);
  pid_t pid;
  pid = fork();
  if (pid == 0) {
   while (1) {
             getnameinfo((struct sockaddr *)&cli, clen, host, NI_MAXHOST, serv, NI_MAXSERV,
NI_NUMERICHOST);
     printf("[PID: %i] Conexión desde %s:%s\n", getpid(), host, serv);

     c = recv(cli_sd, buf, 80, 0);
     buf[c] = '\0';
     if ((buf[0] == 'q') && (c == 2)) {
      printf("Conexión terminada\n");
      exit(0);
     } else {
      send(cli_sd, buf, c, 0);
     }
   }
  } else {
   pid = wait(&status);
   close(cli_sd);
   exit(0);
  }
 }
```

```
  close(cli_sd);

  return 0;
}
```