Práctica 2.5. Sockets

Objetivos

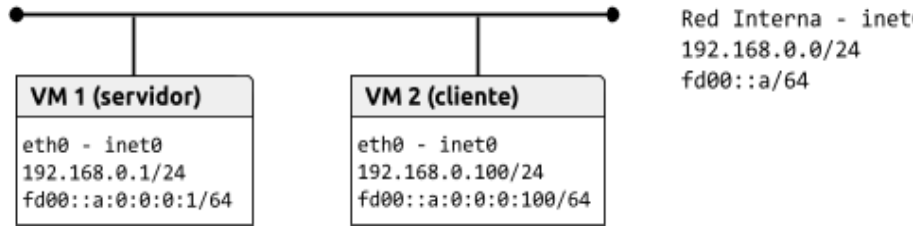
En esta práctica, nos familiarizaremos con la interfaz de programación de sockets como base para la programación de aplicaciones basadas en red, poniendo de manifiesto las diferencias de programación entre los protocolos UDP y TCP. Además, aprenderemos a programar aplicaciones independientes de la familia 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. Antes de comenzar la práctica, configurar los interfaces de red como se indica en la figura y comprobar la conectividad entre las máquinas. Observar que las VMs tienen un interfaz de red con pila dual IPv6 - IPv4.



La realización de esta práctica únicamente requiere el entorno de desarrollo (compilador, editores y depurador), que está disponible en las máquinas virtuales de la asignatura y en la máquina física del laboratorio. Se puede usar cualquier editor gráfico o de terminal. Además, se puede usar tanto el lenguaje C (compilador gcc) como C++ (compilador g++). Para compilar, escribir el comando [`gcc | g++] -o EjercicioXX EjercicioXX.c`. Para ejecutar, escribir `./EjercicioXX`. Por último, para poder acceder a Internet desde la máquina virtual, recuerda configurar correctamente la NAT. Para ello, ejecuta en una terminal `sudo dhclient eth0`.

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. 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.

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. 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.

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`.

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)`.

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.

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`
	`^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. 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)`.

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

int main (int argc, char * argv[]) {
  if (argc < 2) {
    printf("Formato incorrecto. Introduce una dirección IPv4 o IPv6 como comando\n");
    return -1;
  }

  struct addrinfo hints;
  memset(&hints, 0, sizeof(struct addrinfo));
  hints.ai_family = AF_UNSPEC;
  hints.ai_socktype = 0;
  hints.ai_flags = AI_PASSIVE;
  hints.ai_protocol = 0;
  hints.ai_canonname = NULL;
  hints.ai_addr = NULL;
  hints.ai_next = NULL;

  struct addrinfo *result;
  if (getaddrinfo(argv[1], NULL, &hints, &result) != 0) {
      perror("getaddrinfo()");
      return -1;
  }

  struct addrinfo *iterator;
  for (iterator = result; iterator != NULL; iterator = iterator->ai_next) {
    switch(iterator->ai_family) {
      case AF_INET:;
        struct sockaddr_in *info = iterator->ai_addr;
        char ip[INET_ADDRSTRLEN + 1] = "";
        inet_ntop(AF_INET, &(info->sin_addr), ip, INET_ADDRSTRLEN + 1);
                printf("%s\t", ip);
        break;

      case AF_INET6:;
        struct sockaddr_in6 *info6 = iterator->ai_addr;
        char ipv6[INET6_ADDRSTRLEN + 1] = "";
        inet_ntop(AF_INET6, &(info6->sin6_addr), ipv6, INET6_ADDRSTRLEN + 1);
                printf("%s\t", ipv6);
        break;

      default:
        printf("Error al leer ai_family.\n");
        break;
    }
        printf("%i\t%i\t\n", iterator->ai_family, iterator->ai_socktype);
    }

  freeaddrinfo(result);
  return 1;
}

/* 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
69    expresarse en cualquier formato (nombre de host, notación de punto…). Además, el
    servidor debe funcionar con direcciones IPv4 e IPv6 .
70    El servidor recibirá un comando (codificado en un carácter), de forma que 't'
    devuelva la hora,
71    'd' devuelve la fecha y 'q' termina el proceso servidor.
72    En cada mensaje el servidor debe imprimir el nombre y puerto del cliente, usar
    getnameinfo(3).
73  Probar el funcionamiento del servidor con la herramienta Netcat (comando nc o ncat)
    como cliente. Dado
74  que el servidor puede funcionar con direcciones IPv4 e IPv6, hay que usar struct
    sockaddr_storage para
75  acomodar cualquiera de ellas, por ejemplo, en recvfrom(2). */
76  #include <sys/types.h>
77  #include <sys/socket.h>
78  #include <arpa/inet.h>
79  #include <netinet/in.h>
80  #include <netdb.h>
81  #include <time.h>
82  #include <stdlib.h>
83  #include <string.h>
84  #include <stdio.h>
85  #include <errno.h>
86
87  int main (int argc, char * argv[]) {
88    if (argc < 3) {
89      printf("Formato incorrecto. Introduce una dirección IPv4 o IPv6 y un puerto como
      comandos\n");
90      return -1;
91    }
92
93    struct addrinfo hints;
94    memset(&hints, 0, sizeof(struct addrinfo));
95    hints.ai_family = AF_UNSPEC;
96    hints.ai_socktype = SOCK_DGRAM;
97    hints.ai_flags = AI_PASSIVE;
98
99    struct addrinfo *result;
100   if (getaddrinfo(argv[1], argv[2], &hints, &result) != 0) {
101     perror("getaddrinfo()");
102     return -1;
103   }
104
105   int sd = socket(result->ai_family, result->ai_socktype, result->ai_protocol);
106   if (bind(sd, result->ai_addr, result->ai_addrlen) != 0) {
107     perror("bind()");
108     return -1;
109   }
110   freeaddrinfo(result);
111
112   char comando[2];
113   char host[NI_MAXHOST];
114   char serv[NI_MAXSERV];
115
116   struct sockaddr_storage client_addr;
117   socklen_t client_addrlen = sizeof(client_addr);
118
119   while(1) {
120     ssize_t c = recvfrom(sd, comando, 2, 0, (struct sockaddr *) &client_addr, &
      client_addrlen);
121     comando[1] = '\0';
122
123     getnameinfo((struct sockaddr *) &client_addr, client_addrlen, host, NI_MAXHOST,
      serv, NI_MAXSERV, NI_NUMERICHOST|NI_NUMERICSERV);
124     printf("%i byte(s) de %s:%s\n", c, host, serv);
125
126     time_t tiempo = time(NULL);
127     struct tm *tm = localtime(&tiempo);
128     char buf[50];
129
130     switch(comando[0]) {
131       case 't':
132         size_t bytes = strftime(buf, 49, "%I:%M:%S %p", tm);
```

```
133           buf[bytes] = '\0';
134           sendto(sd, buf, bytes, 0, (struct sockaddr *) &client_addr, client_addrlen);
135           break;
136
137         case 'd':
138           size_t bytes = strftime(buf, 49, "%Y-%m-%d", tm);
139           buf[bytes] = '\0';
140           sendto(sd, buf, bytes, 0, (struct sockaddr *) &client_addr, client_addrlen);
141           break;
142
143         case 'q':
144           printf("Terminado el proceso servidor...\n");
145           return 1;
146
147         default:
148           printf("Comando no soportado.\n");
149           break;
150       }
151     }
152
153     return 1;
154   }
155
156   /* Ejercicio 3. Escribir el cliente para el servidor de hora. El cliente recibirá
      como argumentos la
157   dirección del servidor, el puerto del servidor y el comando. */
158   #include <sys/types.h>
159   #include <sys/socket.h>
160   #include <arpa/inet.h>
161   #include <netinet/in.h>
162   #include <netdb.h>
163   #include <time.h>
164   #include <stdlib.h>
165   #include <string.h>
166   #include <stdio.h>
167   #include <errno.h>
168
169   int main (int argc, char * argv[]) {
170     if (argc < 4) {
171       printf("Formato incorrecto. Introduce una dirección IPv4 o IPv6, un puerto y un
        comando t, d o q como comandos\n");
172       return -1;
173     }
174
175     struct addrinfo hints;
176     memset(&hints, 0, sizeof(struct addrinfo));
177     hints.ai_family = AF_UNSPEC;
178     hints.ai_socktype = SOCK_DGRAM;
179     hints.ai_flags = AI_PASSIVE;
180
181     struct addrinfo *result;
182     if (getaddrinfo(argv[1], argv[2], &hints, &result) != 0) {
183       perror("getaddrinfo()");
184       return -1;
185     }
186
187     int sd = socket(result->ai_family, result->ai_socktype, result->ai_protocol);
188     freeaddrinfo(result);
189
190     struct sockaddr_storage client_addr;
191     socklen_t client_addrlen = sizeof(client_addr);
192     sendto(sd, argv[3], 2, 0, result->ai_addr, result->ai_addrlen);
193
194     printf("%s\n", argv[3]);
195     if (argv[3] == 'd' || argv[3] == 't') {
196       char buf[50];
197       ssize_t c = recvfrom(socketUDP, buf, 50, 0, (struct sockaddr *) &client_addr, &
        client_addrlen);
198       buf[c] = '\0';
199       printf("%s\n", buf);
200     }
201
202     return 0;
```

```
203 }
204
205 /* Ejercicio 4. Modificar el servidor para que, además de poder recibir comandos por
    red, los pueda
206 recibir directamente por el terminal, leyendo dos caracteres (el comando y '\n') de
    la entrada
207 estándar. Multiplexar el uso de ambos canales usando select(2). */
208 #include <sys/types.h>
209 #include <sys/socket.h>
210 #include <sys/select.h>
211 #include <arpa/inet.h>
212 #include <netinet/in.h>
213 #include <netdb.h>
214 #include <time.h>
215 #include <unistd.h>
216 #include <stdlib.h>
217 #include <string.h>
218 #include <stdio.h>
219 #include <errno.h>
220
221 int main (int argc, char * argv[]) {
222   if (argc < 3) {
223     printf("Formato incorrecto. Introduce una dirección IPv4 o IPv6 y un puerto como
      comandos\n");
224     return -1;
225   }
226
227   struct addrinfo hints;
228   memset(&hints, 0, sizeof(struct addrinfo));
229   hints.ai_family = AF_UNSPEC;
230   hints.ai_socktype = SOCK_DGRAM;
231   hints.ai_flags = AI_PASSIVE;
232
233   struct addrinfo *result;
234   if (getaddrinfo(argv[1], argv[2], &hints, &result) != 0) {
235       perror("getaddrinfo()");
236       return -1;
237   }
238
239   int sd = socket(result->ai_family, result->ai_socktype, result->ai_protocol);
240   if (bind(sd, result->ai_addr, result->ai_addrlen) != 0) {
241     perror("bind()");
242     return -1;
243   }
244   freeaddrinfo(result);
245
246   char comando[2];
247   char host[NI_MAXHOST];
248   char serv[NI_MAXSERV];
249
250   struct sockaddr_storage client_addr;
251   socklen_t client_addrlen = sizeof(client_addr);
252
253   fd_set dflectura;
254   int df = -1;
255
256   while(1) {
257     while (df == -1) {
258       FD_ZERO(&dflectura);
259       FD_SET(sd, &dflectura);
260       FD_SET(0, &dflectura);
261       df = select(sd + 1, &dflectura, NULL, NULL, NULL);
262     }
263
264     time_t tiempo = time(NULL);
265     struct tm *tm = localtime(&tiempo);
266     char buf[50];
267
268     if (FD_ISSET(sd, &dflectura)) {
269       ssize_t c = recvfrom(sd, comando, 2, 0, (struct sockaddr *) &client_addr, &
        client_addrlen);
270       getnameinfo((struct sockaddr *) &client_addr, client_addrlen, host, NI_MAXHOST,
        serv, NI_MAXSERV, NI_NUMERICHOST|NI_NUMERICSERV);
```

```
271         printf("[RED] %i byte(s) de %s:%s\n", c, host, serv);
272         comando[1] = '\0';
273 
274         switch(comando[0]) {
275           case 't':
276             size_t bytes = strftime(buf, 49, "%I:%M:%S %p", tm);
277             buf[bytes] = '\0';
278             sendto(sd, buf, bytes, 0, (struct sockaddr *) &client_addr, client_addrlen);
279             break;
280 
281           case 'd':
282             size_t bytes = strftime(buf, 49, "%Y-%m-%d", tm);
283             buf[bytes] = '\0';
284             sendto(sd, buf, bytes, 0, (struct sockaddr *) &client_addr, client_addrlen);
285             break;
286 
287           case 'q':
288             printf("Terminado el proceso servidor...\n");
289             return 1;
290 
291           default:
292             printf("Comando no soportado.\n");
293             break;
294         }
295       }
296 
297       else {
298         read(0, comando, 2);
299         printf("[Consola] %i byte(s)\n", 2);
300         comando[1] = '\0';
301 
302         switch(comando[0]) {
303           case 't':
304             size_t bytes = strftime(buf, 49, "%I:%M:%S %p", tm);
305             buf[bytes] = '\0';
306             printf("%s\n", buf);
307             break;
308 
309           case 'd':
310             size_t bytes = strftime(buf, 49, "%Y-%m-%d", tm);
311             buf[bytes] = '\0';
312             printf("%s\n", s);
313             break;
314 
315           case 'q':
316             printf("Terminado el proceso servidor...\n");
317             return 1;
318 
319           default:
320             printf("Comando no soportado.\n");
321             break;
322         }
323       }
324     }
325 
326     return 1;
327   }
328 
329   /* Ejercicio 5. Convertir el servidor UDP en multi-proceso siguiendo el patrón
      pre-fork. Una vez
330   asociado el socket a la dirección local con bind(2), crear varios procesos que llamen
      a recvfrom(2) de
331   forma que cada uno atenderá un mensaje de forma concurrente. Imprimir el PID del
      proceso servidor para
332   comprobarlo. Para terminar el servidor, enviar la señal SIGTERM al grupo de procesos.
      */
333   #include <sys/types.h>
334   #include <sys/socket.h>
335   #include <arpa/inet.h>
336   #include <netinet/in.h>
337   #include <netdb.h>
338   #include <time.h>
339   #include <stdlib.h>
```

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

int main (int argc, char * argv[]) {
 if (argc < 3) {
    printf("Formato incorrecto. Introduce una dirección IPv4 o IPv6 y un puerto como comandos\n");
    return -1;
  }

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

  struct addrinfo *result;
  if (getaddrinfo(argv[1], argv[2], &hints, &result) != 0) {
    perror("getaddrinfo()");
    return -1;
  }

  int sd = socket(result->ai_family, result->ai_socktype, result->ai_protocol);
  if (bind(sd, result->ai_addr, result->ai_addrlen) != 0) {
    perror("bind()");
    return -1;
  }
  freeaddrinfo(result);
  signal(SIGCHLD, hler);

  char comando[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 = fork();
    if (pid == 0) {
      while(1) {
        ssize_t c = recvfrom(sd, comando, 2, 0, (struct sockaddr *) &client_addr, &client_addrlen);
        comando[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);
        char buf[50];

        switch(comando[0]) {
          case 't':
            size_t bytes = strftime(buf, 49, "%I:%M:%S %p", tm);
            buf[bytes] = '\0';
            sendto(sd, buf, bytes, 0, (struct sockaddr *) &client_addr, client_addrlen);
            break;

          case 'd':
            size_t bytes = strftime(buf, 49, "%Y-%m-%d", tm);
            buf[bytes] = '\0';
            sendto(sd, buf, bytes, 0, (struct sockaddr *) &client_addr, client_addrlen);
            break;

          case 'q':
```

```
408               printf("Terminado el proceso servidor...\n");
409               return 1;
410
411             default:
412               printf("Comando no soportado.\n");
413               break;
414           }
415       }
416     }
417
418     else {
419       pid = wait(&status);
420     }
421   }
422
423   return 1;
424 }
425
426 /* Ejercicio 6. Crear un servidor TCP de eco que escuche por conexiones entrantes en una dirección
427 (IPv4 o IPv6) y puerto dados. Cuando reciba una conexión entrante, debe mostrar la dirección y número de
428 puerto del cliente. A partir de ese momento, enviará al cliente todo lo que reciba desde el
429 mismo (eco). Comprobar su funcionamiento empleando la herramienta Netcat como cliente. Comprobar qué
430 sucede si varios clientes intentan conectar al mismo tiempo. */
431 #include <sys/types.h>
432 #include <sys/socket.h>
433 #include <netdb.h>
434 #include <stdlib.h>
435 #include <string.h>
436 #include <stdio.h>
437 #include <errno.h>
438
439 int main(int argc, char * argv[]) {
440   if (argc < 3) {
441     printf("Formato incorrecto. Introduce una dirección IPv4 o IPv6 y un puerto como comandos.\n");
442     return -1;
443   }
444
445   struct addrinfo hints;
446   memset(&hints, 0, sizeof(struct addrinfo));
447   hints.ai_family = AF_UNSPEC;
448   hints.ai_socktype = SOCK_STREAM;
449   hints.ai_flags = AI_PASSIVE;
450
451   struct addrinfo *result;
452   if (getaddrinfo(argv[1], argv[2], &hints, &result) != 0) {
453     perror("getaddrinfo()");
454     return -1;
455   }
456
457   int sd = socket(result->ai_family, result->ai_socktype, 0);
458   bind(sd, result->ai_addr, result->ai_addrlen);
459   freeaddrinfo(result);
460   listen(sd, 5);
461
462   struct sockaddr_storage cli;
463   socklen_t clen = sizeof(cli);
464
465   char buf[80];
466   int clisd;
467   char host[NI_MAXHOST];
468   char serv[NI_MAXSERV];
469   ssize_t c;
470
471   while (1) {
472     clisd = accept(sd, (struct sockaddr *) &cli, &clen);
473     while (1) {
474       getnameinfo((struct sockaddr *)&cli, clen, host, NI_MAXHOST, serv, NI_MAXSERV,
          NI_NUMERICHOST);
```

```
475         printf("Conexión desde %s:%s\n", host, serv);
476
477         c = recv(clisd, buf, 79, 0);
478         buf[c] = '\0';
479
480         if ((buf[0] == 'q') && (c == 2)) {
481           printf("Conexión terminada\n");
482           return 1;
483         }
484         send(clisd, buf, c, 0);
485       }
486     }
487
488     close(clisd);
489     return 1;
490   }
491
492   /* Ejercicio 7. Escribir el cliente para conectarse con el servidor del ejercicio anterior. El cliente
493   recibirá la dirección y el puerto del servidor como argumentos y, una vez establecida la conexión con
494   el servidor, le enviará lo que el usuario escriba por teclado. Mostrará en la consola la respuesta
495   recibida desde el servidor. Cuando el usuario escriba el carácter 'Q' como único carácter de una
496   línea, el cliente cerrará la conexión con el servidor y terminará. */
497   #include <sys/types.h>
498   #include <sys/socket.h>
499   #include <netdb.h>
500   #include <stdlib.h>
501   #include <string.h>
502   #include <stdio.h>
503   #include <errno.h>
504
505   int main(int argc, char * argv[]) {
506     if (argc < 3) {
507       printf("Formato incorrecto. Introduce una dirección IPv4 o IPv6 y un puerto como comandos.\n");
508       return -1;
509     }
510
511     struct addrinfo hints;
512     memset(&hints, 0, sizeof(struct addrinfo));
513     hints.ai_family = AF_UNSPEC;
514     hints.ai_socktype = SOCK_STREAM;
515     hints.ai_flags = AI_PASSIVE;
516
517     struct addrinfo *result;
518     if (getaddrinfo(argv[1], argv[2], &hints, &result) != 0) {
519       perror("getaddrinfo()");
520       return -1;
521     }
522
523     int sd = socket(result->ai_family, result->ai_socktype, 0);
524     connect(sd, (struct sockaddr *)result->ai_addr, result->ai_addrlen);
525     freeaddrinfo(result);
526
527     char buf_in[255];
528     char buf_out[255];
529     ssize_t c;
530
531     while (1) {
532       c = read(0, buf_in, 254);
533       buf_in[c] = '\0';
534       send(sd, buf_in, c, 0);
535
536       if ((buf_in[1] == 'q') && (c == 2)) {
537         printf("Conexión terminada\n");
538         return 1;
539       }
540
541       c = recv(sd, buf_out, 254, 0);
542       buf_out[c] = '\0';
```

```
543       printf("[OUT]:%s\n", buf_out);
544     }
545
546     close(sd);
547     return 1;
548   }
549
550   /* Ejercicio 8. Modificar el código del servidor para que acepte varias conexiones
      simultáneas. Cada
551   petición debe gestionarse en un proceso diferente, siguiendo el patrón
      accept-and-fork. El proceso
552   padre debe cerrar el socket devuelto por accept(2). */
553   #include <sys/types.h>
554   #include <sys/socket.h>
555   #include <netdb.h>
556   #include <stdlib.h>
557   #include <string.h>
558   #include <stdio.h>
559   #include <errno.h>
560
561   int main(int argc, char * argv[]) {
562     if (argc < 3) {
563       printf("Formato incorrecto. Introduce una dirección IPv4 o IPv6 y un puerto como
          comandos.\n");
564       return -1;
565     }
566
567     struct addrinfo hints;
568     memset(&hints, 0, sizeof(struct addrinfo));
569     hints.ai_family = AF_UNSPEC;
570     hints.ai_socktype = SOCK_STREAM;
571     hints.ai_flags = AI_PASSIVE;
572
573     struct addrinfo *result;
574     if (getaddrinfo(argv[1], argv[2], &hints, &result) != 0) {
575       perror("getaddrinfo()");
576       return -1;
577     }
578
579     int sd = socket(result->ai_family, result->ai_socktype, 0);
580     bind(sd, result->ai_addr, result->ai_addrlen);
581     freeaddrinfo(result);
582     listen(sd, 5);
583
584     struct sockaddr_storage cli;
585     socklen_t clen = sizeof(cli);
586
587     char buf[80];
588     int clisd;
589     char host[NI_MAXHOST];
590     char serv[NI_MAXSERV];
591     ssize_t c;
592
593     while (1) {
594       clisd = accept(sd, (struct sockaddr *) &cli, &clen);
595       pid_t pid = fork();
596       if (pid == 0) {
597         while (1) {
598           getnameinfo((struct sockaddr *)&cli, clen, host, NI_MAXHOST, serv, NI_MAXSERV,
               NI_NUMERICHOST);
599           printf("[PID: %i] Conexión desde %s:%s\n", getpid(), host, serv);
600           c = recv(clisd, buf, 79, 0);
601           buf[c] = '\0';
602
603           if ((buf[0] == 'q') && (c == 2)) {
604             printf("Conexión terminada\n");
605             return 1;
606           }
607
608           send(clisd, buf, c, 0);
609         }
610         close(clisd);
611       }
```

```
612
613       else {
614         close(clisd);
615       }
616     }
617
618     close(clisd);
619     return 1;
620   }
621
622   /* Ejercicio 9. Añadir la lógica necesaria en el servidor para que no quede ningún
      proceso en estado
623   zombie. Para ello, se deberá capturar la señal SIGCHLD y obtener la información de
      estado de los
624   procesos hijos finalizados. */
625   #include <sys/types.h>
626   #include <sys/socket.h>
627   #include <sys/wait.h>
628   #include <sys/stat.h>
629   #include <netdb.h>
630   #include <signal.h>
631   #include <time.h>
632   #include <fcntl.h>
633   #include <stdlib.h>
634   #include <string.h>
635   #include <stdio.h>
636   #include <errno.h>
637
638   void handler(int signal){
639     pid_t pid;
640     pid = wait(NULL);
641   }
642
643   int main(int argc, char * argv[]) {
644     if (argc < 3) {
645       printf("Formato incorrecto. Introduce una dirección IPv4 o IPv6 y un puerto como
        comandos.\n");
646       return -1;
647     }
648
649     struct addrinfo hints;
650     memset(&hints, 0, sizeof(struct addrinfo));
651     hints.ai_family = AF_UNSPEC;
652     hints.ai_socktype = SOCK_STREAM;
653     hints.ai_flags = AI_PASSIVE;
654
655     struct addrinfo *result;
656     if (getaddrinfo(argv[1], argv[2], &hints, &result) != 0) {
657       perror("getaddrinfo()");
658       return -1;
659     }
660
661     int sd = socket(result->ai_family, result->ai_socktype, 0);
662     bind(sd, result->ai_addr, result->ai_addrlen);
663     freeaddrinfo(result);
664     listen(sd, 5);
665
666     struct sockaddr_storage cli;
667     socklen_t clen = sizeof(cli);
668
669     char buf[80];
670     int clisd;
671     char host[NI_MAXHOST];
672     char serv[NI_MAXSERV];
673     ssize_t c;
674
675     signal(SIGCHLD, handler);
676     int status;
677
678     while (1) {
679       clisd = accept(sd, (struct sockaddr *) &cli, &clen);
680       pid_t pid = fork();
681       if (pid == 0) {
```

```
682         while (1) {
683           getnameinfo((struct sockaddr *)&cli, clen, host, NI_MAXHOST, serv, NI_MAXSERV,
               NI_NUMERICHOST);
684           printf("[PID: %i] Conexión desde %s:%s\n", getpid(), host, serv);
685           c = recv(clisd, buf, 79, 0);
686           buf[c] = '\0';
687 
688           if ((buf[0] == 'q') && (c == 2)) {
689             printf("Conexión terminada\n");
690             return 1;
691           }
692           send(clisd, buf, c, 0);
693         }
694       }
695 
696       else {
697         pid = wait(&status);
698         close(clisd);
699         return 1;
700       }
701     }
702 
703     close(clisd);
704     return 1;
705   }
```