

Práctica 3:

Sockets sin conexión

Programas que envían y reciben usando UDP

Fort Muñoz, Marcelo ; Vidal Villalba, Pedro

19 de noviembre de 2023

Índice

1. Introducción	3
2. Ejercicio 1. Apartado c	4
2.1. Modificaciones introducidas en el código	4
2.2. Resultado obtenido	4
3. Ejercicio 1. Apartado d	6
3.1. Modificaciones introducidas en el código	6
3.2. Resultado obtenido	8
4. Ejercicio 3	9
4.1. Modificaciones introducidas en el código	9
4.2. Resultado obtenido	9

1 Introducción

En este informe se recogerán los distintos comportamientos asociados a los códigos de los apartados (1.c), (1.d) y (3) de la Práctica 3 de la materia de *Redes*.

En el primer ejercicio de esta práctica se han implementado los programas `emisor.c` y `receptor.c`. El primero de ellos se encarga de enviar un mensaje de saludo al receptor, y este debe imprimir el mensaje recibido, así como los bytes que ocupa el mensaje.

En el apartado (1.c) comprobaremos qué ocurre si la función `recvfrom` lee menos datos de los que envió la función `sendto`, viendo si es posible recuperar los datos restantes con una nueva llamada a `recvfrom`.

En el apartado (1.d) modificaremos los programas para que, en vez de transmitir cadenas de texto, se transmitan arrays de número tipo `float`. En este caso, el número de datos enviados se debe especificar únicamente en el programa emisor, es decir, el programa que recibe debe ser capaz de recibir todos los datos enviados y determinar dicho número de datos.

En el segundo ejercicio de la práctica se implementaron también un par de programas `servidorUDP.c` y `clienteUDP.c`. En estos, que son adaptaciones a UDP de los programas análogos de la Práctica 2 ya realizada, el cliente lee un archivo de texto como entrada y se lo envía al servidor, línea a línea, a través de mensajes UDP. El servidor debe pasar los caracteres de la línea a mayúsculas y devolverle al cliente la línea convertida. Por último, el cliente va recibiendo las líneas y las va escribiendo en un archivo de salida, que tendrá el mismo nombre que el archivo de entrada pero todo en mayúsculas.

En el tercer ejercicio se debe comprobar que el servidor programado en el ejercicio (2) puede atender a varios clientes simultáneamente.

2 Ejercicio 1. Apartado c

Se nos ha solicitado comprobar qué ocurre si la función `recvfrom` lee menos datos que los que envió la función `sendto` y si es posible recuperar los datos restantes con una nueva llamada a `recvfrom`.

2.1 *Modificaciones introducidas en el código*

Para poder realizar esta prueba hemos modificado el código con los siguientes cambios:

- Hemos añadido la posibilidad de añadir como parámetro por la línea de comandos el tamaño máximo en bytes a leer desde el receptor con `recvfrom`.
- Hemos añadiendo una comprobación en la que, si el total de bytes coincide con el tamaño máximo solicitado, realizamos inmediatamente una segunda llamada a `recvfrom` por si pudiéramos de esa forma recuperar la información enviada.

2.2 *Resultado obtenido*

Para determinar qué sucedería en este caso hemos indicado al programa que queremos recibir sólo 10 bytes y le hemos enviado un mensaje de mayor longitud.

El resultado ha sido que el primer `recvfrom` ha recibido los 10 primeros bytes y en el segundo no hemos conseguido recibir nada.

Con esta prueba hemos determinado que no se pueden recuperar los datos restantes, puesto que, en la segunda llamada a `recvfrom` el sistema nos da un error indicando que no queda ningún dato pendiente.

En la Figura 1 podemos ver un ejemplo de ejecución en el receptor.

```

usuario@mun12:~/Repos/Redes-P3/basic$ ./receptor -b 10
Host creado con éxito.
Hostname: mun12; IPs v4 locales:127.0.0.1, 192.168.68.143; IPs v6 locales: ::1, fe80::4197:9b46:9636:3332%enp0s17; Puerto: 8200; IP pública: 93.156.219.0
IPs v4 del receptor : 127.0.0.1, 192.168.68.143
IPs v6 del receptor : ::1, fe80::4197:9b46:9636:3332%enp0s17
Puerto del receptor : 8200 UDP
IP pública del receptor : 93.156.219.0
Máximo de bytes a leer : 10 (apartado c)

=====
Escuchando en el puerto : 8200 UDP...

! Recibida señal 29 (SIGIO)
=====
Posible mensaje recibido...
=====
Mensaje recibido : "El host mu"
Bytes recibidos : 10
IP del emisor : 192.168.68.122
Puerto del emisor : 8100 UDP
Como hemos recibido el máximo de bytes (10), es posible que haya más datos pendientes de recibir.
Volvamos a llamar (por si acaso) de nuevo a recvfrom()...

Falsa alarma, no había mensajes pendientes o se recibió una señal de terminación
=====

```

Figura 1: *Apartado (1.c)* - Ejecución del cliente leyendo menos datos de los que se envían

3 Ejercicio 1. Apartado d

En este apartado, partiendo del anterior, modificamos los programas para que, en lugar de transmitir cadenas de texto envíen un array de números de tipo `float`. El número de datos deberá ser especificado únicamente en el programa que envía, mientras que el programa receptor debe ser capaz de recibir todos los datos enviados y determinar dicho número.

3.1 Modificaciones introducidas en el código

Para lograr esto, se introdujo en **emisor.c** una nueva opción por línea de comandos para poder indicar el número máximo de bytes a enviar. Nótese que como el tamaño de un `float` es de 4 bytes, el tamaño del array enviado será 4 veces menor que el número especificado por terminal.

Además de esto, se modificó la función `send_message`, que envía el mensaje del emisor al receptor, haciendo que ahora acepte un parámetro adicional, que es el número máximo de bytes a enviar (análogo a lo que se hizo en el apartado anterior para la recepción), y se modificó por supuesto el cuerpo de la función para que ahora el array del mensaje a enviar fuese de `float` en lugar de `char`. Se añadió también código para rellenar ese array con número pseudoaleatorios, en lugar de con un mensaje de saludo, y se adaptó la salida por terminal de forma acorde.

Así queda la función `send_message` tras los cambios (las modificaciones en el procesado de argumentos se obviarán de este informe por carecer de relevancia en las competencias propias de esta práctica):

```
1 static void send_message(Host *sender, Host *remote, size_t
   max_bytes_to_send) {
2     float message_to_send[max_bytes_to_send / sizeof(float)];
3     ssize_t sent_bytes;
4     int i;
5
6     log_and_stdout_printf(sender->log, "Puerto del emisor   : %d
       UDP\n", sender->port);
7     log_and_stdout_printf(sender->log, "IP       del receptor : %s\n
       ", remote->ip);
8     log_and_stdout_printf(sender->log, "Puerto del receptor : %d
       UDP\n", remote->port);
9
10    for (i = 0; i < max_bytes_to_send / sizeof(float) + 1; i++) {
11        message_to_send[i] = drand48();
12    }
```

```

13
14 // Enviamos el mensaje al cliente
15 sent_bytes = sendto(sender->socket, message_to_send,
16                     max_bytes_to_send, /*__flags*/ 0, (struct sockaddr *) &
17                     remote->address, sizeof(remote->address));
18
19 if (sent_bytes == -1) {
20     log_printf_err(sender->log, "ERROR: Se produjo un error
21     cuando se intentaba enviar el mensaje\n");
22     fail("ERROR: Se produjo un error cuando se intentaba
23     enviar el mensaje");
24 }
25
26 log_and_stdout_printf(sender->log, "Mensaje enviado      : ");
27 for (i = 0; i < max_bytes_to_send / sizeof(float); i++) {
28     printf("%f; ", message_to_send[i]);
29 }
30 printf("\b\b  \n");
31 log_and_stdout_printf(sender->log, "Bytes enviados      : %ld\n",
32 sent_bytes);
33 }

```

En **receptor.c**, los únicos cambios introducidos fueron en la función `handle_message`, que ahora, claro está, debe tener `float` como tipo de datos para el array en el que guardar el mensaje recibido. Ese, y la introducción de un bucle para mostrar los datos leídos en lugar de imprimir directamente la `string`, como sí se podía hacer en los anteriores apartados, fueron los únicos cambios introducidos. Así queda la función `handle_message` tras los cambios:

```

1 static ssize_t handle_message(Host *self, size_t max_bytes_to_read
2 ) {
3     float received_message[max_bytes_to_read + 1];
4     struct sockaddr_in remote_connection_info;
5     ssize_t received_bytes;
6     socklen_t addr_len = sizeof(struct sockaddr_in);
7     int i;
8
9     received_bytes = recvfrom(self->socket, received_message,
10                             max_bytes_to_read, 0, (struct sockaddr *) &(
11                             remote_connection_info), &addr_len);
12
13     if (received_bytes == -1) {
14         if (errno == EAGAIN || errno == EWOULDBLOCK) { /* Hemos
15             marcado al socket con O_NONBLOCK; no hay mensajes
16             pendientes, así que lo registramos y salimos */
17             socket_io_pending = 0;
18             return received_bytes;
19         }
20     }
21     log_printf_err(self->log, "ERROR: Se produjo un error en

```

```

16         la recepcion del mensaje\n");
17     fail("ERROR: Se produjo un error en la recepcion del
18         mensaje");
19 }
20 log_and_stdout_printf(self->log, "Mensaje recibido : ");
21 for (i = 0; i < received_bytes / sizeof(float); i++) {
22     printf("%f; ", received_message[i]);
23 }
24 printf("\b\b \n");
25 log_and_stdout_printf(self->log, "Bytes recibidos : %ld\n",
26     received_bytes);
27 log_and_stdout_printf(self->log, "IP del emisor : %s\n",
28     inet_ntoa(remote_connection_info.sin_addr));
29 log_and_stdout_printf(self->log, "Puerto del emisor : %d UDP\n",
30     ntohs(remote_connection_info.sin_port));
31
32 log_and_stdout_printf(self->log, "
33     -----\n");
34
35 return received_bytes;
36 }

```

3.2 Resultado obtenido

Podemos ver en la Figura 2 el resultado obtenido, y cómo efectivamente se envían correctamente los datos del array de floats.

```

pedro@pedro-GL65-95EK: ~/OneDrive/Tercero/Primer_Cuatr/Redes/Practicas/P3/basic
$ ./emisor -o 7200 -i localhost -p 8120 -b 80
Host creado con éxito.
Hostname: pedro-GL65-95EK; IP: 83.35.183.160; Puerto: 7200

[Sun, 19 Nov 2023, 21:11:18.602851; PID=6270] Puerto del emisor : 7200 UDP
[Sun, 19 Nov 2023, 21:11:18.602887; PID=6270] IP del receptor : 127.0.0.1
[Sun, 19 Nov 2023, 21:11:18.602922; PID=6270] Puerto del receptor : 8120 UDP
[Sun, 19 Nov 2023, 21:11:18.603008; PID=6270] Mensaje enviado : 0.000000; 0.000985; 0.0
41631; 0.176643; 0.364602; 0.691331; 0.692298; 0.487217; 0.526750; 0.454433; 0.233178; 0.83
1292; 0.931731; 0.568060; 0.556094; 0.050832; 0.767051; 0.018915; 0.252360; 0.298197
[Sun, 19 Nov 2023, 21:11:18.603082; PID=6270] Bytes enviados : 80
pedro@pedro-GL65-95EK: ~/OneDrive/Tercero/Primer_Cuatr/Redes/Practicas/P3/basic
$

pedro@pedro-GL65-95EK: ~/OneDrive/Tercero/Primer_Cuatr/Redes/Practicas/P3/basic
$ ./receptor -p 8120
Host creado con éxito.
Hostname: pedro-GL65-95EK; IP: 83.35.183.160; Puerto: 8120

[Sun, 19 Nov 2023, 21:11:18.602355; PID=6269] -----
[Sun, 19 Nov 2023, 21:11:18.602382; PID=6269] Máximo de bytes a leer: 1000
[Sun, 19 Nov 2023, 21:11:18.602388; PID=6269] Escuchando en el puerto 8120 UDP...
[Sun, 19 Nov 2023, 21:11:18.603072; PID=6269] Mensaje recibido : 0.000000; 0.000985; 0.041
631; 0.176643; 0.364602; 0.691331; 0.692298; 0.487217; 0.526750; 0.454433; 0.233178; 0.8312
92; 0.931731; 0.568060; 0.556094; 0.050832; 0.767051; 0.018915; 0.252360; 0.298197
[Sun, 19 Nov 2023, 21:11:18.603197; PID=6269] Bytes recibidos : 80
[Sun, 19 Nov 2023, 21:11:18.603267; PID=6269] IP del emisor : 127.0.0.1
[Sun, 19 Nov 2023, 21:11:18.603324; PID=6269] Puerto del emisor : 7200 UDP
[Sun, 19 Nov 2023, 21:11:18.603375; PID=6269] -----
Cerrando el receptor y saliendo...

```

Figura 2: Apartado (1.d) - Envío y recepción de un array de floats

4 Ejercicio 3

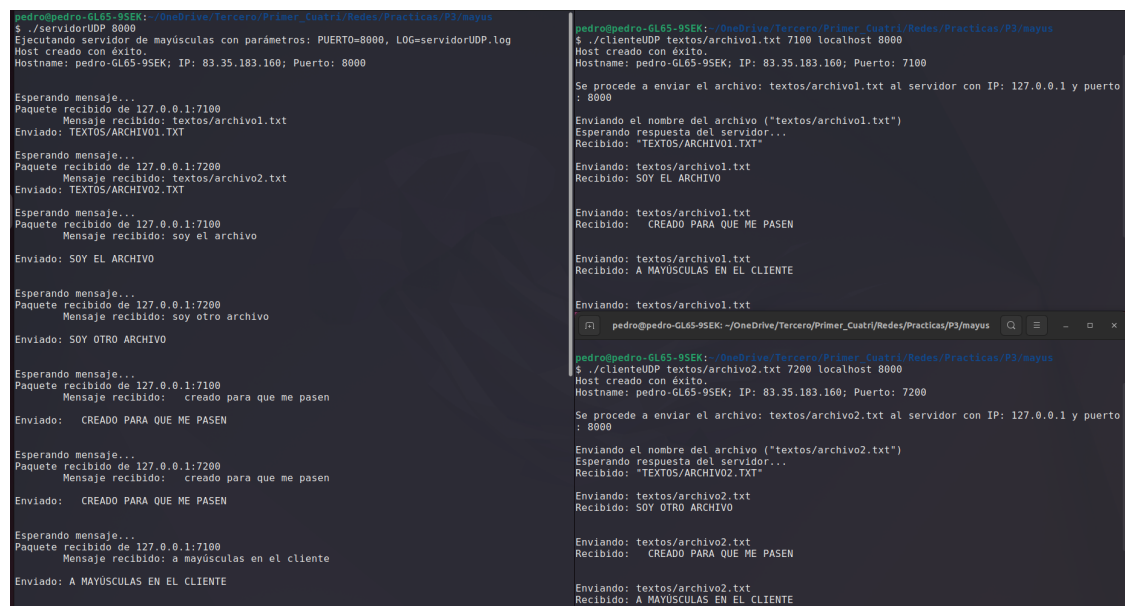
En este ejercicio debemos comprobar que el servidor de mayúsculas programado en el ejercicio (2) puede atender a varios clientes **simultáneamente**.

4.1 Modificaciones introducidas en el código

Para ver esto, simplemente debemos modificar el lazo del cliente de mayúsculas en el que se van leyendo líneas del archivo, introduciendo un `sleep` para que dé tiempo a lanzar un segundo cliente desde otra terminal.

4.2 Resultado obtenido

En la Figura 3 se puede ver cómo al ejecutar los programas con este cambio, un servidor y dos clientes conectándose a él de forma simultánea, en efecto el servidor es capaz de atender los mensajes de ambos de forma simultánea.



```
pedro@pedro-GL65-95EK: ~/OneDrive/Tercero/Primer_Cuatril/Redes/Practicas/P3/mayus
$ ./servidorUDP 8000
Ejecutando servidor de mayúsculas con parámetros: PUERTO=8000, LOG=servidorUDP.log
Host creado con éxito.
Hostname: pedro-GL65-95EK; IP: 83.35.183.160; Puerto: 8000

Esperando mensaje...
Paquete recibido de 127.0.0.1:7100
Mensaje recibido: textos/archivo1.txt
Enviado: TEXTOS/ARCHIVO1.TXT

Esperando mensaje...
Paquete recibido de 127.0.0.1:7200
Mensaje recibido: textos/archivo2.txt
Enviado: TEXTOS/ARCHIVO2.TXT

Esperando mensaje...
Paquete recibido de 127.0.0.1:7100
Mensaje recibido: soy el archivo
Enviado: SOY EL ARCHIVO

Esperando mensaje...
Paquete recibido de 127.0.0.1:7200
Mensaje recibido: soy otro archivo
Enviado: SOY OTRO ARCHIVO

Esperando mensaje...
Paquete recibido de 127.0.0.1:7100
Mensaje recibido: creado para que me pasen
Enviado: CREADO PARA QUE ME PASEN

Esperando mensaje...
Paquete recibido de 127.0.0.1:7200
Mensaje recibido: creado para que me pasen
Enviado: CREADO PARA QUE ME PASEN

Esperando mensaje...
Paquete recibido de 127.0.0.1:7100
Mensaje recibido: a mayúsculas en el cliente
Enviado: A MAYÚSCULAS EN EL CLIENTE

pedro@pedro-GL65-95EK: ~/OneDrive/Tercero/Primer_Cuatril/Redes/Practicas/P3/mayus
$ ./clienteUDP textos/archivo1.txt 7100 localhost 8000
Host creado con éxito.
Hostname: pedro-GL65-95EK; IP: 83.35.183.160; Puerto: 7100

Se procede a enviar el archivo: textos/archivo1.txt al servidor con IP: 127.0.0.1 y puerto : 8000

Enviando el nombre del archivo ("textos/archivo1.txt")
Esperando respuesta del servidor...
Recibido: "TEXTOS/ARCHIVO1.TXT"

Enviando: textos/archivo1.txt
Recibido: SOY EL ARCHIVO

Enviando: textos/archivo1.txt
Recibido: CREADO PARA QUE ME PASEN

Enviando: textos/archivo1.txt
Recibido: A MAYÚSCULAS EN EL CLIENTE

pedro@pedro-GL65-95EK: ~/OneDrive/Tercero/Primer_Cuatril/Redes/Practicas/P3/mayus
$ ./clienteUDP textos/archivo2.txt 7200 localhost 8000
Host creado con éxito.
Hostname: pedro-GL65-95EK; IP: 83.35.183.160; Puerto: 7200

Se procede a enviar el archivo: textos/archivo2.txt al servidor con IP: 127.0.0.1 y puerto : 8000

Enviando el nombre del archivo ("textos/archivo2.txt")
Esperando respuesta del servidor...
Recibido: "TEXTOS/ARCHIVO2.TXT"

Enviando: textos/archivo2.txt
Recibido: SOY OTRO ARCHIVO

Enviando: textos/archivo2.txt
Recibido: CREADO PARA QUE ME PASEN

Enviando: textos/archivo2.txt
Recibido: A MAYÚSCULAS EN EL CLIENTE
```

Figura 3: *Ejercicio (3)* - Ejecución simultánea de un servidor de mayúsculas y dos clientes

Como debería ocurrir, el servidor comienza a ejecutarse y espera por mensajes que pasar a mayúsculas y reenviar a quien se lo haya enviado. Entonces, el primer cliente procede a enviarle las líneas del archivo que quiere convertir, y el servidor las va pasando a mayúsculas según le van llegando.

Paralelamente, cuando se ejecuta el segundo cliente, este también va enviando las líneas que él lee de su archivo, y el servidor las va transformando a mayúsculas y reenviándolas a medida que las recibe.

A diferencia de lo que pasaba con los cliente y servidor en TCP, al ser UDP no orientado a conexión, cada parte envía los mensajes según los va teniendo, sin necesidad de conectarse ni esperar por la otra parte salvo en los momentos en los que la lógica del programa lo requiera (por ejemplo, los clientes deben de esperar por la respuesta del servidor para ir escribiendo las líneas en el archivo de mayúsculas en orden).

Así, hemos verificado que el servidor con UDP es efectivamente capaz de manejar simultáneamente varias peticiones que lleguen a la vez desde distintos clientes.