

# Práctica 2 - Informe: Sockets orientados a conexión. Servidor y cliente TCP.

Redes

Xiana Carrera Alonso

Noviembre 2021

# Índice

<b>1. Cliente/servidor básicos</b>	<b>3</b>
1.c. Dos mensajes con un único <i>recv()</i> . . . . .	3
1.d. Bucle con número de bytes a recibir . . . . .	7
<b>3. Múltiples clientes secuenciales en servidor de mayúsculas</b>	<b>11</b>
<b>Anexo</b>	<b>17</b>

## 1. Cliente/servidor básicos

### 1.c. Dos mensajes con un único *recv()*

En este apartado se describe el comportamiento observado cuando el servidor básico (servidor.c) envía dos mensajes en dos funciones *send()* distintas, y el cliente los recibe llamando una única vez a *recv()*.

Definimos pues una nueva variable en el programa servidor para guardar el segundo mensaje a enviar, que explicitamos directamente (línea 2):

```
1 char mensaje1 [] = "Hey! Listen!";      // Primer mensaje que se enviará
2 char mensaje2 [] = "¡Hola desde el otro lado!"; // Segundo mensaje
```

El resto de cambios en su código se producen en el envío de datos, que quedaría:

```
1 //***** ENVÍO DE DATOS *****/
2
3 /*
4  * Enviamos el primer mensaje
5  * Guardamos en nbytes el número de bytes transmitidos
6  * Si send() no logra enviar todos los bytes, repetimos el envío con
7  * la parte del mensaje que aún no se haya transmitido.
8 */
9 i = 0;
10 while ((nbytes = enviar(sockcon, &mensaje1[i])) != strlen(&mensaje1[i]) + 1){
11     i += nbytes;
12 }
14
15 // Aunque el formato estándar para ssize_t es %zd, los sistemas
16 // antiguos no lo soportan. Por seguridad, realizamos un cast a long
17 printf("***** Envío 1 *****\n");
18 printf("\tSe han enviado %ld bytes.\n", (long) nbytes);
19 printf("\tLongitud del mensaje original: %ld bytes\n",
20       strlen(mensaje1) + 1);
21 printf("\tMensaje original: %s\n\n", mensaje1);
22 /*
23  * Como sizeof(char) es siempre 1, strlen(mensaje) y
24  * strlen(mensaje) * sizeof(char) son equivalentes. Añadimos
25  * 1 byte extra por el carácter nulo, que siempre enviamos.
26 */
27
28 // Enviamos el segundo mensaje al mismo cliente
29 i = 0;
30 while ((nbytes = enviar(sockcon, &mensaje2[i])) != strlen(&mensaje2[i]) + 1){
31     i += nbytes;
32 }
34
35 printf("***** Envío 2 *****\n");
36 printf("\tSe han enviado %ld bytes.\n", (long) nbytes);
37 printf("\tLongitud del mensaje 2 original: %ld bytes\n",
38       strlen(mensaje2) + 1);
39 printf("\tMensaje 2 original: %s\n\n", mensaje2);
```

En primer lugar, se probó a ejecutar los programas introduciendo una pequeña espera de 3 segundos usando *sleep()* después de que el cliente (*cliente.c*) establezca la conexión con el servidor y antes de que llame a *recv()*. Se muestra a continuación cómo quedaría el código de *cliente.c* con respecto a la recepción de mensajes, que es donde se producen todos los cambios (sin contar la adición de la librería *unistd.h* para usar *sleep()*).

```

1  ****RECEPCIÓN DEL MENSAJE*****
2
3 // 1c) Esperamos 3 s para que al servidor le de tiempo a enviar
4 // ambos mensajes
5 sleep(3);
6
7 /*
8 * Por si el mensaje que envía el servidor no acaba en '\0', llenamos
9 * el buffer con '\0', de forma que aseguramos que siempre se encuentre
10 * el final de la cadena. Además, al tener tamaño N + 1, habrá '\0'
11 * incluso si se reciben N bytes, el máximo especificado.
12 */
13 memset(&buffer[0], '\0', sizeof(buffer));
14
15 /*
16 * Llamamos a recv(), que espera a que lleguen datos mientras sockcli,
17 * el socket de la conexión, continúa abierto (ya que indicamos el
18 * comportamiento por defecto).
19 * El número de bytes recibidos se guarda en res_recv.
20 */
21 res_recv = recibir(sockcli, &buffer[0]);
22
23 printf(" Ha llegado un mensaje del servidor!\n");
24 // Aunque el formato estándar para ssize_t es %zd, los sistemas
25 // antiguos no lo soportan. Por seguridad, realizamos un cast a long
26 printf("\tNúmero de bytes recibidos: %d\n", (long) res_recv);
27
28
29 // El mensaje que envíe el servidor puede contener caracteres nulos
30 // intermedios. Para asegurar que lo mostramos en su totalidad,
31 // llevamos la cuenta del número de bytes leídos, teniendo en cuenta
32 // que sizeof(char) = 1.
33 printf("\tMensaje: ");
34 leido = 0;
35 while (leido < res_recv){
36     frase = &buffer[leido]; // Nos colocamos en el primer carácter sin leer
37     printf("%s\n\t", frase); // Imprimimos hasta encontrar '\0'
38
39     // Aumentamos leido para que apunte al carácter siguiente a '\0'.
40     // Nótese que strlen() no cuenta '\0', de modo que tenemos que
41     // debemos sumar 1 a la longitud de la cadena.
42     // Aunque el mensaje recibido no termine en '\0', lo hemos añadido
43     // artificialmente, de forma que estas operaciones tienen sentido.
44     leido += strlen(frase) + 1;
45 }
```

Con esta implementación, los dos mensajes enviados por el servidor quedarán disponibles en el socket de la conexión. Cuando el cliente despierte y ejecute *recv()* (a través de *recibir()*),

la función comprobará que en el socket hay datos. En consecuencia, devolverá todos los bytes que pueda, hasta alcanzar el máximo que le fue establecido a través del parámetro *len*:

```

1  /*
2   * Función que encapsula recv() y su gestión de errores.
3   * Es utilizada por el servidor y por el cliente.
4   *
5   * Se reciben datos pasados a través de la conexión establecida sobre el socket
6   * de identificador sockcon. Se intenta recibir todos los datos que contiene el
7   * mensaje, teniendo en cuenta que se ha impuesto un límite de longitud igual a
8   * N (macro definida en lib.h). La función devuelve el número de bytes
9   * recibidos.
10  * En caso de error, se para la ejecución y se imprime un mensaje de error.
11  *
12  * Parámetros:
13  * - sockcon -> Entrada. Identificador del socket de la conexión.
14  * - buffer -> Salida. Puntero al buffer donde se guardará el mensaje recibido.
15  */
16 ssize_t recibir(int sockcon, char * buffer){
17     ssize_t res_recv;           // Número de bytes recibidos
18
19     /*
20      * Llamamos a recv() con argumentos:
21      * - el entero identificador del socket de la conexión
22      * - un puntero al buffer donde guardar los datos
23      * - el máximo de bytes a recibir. Como no sabemos cómo será el mensaje,
24      *     indicamos el tamaño del array, N
25      * - las opciones predeterminadas (0)
26      */
27     if ((res_recv = recv(sockcon, (void *) buffer, (size_t) N, 0)) < 0){
28         perror("Error en la recepción de datos del servidor");
29         exit(EXIT_FAILURE);
30     } else if (res_recv == 0){
31         // No se ha recibido ningún byte o el socket se ha cerrado
32         printf("\nNo se han recibido datos\n");
33     }
34
35     return res_recv;
36 }
```

En este caso, pasamos N, definido como 1000, como tercer argumento de *recv()*. Dado que el tamaño conjunto de los dos mensajes es muy inferior ( $13 + 27 = 40$  bytes), ambos serán devueltos, sin tener en cuenta que se hayan enviado de forma separada (se realiza el vaciado del buffer del socket sin tener en cuenta parámetros adicionales).

Nótese que a la hora de leer los mensajes es importante el bucle *while()*, pues si simplemente tuviéramos *printf("%s\n", buffer)*, la cadena solo se leería hasta alcanzar el primer carácter nulo, '\0', correspondiente al primer mensaje. En la figura 1 se muestra la salida observada en este caso. Vemos que el número de bytes del resultado de *recv()* es la suma de los bytes de los dos mensajes, 40. En *buffer* habría entonces guardados ambos textos, uno después del otro y separados por el carácter nulo del primero.

Tendríamos varias opciones para corregir este defecto: no enviar el último byte de cada mensaje, correspondiente al carácter nulo, en servidor.c, o incluir una comprobación en cliente.c

```
xiana@xiana-Lenovo-V130-15IKB:~/Escritorio/Redes/Repositorio/Practicas/Practica_2$ ./servidor 8888
Se ha establecido una conexión con el cliente con:
  IP: 127.0.0.1
  Puerto: 41762
***** Envío 1 *****
  Se han enviado 13 bytes.
  Longitud del mensaje original: 13 bytes
  Mensaje original: Hey! Listen!
***** Envío 2 *****
  Se han enviado 27 bytes.
  Longitud del mensaje 2 original: 27 bytes
  Mensaje 2 original: ¡Hola desde el otro lado!
^C
xiana@xiana-Lenovo-V130-15IKB:~/Escritorio/Redes/Repositorio/Practicas/Practica_2$ 
```

```
xiana@xiana-Lenovo-V130-15IKB:~/Escritorio/Redes/Repositorio/Practicas/Practica_2$ ./cliente 127.0.0.1 8888
Se ha establecido la conexión con el servidor de:
  IP: 127.0.0.1
  Puerto: 8888
¡Ha llegado un mensaje del servidor!
  Número de bytes recibidos: 40
  Mensaje: Hey! Listen!
Cerrando cliente...
xiana@xiana-Lenovo-V130-15IKB:~/Escritorio/Redes/Repositorio/Practicas/Practica_2$ 
```

Figura 1: Impresión de los dos mensajes recibidos sin tener en cuenta el carácter nulo que los separa. Aunque se reciben 40 bytes (mensajes 1 y 2), solo se muestran 13 (mensaje 1).

que permita varios caracteres '\0' en los datos recibidos, que es lo que se ha elegido.

Hecha dicha sustitución, que se puede ver en el código mostrado en la página 4, la salida pasará a mostrar ambos mensajes sin separación alguna entre ellos. No obstante, para mayor claridad, construimos el bucle de forma que se muestren los dos textos en líneas distintas indentadas al imprimirlos por pantalla:

```
xiana@xiana-Lenovo-V130-15IKB:~/Escritorio/Redes/Repositorio/Practicas/Practica_2$ ./servidor 8888
Se ha establecido una conexión con el cliente con:
  IP: 127.0.0.1
  Puerto: 41936
***** Envío 1 *****
  Se han enviado 13 bytes.
  Longitud del mensaje original: 13 bytes
  Mensaje original: Hey! Listen!
***** Envío 2 *****
  Se han enviado 27 bytes.
  Longitud del mensaje 2 original: 27 bytes
  Mensaje 2 original: ¡Hola desde el otro lado!
^C
xiana@xiana-Lenovo-V130-15IKB:~/Escritorio/Redes/Repositorio/Practicas/Practica_2$ 
```

```
xiana@xiana-Lenovo-V130-15IKB:~/Escritorio/Redes/Repositorio/Practicas/Practica_2$ ./cliente 127.0.0.1 8888
Se ha establecido la conexión con el servidor de:
  IP: 127.0.0.1
  Puerto: 8888
¡Ha llegado un mensaje del servidor!
  Número de bytes recibidos: 40
  Mensaje: Hey! Listen!
  ¡Hola desde el otro lado!
Cerrando cliente...
xiana@xiana-Lenovo-V130-15IKB:~/Escritorio/Redes/Repositorio/Practicas/Practica_2$ 
```

Figura 2: Recepción de dos mensajes con único *recv()* con una espera antes de su ejecución. Nótese que la línea e indentación que los separa no forman parte del envío recibido.

Si eliminamos la línea *sleep(3)*, el comportamiento resultante es impredecible. Dado que el cliente y el servidor se ejecutan a la vez, puede ocurrir que *recv()* recoja los datos que hay en el socket sin que el servidor haya llegado a ejecutar el segundo *send()*, de forma que solo el primer mensaje llega al cliente. No obstante, en otras ocasiones el servidor podría ser más rápido y sí llegar a realizar ambos envíos antes de que actúe *recv()*. En este caso, ambos mensajes serían recibidos y el resultado sería el mismo que incluyendo la llamada a *sleep()*.

```
xiana@xiana-Lenovo-V130-15IKB:~/Escritorio/Redes/Repositorio/Practicas/Practica_2$ ./servidor 8888
Se ha establecido una conexión con el cliente con:
  IP: 127.0.0.1
  Puerto: 41978
***** Envío 1 *****
  Se han enviado 13 bytes.
  Longitud del mensaje original: 13 bytes
  Mensaje original: Hey! Listen!
***** Envío 2 *****
  Se han enviado 27 bytes.
  Longitud del mensaje 2 original: 27 bytes
  Mensaje 2 original: ¡Hola desde el otro lado!
Se ha establecido una conexión con el cliente con:
  IP: 127.0.0.1
  Puerto: 41980
***** Envío 1 *****
  Se han enviado 13 bytes.
  Longitud del mensaje original: 13 bytes
  Mensaje original: Hey! Listen!
***** Envío 2 *****
  Se han enviado 27 bytes.
  Longitud del mensaje 2 original: 27 bytes
  Mensaje 2 original: ¡Hola desde el otro lado!
^C
xiana@xiana-Lenovo-V130-15IKB:~/Escritorio/Redes/Repositorio/Practicas/Practica_2$ 
```

```
xiana@xiana-Lenovo-V130-15IKB:~/Escritorio/Redes/Repositorio/Practicas/Practica_2$ ./cliente 127.0.0.1 8888
Se ha establecido la conexión con el servidor de:
  IP: 127.0.0.1
  Puerto: 8888
¡Ha llegado un mensaje del servidor!
  Número de bytes recibidos: 13
  Mensaje: Hey! Listen!
Cerrando cliente...
xiana@xiana-Lenovo-V130-15IKB:~/Escritorio/Redes/Repositorio/Practicas/Practica_2$ 
```

```
xiana@xiana-Lenovo-V130-15IKB:~/Escritorio/Redes/Repositorio/Practicas/Practica_2$ ./cliente 127.0.0.1 8888
Se ha establecido la conexión con el servidor de:
  IP: 127.0.0.1
  Puerto: 8888
¡Ha llegado un mensaje del servidor!
  Número de bytes recibidos: 40
  Mensaje: Hey! Listen!
  ¡Hola desde el otro lado!
Cerrando cliente...
xiana@xiana-Lenovo-V130-15IKB:~/Escritorio/Redes/Repositorio/Practicas/Practica_2$ 
```

Figura 3: Captura con los dos resultados posibles para el cliente cuando se elimina la espera. En el primero, se llega demasiado pronto y solo se llega a leer el primer mensaje; en el segundo, se llega lo suficientemente tarde como para ver ambos mensajes.

En la figura 3 se puede ver que en el primer caso, correspondiente a la primera ejecución de cliente.c, solo se toman los 13 bytes del mensaje 1, mientras que en el segundo (la segunda ejecución) se cogen estos y los 27 correspondientes al mensaje 2, 40 en total.

### 1.d. Bucle con número de bytes a recibir

En este apartado se analiza qué ocurre cuando en lugar de llevar a cabo una única llamada a *recv()*, realizamos un bucle con múltiples llamadas, especificando tamaños máximos de recepción pequeños y progresivamente crecientes.

Para ello, definimos una nueva constante, *BYTES\_A\_RECIBIR*, en la que indicamos el tamaño máximo en bytes que se pueden guardar en el buffer del mensaje en cada iteración del bucle. Este será el número máximo de bytes que *recv()* lea del socket en cada llamada a la función, de forma que, si tal valor es menor que el total en bytes que contiene el socket, el programa cliente irá mostrando los mensajes de forma fraccionada. Los bytes que no quepan según el tamaño del buffer pasado como argumento serían descartados y permanecerían en el socket, pero se podrían recuperar una posterior llamada a *recv()*.

Para este estudio se ha adaptado la función *recibir()* como *recibir\_nbytes()* que, a diferencia de la anterior, permite variar el tercer argumento que se le pasa a *recv()*.

```

1  /*
2   * Función que encapsula recv() y su gestión de errores .
3   * Es utilizada por el servidor y por el cliente .
4   *
5   * Se reciben datos pasados a través de la conexión establecida sobre el socket
6   * de identificador sockcon . Se toman solo una cierta cantidad de bytes
7   * del mensaje . La función devuelve el número de bytes recibidos .
8   * En caso de error , se para la ejecución y se imprime un mensaje de error .
9   *
10  * Parámetros :
11  * - sockcon → Entrada . Identificador del socket de la conexión .
12  * - buffer → Salida . Puntero al buffer donde se guardará el mensaje recibido .
13  * - numbytes → Entrada . Número de bytes del mensaje que se deben recoger .
14  */
15 ssize_t recibir_nbytes(int sockcon , char * buffer , size_t numbytes){
16     ssize_t res_recv;           // Número de bytes recibidos
17
18     /*
19      * Llamamos a recv() con argumentos :
20      * - el entero identificador del socket de la conexión
21      * - un puntero al buffer donde guardar los datos
22      * - el máximo de bytes a recibir , que hemos elegido específicamente
23      * - las opciones predeterminadas (0)
24      */
25     if ((res_recv = recv(sockcon , (void *) buffer , numbytes , 0)) < 0){
26         perror("Error en la recepción de datos del servidor");
27         exit(EXIT_FAILURE);
28     } else if (res_recv == 0){
29         // No se ha recibido ningún byte o el socket se ha cerrado
30         printf("\nNo se han recibido datos\n");
31     }
32 }
```

```

33     return res_recv;
34 }
```

La manera de efectuar el procesamiento de los datos recibidos en cada iteración es la misma que describíamos en el apartado 1.c). Hay una única excepción: dado que estamos reutilizando *buffer* en sucesivas iteraciones, tenemos que asegurarnos de que, al imprimirla, no estemos mostrando caracteres correspondientes a lecturas anteriores. De no hacerlo, podría suceder, por ejemplo, que en una iteración se guardasen 5 bytes y en la siguiente, únicamente 3, sin ningún carácter nulo en ninguna de ellas. En tal caso, al imprimir esos 3 caracteres aparecerían también los 2 últimos de la iteración anterior (y cualquiera posterior, hasta encontrar un carácter nulo). Para impedirlo, en primer lugar inicializamos todos los elementos de la cadena con '\0', y después de cada llamada a cada llamada a *recibir\_nbytes()* sobreescribimos las posiciones potencialmente usadas con el carácter nulo antes de volver a llamar a la función. De esta forma, la cadena siempre quedará completamente vacía antes de guardar nuevos bytes en *buffer*.

```

1  ****RECEPCIÓN DEL MENSAJE ****
2
3 /*
4  * Por si el mensaje que envía el servidor no acaba en '\0', llenamos
5  * el buffer con '\0', de forma que aseguramos que siempre se encuentre
6  * el final de la cadena. Además, al tener tamaño N + 1, habrá '\0'
7  * incluso si se reciben N bytes, el máximo especificado.
8 */
9 memset(&buffer[0], '\0', sizeof(buffer));
10 /*
11 */
12 * Llamamos a recibir_nbytes(), una modificación de recibir() que
13 * especifica en recv() un cierto número de bytes a recoger del envío
14 * (el tercer argumento de recibir_nbytes()).
15 *
16 * Con este bucle, mantendremos la conexión abierta hasta que se reciban 0
17 * bytes o el servidor cierre su socket de conexión. De esta forma,
18 * aseguramos la recepción de todos los mensajes que envíe el servidor.
19 *
20 * En cada iteración vamos a sobreescribir el buffer. Ahora bien, solo
21 * aseguraríamos que la lectura de la cadena es correcta si el servidor
22 * siempre envía '\0' al final de sus mensajes, pues podría haber datos
23 * de anteriores envíos guardados en el buffer. Por precaución, vaciamos
24 * lo usado en la cadena antes de cada recepción.
25 */
26 while ((res_recv = recibir_nbytes(sockcli, &buffer[0],
27         (size_t) BYTES_A_RECIBIR)) > 0){
28     leido = 0;      // Reiniciamos leido
29
30     printf("\n Ha llegado un mensaje del servidor!\n");
31     // Aunque el formato estándar para ssize_t es %zd, los sistemas
32     // antiguos no lo soportan. Por seguridad, realizamos un cast a long
33     printf("\tNúmero de bytes recibidos: %ld\n", (long) res_recv);
34
35     // El mensaje que envíe el servidor puede contener caracteres nulos
36     // intermedios. Para asegurar que lo mostramos en su totalidad,
37     // llevamos la cuenta del número de bytes leídos, teniendo en cuenta
38     // que sizeof(char) = 1.
```

```

39 printf("\tMensaje: ");
40 while (leido < res_recv){
41     frase = &buffer[leido]; // Primer carácter sin leer
42     printf("%s\n\t", frase); // Imprimimos hasta encontrar '\0'
43
44     // Aumentamos leido para que apunte al carácter siguiente a '\0'.
45     // Nótese que strlen() no cuenta '\0', de modo que tenemos que
46     // debemos sumar 1 a la longitud de la cadena.
47     // Aunque el mensaje recibido no termine en '\0', lo hemos añadido
48     // artificialmente, de forma que estas operaciones tienen sentido.
49     leido += strlen(frase) + 1;
50 }
51
52 // Vaciamos la memoria que podríamos haber usado. En este caso, dado
53 // que trabajamos con arrays de caracteres, número de bytes y número
54 // de posiciones son comparables, pues sizeof(char) es 1. Así,
55 // llenaremos BYTES_A_RECIBIR posiciones con '\0'.
56 memset(&buffer[0], '\0', BYTES_A_RECIBIR);
57 }

```

Si comenzamos probando un valor de *BYTES\_A\_RECIBIR* igual a 5, lo que ocurrirá es que *recv()* irá tomando los datos del socket en grupos de 5, incluyendo los espacios y signos de puntuación:

```

xiana@xiana-Lenovo-V130-15IKB:~/Escritorio/Redes/Repositorio/Practicas/Practica_2$ ./servidor 5555
Se ha establecido una conexión con el cliente con:
  IP: 127.0.0.1
  Puerto: 40074
***** Envío 1 *****
  Se han enviado 13 bytes.
  Longitud del mensaje original: 13 bytes
  Mensaje original: Hey! Listen!
***** Envío 2 *****
  Se han enviado 27 bytes.
  Longitud del mensaje 2 original: 27 bytes
  Mensaje 2 original: ¡Hola desde el otro lado!
^C
xiana@xiana-Lenovo-V130-15IKB:~/Escritorio/Redes/Repositorio/Practicas/Practica_2$ [REDACTED]
xiana@xiana-Lenovo-V130-15IKB:~/Escritorio/Redes/Repositorio/Practicas/Practica_2$ ./cliente 127.0.0.1
1 5555
Se ha establecido la conexión con el servidor de:
  IP: 127.0.0.1
  Puerto: 5555
;Ha llegado un mensaje del servidor!
  Número de bytes recibidos: 5
  Mensaje: Hey!
;Ha llegado un mensaje del servidor!
  Número de bytes recibidos: 5
  Mensaje: Listen
;Ha llegado un mensaje del servidor!
  Número de bytes recibidos: 5
  Mensaje: n!
;
;Ha llegado un mensaje del servidor!
  Número de bytes recibidos: 5
  Mensaje: Hola
;Ha llegado un mensaje del servidor!
  Número de bytes recibidos: 5
  Mensaje: desde
;Ha llegado un mensaje del servidor!
  Número de bytes recibidos: 5
  Mensaje: el o
;Ha llegado un mensaje del servidor!
  Número de bytes recibidos: 5
  Mensaje: tro l
;Ha llegado un mensaje del servidor!
  Número de bytes recibidos: 5
  Mensaje: adol
No se han recibido datos
Cerrando cliente...
xiana@xiana-Lenovo-V130-15IKB:~/Escritorio/Redes/Repositorio/Practicas/Practica_2$ [REDACTED]

```

Figura 4: Ejecución en la que los bytes se pueden recibir, como máximo, en grupos de 5.

Si probamos con un número de bytes que no sea divisor del total de bytes enviados, como 7, vemos que la última recepción es de menos bytes que el valor indicado: `recv()` solo encuentra 6 bytes en el socket. Nótese que esto no es un fallo de la función, pues el parámetro `len` es, en teoría, el tamaño del buffer donde se guardarán los datos, por lo que únicamente sirve a modo de máximo de bytes a obtener, y no como mínimo.

Curiosamente, también se toman 6 bytes en lugar de 7 en otro punto de la ejecución: el correspondiente al final de la lectura del primer mensaje. Esto se debe a que, dado que

```
xiana@xiana-Lenovo-V130-15IKB:~/Escritorio/Redes/Repository/Practicas/Practica_2$ ./servidor 8888
Se ha establecido una conexión con el cliente con:
  IP: 127.0.0.1
  Puerto: 42482
***** Envío 1 *****
  Se han enviado 13 bytes.
  Longitud del mensaje original: 13 bytes
  Mensaje original: Hey! Listen!
***** Envío 2 *****
  Se han enviado 27 bytes.
  Longitud del mensaje 2 original: 27 bytes
  Mensaje 2 original: ¡Hola desde el otro lado!
^C
xiana@xiana-Lenovo-V130-15IKB:~/Escritorio/Redes/Repository/Practicas/Practica_2$ 
```

```
xiana@xiana-Lenovo-V130-15IKB:~/Escritorio/Redes/Repository/Practicas/Practica_2$ ./cliente 127.0.0.1
  8888
Se ha establecido la conexión con el servidor de:
  IP: 127.0.0.1
  Puerto: 8888
  ;Ha llegado un mensaje del servidor!
  Número de bytes recibidos: 7
  Mensaje: Hey! Li
  ;Ha llegado un mensaje del servidor!
  Número de bytes recibidos: 6
  Mensaje: sten!
  ;Ha llegado un mensaje del servidor!
  Número de bytes recibidos: 7
  Mensaje: ¡Hola
  ;Ha llegado un mensaje del servidor!
  Número de bytes recibidos: 7
  Mensaje: desde e
  ;Ha llegado un mensaje del servidor!
  Número de bytes recibidos: 7
  Mensaje: l otro
  ;Ha llegado un mensaje del servidor!
  Número de bytes recibidos: 6
  Mensaje: lado!
No se han recibido datos
Cerrando cliente...
xiana@xiana-Lenovo-V130-15IKB:~/Escritorio/Redes/Repository/Practicas/Practica_2$ 
```

Figura 5: Ejecución en la que los bytes se pueden recibir, como máximo, en grupos de 7.

eliminamos la espera, no estamos asegurando que la primera recepción se realice una vez hayan llegado ambos textos al cliente. Como ocurría en el apartado 1.c), no podemos determinar con seguridad qué ocurrirá en cada ejecución. Vemos que, por ejemplo, en la ejecución de la figura 4 se leen los datos una vez todos han llegado, al contrario de lo que ocurre en la figura 5.

Podemos probar con tamaños algo mayores, como 38:

```
xiana@xiana-Lenovo-V130-15IKB:~/Escritorio/Redes/Repository/Practicas/Practica_2$ ./servidor 5656
Se ha establecido una conexión con el cliente con:
  IP: 127.0.0.1
  Puerto: 55584
***** Envío 1 *****
  Se han enviado 13 bytes.
  Longitud del mensaje original: 13 bytes
  Mensaje original: Hey! Listen!
***** Envío 2 *****
  Se han enviado 27 bytes.
  Longitud del mensaje 2 original: 27 bytes
  Mensaje 2 original: ¡Hola desde el otro lado!
Se ha establecido una conexión con el cliente con:
  IP: 127.0.0.1
  Puerto: 55586
***** Envío 1 *****
  Se han enviado 13 bytes.
  Longitud del mensaje original: 13 bytes
  Mensaje original: Hey! Listen!
***** Envío 2 *****
  Se han enviado 27 bytes.
  Longitud del mensaje 2 original: 27 bytes
  Mensaje 2 original: ¡Hola desde el otro lado!
^C
xiana@xiana-Lenovo-V130-15IKB:~/Escritorio/Redes/Repository/Practicas/Practica_2$ 
```

```
xiana@xiana-Lenovo-V130-15IKB:~/Escritorio/Redes/Repository/Practicas/Practica_2$ ./cliente 127.0.0.1
  5656
Se ha establecido la conexión con el servidor de:
  IP: 127.0.0.1
  Puerto: 5656
  ;Ha llegado un mensaje del servidor!
  Número de bytes recibidos: 13
  Mensaje: Hey! Listen!
  ;Ha llegado un mensaje del servidor!
  Número de bytes recibidos: 27
  Mensaje: ¡Hola desde el otro lado!
No se han recibido datos
Cerrando cliente...
xiana@xiana-Lenovo-V130-15IKB:~/Escritorio/Redes/Repository/Practicas/Practica_2$ 
```

Figura 6: Dos ejecuciones en las que los bytes se toman, como máximo, en grupos de 38. En la primera, se leen los dos mensajes enviados por separado. En la segunda, se leen juntos.

De nuevo, habrá ocasiones en las que `recv()` se ejecute antes de que el servidor haya conseguido enviar el segundo mensaje, de forma que solo podrá recuperar los 13 bytes correspondientes al primero. Esto es lo que se muestra en la primera ejecución del programa cliente. Sin embargo, ocasionalmente puede que `recv()` se efectúe una vez los 40 bytes de los dos mensajes están en el socket. En ese caso, se tomarán 38, 2 y 0 bytes en cada llamada a la función, como se puede ver en la segunda ejecución.

Si finalmente probamos con un valor de `BYTES_A_RECIBIR` mayor que la suma de los tamaños de los dos mensajes, como por ejemplo 100, el comportamiento del programa será el

mismo que en el caso del apartado 1.c) sin la espera. Es decir, habrá ocasiones en las que los mensajes se tomen en iteraciones distintas, y otras en las que se tomen juntos:

```

xianaxiana-Lenovo-V130-15IKB:~/Escritorio/Redes/Repositorio/Practicas/Practica_2$ ./servidor 5555
Se ha establecido una conexión con el cliente con:
  IP: 127.0.0.1
  Puerto: 40044
***** Envío 1 *****
  Se han enviado 13 bytes.
  Longitud del mensaje original: 13 bytes
  Mensaje original: Hey! Listen!
***** Envío 2 *****
  Se han enviado 27 bytes.
  Longitud del mensaje 2 original: 27 bytes
  Mensaje 2 original: ;Hola desde el otro lado!
Se ha establecido una conexión con el cliente con:
  IP: 127.0.0.1
  Puerto: 40046
***** Envío 1 *****
  Se han enviado 13 bytes.
  Longitud del mensaje original: 13 bytes
  Mensaje original: Hey! Listen!
***** Envío 2 *****
  Se han enviado 27 bytes.
  Longitud del mensaje 2 original: 27 bytes
  Mensaje 2 original: ;Hola desde el otro lado!
^C
xianaxiana-Lenovo-V130-15IKB:~/Escritorio/Redes/Repositorio/Practicas/Practica_2$ 

xianaxiana-Lenovo-V130-15IKB:~/Escritorio/Redes/Repositorio/Practicas/Practica_2$ ./cliente 127.0.0.
1 5555
Se ha establecido la conexión con el servidor de:
  IP: 127.0.0.1
  Puerto: 5555
;Ha llegado un mensaje del servidor!
  Número de bytes recibidos: 13
  Mensaje: Hey! Listen!
;Ha llegado un mensaje del servidor!
  Número de bytes recibidos: 27
  Mensaje: ;Hola desde el otro lado!
No se han recibido datos
Cerrando cliente...
xianaxiana-Lenovo-V130-15IKB:~/Escritorio/Redes/Repositorio/Practicas/Practica_2$ ./cliente 127.0.0.
1 5555
Se ha establecido la conexión con el servidor de:
  IP: 127.0.0.1
  Puerto: 5555
;Ha llegado un mensaje del servidor!
  Número de bytes recibidos: 40
  Mensaje: Hey! Listen!
  ;Hola desde el otro lado!
No se han recibido datos
Cerrando cliente...
xianaxiana-Lenovo-V130-15IKB:~/Escritorio/Redes/Repositorio/Practicas/Practica_2$ 

```

Figura 7: Dos ejecuciones en las que los bytes se toman, como máximo, en grupos de 100. En la primera, se leen los dos mensajes enviados por separado. En la segunda, se leen juntos.

En todos los casos, el bucle finaliza cuando, tras imprimir los últimos bytes hallados (la parte final del segundo mensaje), *recv()* vuelve a intentar recuperar datos del socket y se encuentra con que ya no hay ninguno. En consecuencia, la función devuelve 0, indicando que no se ha recuperado ningún byte, y el bucle finaliza. Si en algún punto se cerrara el socket de conexión, *recv()* también devolvería 0 y el programa cliente terminaría.

### 3. Múltiples clientes secuenciales en servidor de mayúsculas

En este apartado se ha comprobado que el servidor de mayúsculas del apartado 2 puede mantener a varios clientes aguardando, para atenderlos uno después del otro.

Vale la pena mencionar que el número de clientes que pueden mantenerse esperando a ser atendidos no se corresponde con el valor que se selecciona como longitud máxima de la cola de espera en la función *listen()*, debido a la implementación que puedan tener internamente los sockets.

Para permitir lanzar un segundo programa desde otra terminal antes de que termine el primer cliente, se ha añadido un retardo de 2 segundos en el lazo de lectura del fichero, antes de que cada cliente le envíe la línea que ha obtenido al servidor.

En la secuencia de ejecuciones siguiente, vemos como los dos clientes establecen sendas conexiones, pero mientras que el primero (ventana superior derecha) comienza a recibir sus líneas pasadas a mayúsculas desde el servidor (ventana izquierda) inmediatamente, el segundo cliente (ventana inferior derecha), tras enviar la primera línea, queda esperando a que el primero finalice su ejecución para empezar a recibir su texto convertido. Es decir, el servidor nunca alterna su servicio entre varios clientes, debido a la construcción del código. Solo comienza a responder a las peticiones de un nuevo cliente tras desconectarse del anterior.

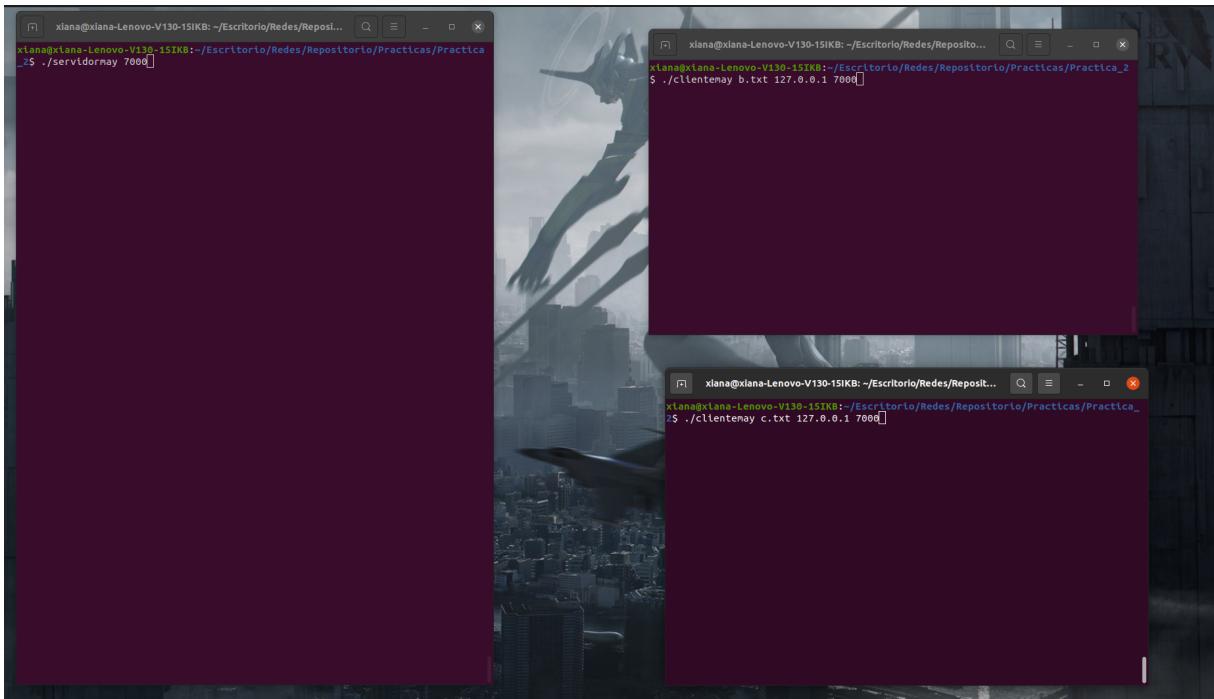


Figura 8: Programas que se ejecutarán en cada terminal.

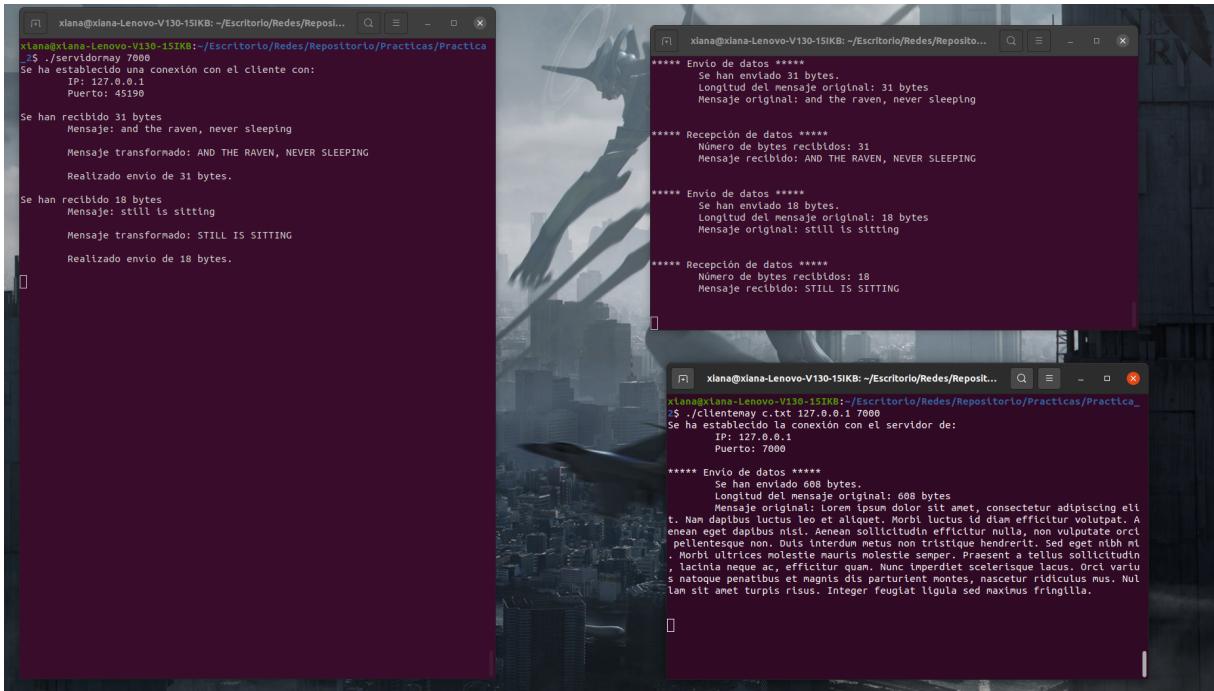


Figura 9: Captura tomada poco tiempo después de comenzar la ejecución de ambos programas. El segundo cliente queda a la espera de que el mensaje que ha enviado sea devuelto una vez pasado a mayúsculas. El primer cliente está siendo asistido.

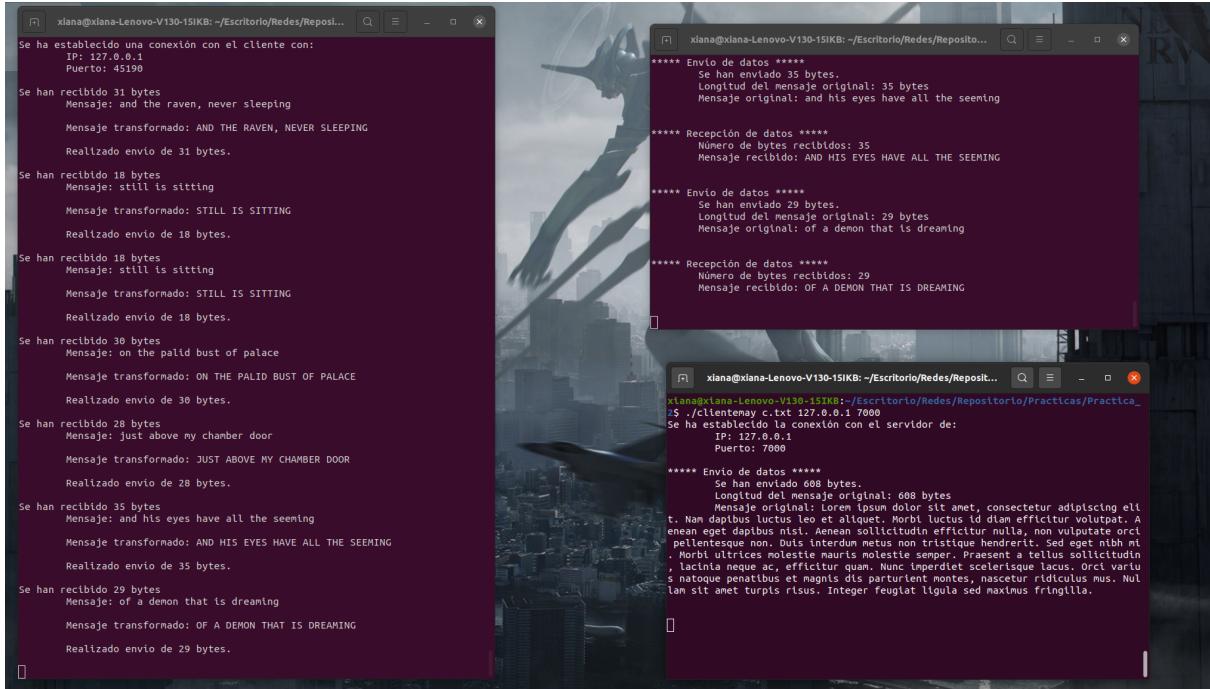


Figura 10: Únicamente se procesan las peticiones del primer cliente. El segundo sigue esperando.

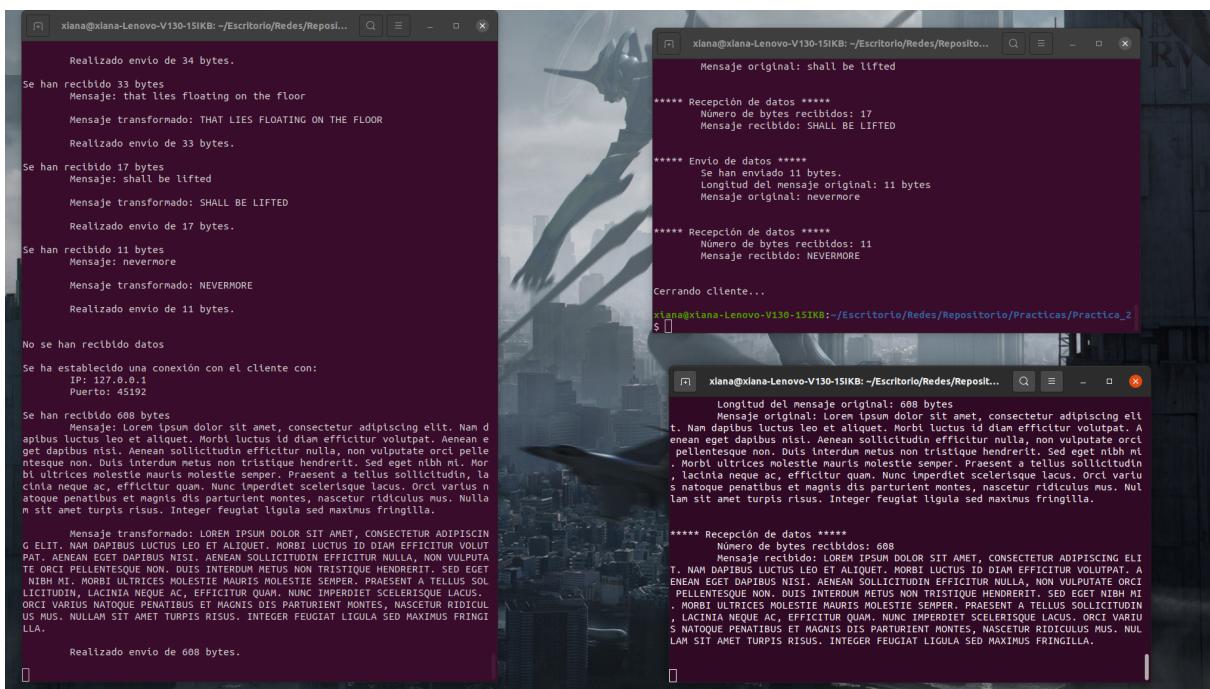


Figura 11: Cuando el primer cliente finaliza, el segundo es atendido.

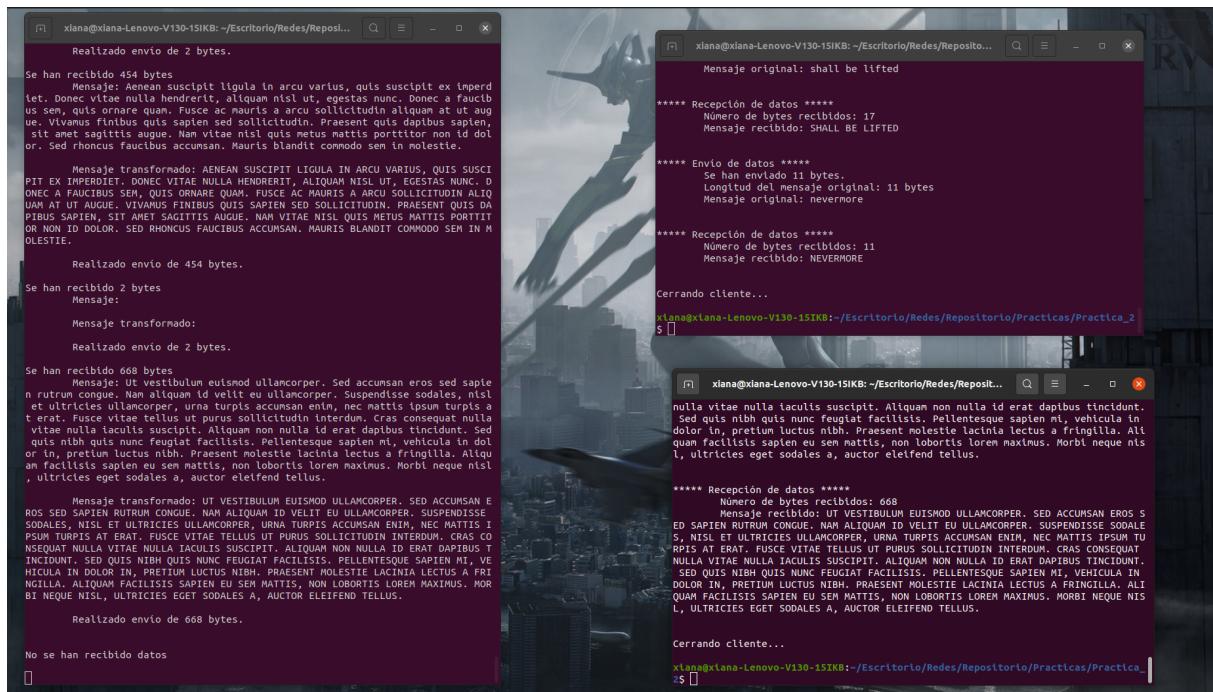


Figura 12: Captura cuando ambos clientes han finalizado y solamente el servidor sigue vivo.

Si se prueba con 3 clientes, los resultados son análogos. El servidor se conecta a ellos y les proporciona servicio de uno en uno.

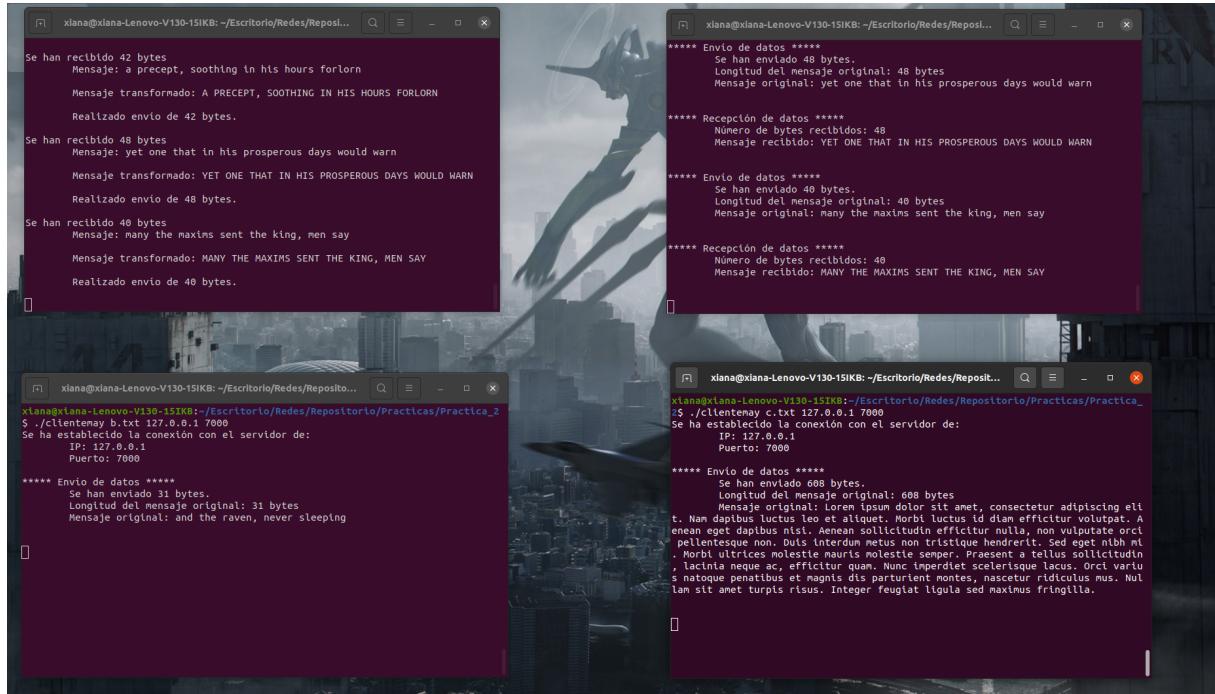


Figura 13: Atención al cliente 1 de 3.

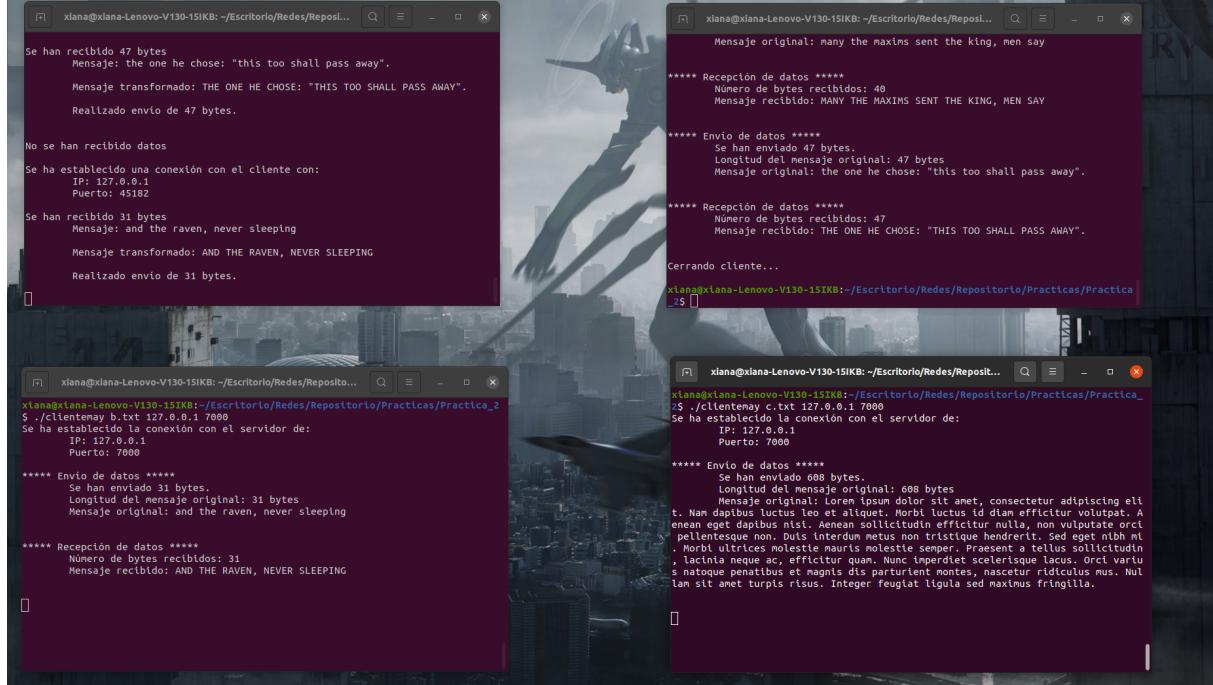


Figura 14: Atención al cliente 2 de 3.

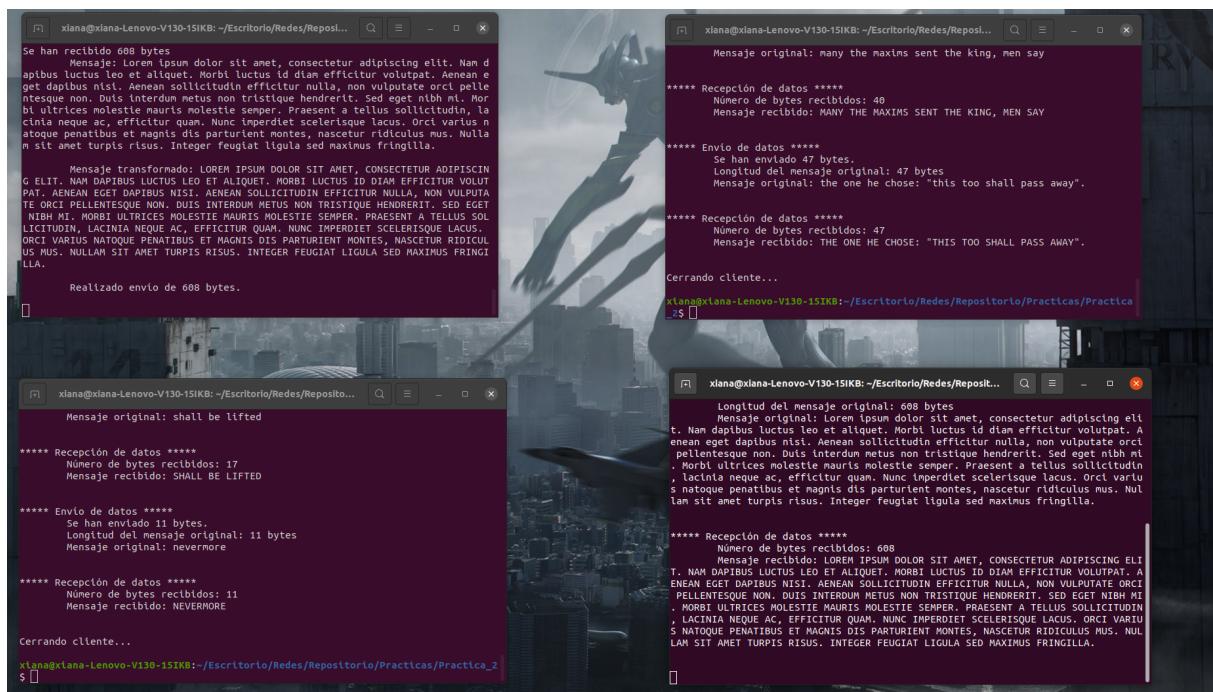


Figura 15: Atención al cliente 3 de 3

## Anexo

Se incluye a continuación una captura de la ejecución de los programas del apartado 2. en equipos distintos. La conexión se ha realizado a través de SSH.

Actividades Terminal ▾ Ven 19 de Nov 13:03

RAl\xiana.carrera@e220a045h169l:~/Practica\_2\$ ./servidormay 5555  
Se ha establecido una conexión con el cliente con:  
IP: 172.25.45.169  
Puerto: 57210  
  
Se han recibido 67 bytes  
Mensaje: And the Raven, never flitting, still is sitting, still is sitting  
Mensaje transformado: AND THE RAVEN, NEVER FLITTING, STILL IS SITTING, STILL  
IS SITTING  
Realizado envío de 67 bytes.  
  
Se han recibido 58 bytes  
Mensaje: On the pallid bust of Pallas just above my chamber door;  
Mensaje transformado: ON THE PALLID BUST OF PALLAS JUST ABOVE MY CHAMBER DOO  
R;  
Realizado envío de 58 bytes.  
  
Se han recibido 68 bytes  
Mensaje: And his eyes have all the seeming of a demon's that is dreaming,  
Mensaje transformado: AND HIS EYES HAVE ALL THE SEEING OF A DEMON'S THAT IS  
DREAMING,  
Realizado envío de 68 bytes.  
  
Se han recibido 73 bytes  
Mensaje: And the lamp-light o'er him streaming throws his shadow on the floo  
r;  
Mensaje transformado: AND THE LAMP-LIGHT O'ER HIM STREAMING THROWS HIS SHAD  
O W ON THE FLOOR;  
Realizado envío de 73 bytes.  
  
Se han recibido 66 bytes  
Mensaje: And my soul from out that shadow that lies floating on the floor  
Mensaje transformado: AND MY SOUL FROM OUT THAT SHADOW THAT LIES FLOATING ON  
THE FLOOR  
Realizado envío de 66 bytes.  
  
Se han recibido 30 bytes  
Mensaje: Shall be lifted-nevermore!  
Mensaje transformado: SHALL BE LIFTED-NEVERMORE!  
Realizado envío de 30 bytes.  
  
No se han recibido datos

RAl\xiana.carrera@e220a045h165l:~/Escritorio/Practica\_2\$ ./clientemay a.txt 172.25.45.169 5555  
Se ha establecido la conexión con el servidor de:  
IP: 172.25.45.169  
Puerto: 5555  
\*\*\*\*\* Envío de datos \*\*\*\*\*  
Se han enviado 67 bytes.  
Longitud del mensaje original: 67 bytes  
Mensaje original: And the Raven, never flitting, still is sitting, still is sitting  
  
\*\*\*\*\* Recepción de datos \*\*\*\*\*  
Número de bytes recibidos: 67  
Mensaje recibido: AND THE RAVEN, NEVER FLITTING, STILL IS SITTING, STILL IS SITTING  
  
\*\*\*\*\* Envío de datos \*\*\*\*\*  
Se han enviado 58 bytes.  
Longitud del mensaje original: 58 bytes  
Mensaje original: On the pallid bust of Pallas just above my chamber door;  
  
\*\*\*\*\* Recepción de datos \*\*\*\*\*  
Número de bytes recibidos: 58  
Mensaje recibido: ON THE PALLID BUST OF PALLAS JUST ABOVE MY CHAMBER DOOR;  
  
\*\*\*\*\* Envío de datos \*\*\*\*\*  
Se han enviado 68 bytes.  
Longitud del mensaje original: 68 bytes  
Mensaje original: And his eyes have all the seeming of a demon's that is dreaming,  
  
\*\*\*\*\* Recepción de datos \*\*\*\*\*  
Número de bytes recibidos: 68  
Mensaje recibido: AND HIS EYES HAVE ALL THE SEEING OF A DEMON'S THAT IS DREAMING,  
  
\*\*\*\*\* Envío de datos \*\*\*\*\*  
Se han enviado 73 bytes.  
Longitud del mensaje original: 73 bytes  
Mensaje original: And the lamp-light o'er him streaming throws his shadow on the floor;  
  
\*\*\*\*\* Recepción de datos \*\*\*\*\*  
Número de bytes recibidos: 73  
Mensaje recibido: AND THE LAMP-LIGHT O'ER HIM STREAMING THROWS HIS SHADOW ON THE FLOOR;  
  
\*\*\*\*\* Envío de datos \*\*\*\*\*  
Se han enviado 66 bytes.  
Longitud del mensaje original: 66 bytes  
Mensaje original: And my soul from out that shadow that lies floating on the floor  
  
\*\*\*\*\* Recepción de datos \*\*\*\*\*  
Número de bytes recibidos: 66  
Mensaje recibido: AND MY SOUL FROM OUT THAT SHADOW THAT LIES FLOATING ON THE FLOOR  
  
\*\*\*\*\* Envío de datos \*\*\*\*\*  
Se han enviado 30 bytes.  
Longitud del mensaje original: 30 bytes  
Mensaje original: Shall be lifted-nevermore!  
  
\*\*\*\*\* Recepción de datos \*\*\*\*\*  
Número de bytes recibidos: 30  
Mensaje recibido: SHALL BE LIFTED-NEVERMORE!

Cerrando cliente...

Figura 16: Captura de la ejecución del servidor y cliente de mayúsculas en dos ordenadores del aula de prácticas