Sesión 4: Programación con sockets

Transmisión de datos binarios

José Luis Díaz

Curso 2011-2012

En los ejercicios anteriores los datos que transmitíamos por lossockets eran de tipo `char` (texto que el usuario tecleaba en la consola). Sin embargo es posible transmitir cualquier tipo de dato por un *socket*, puesto que las funciones `read/write` (o `recv/send`) se limitan a transferir bytes.

En esta sesión escribiremos un cliente para el servicio `time`, el cual, de acuerdo con el RFC, envía un entero largo. Veremos los problemas de ordenación de los bytes dentro de ese entero *endianity*, y cómo convertir su valor a algo con significado para el usuario.

1. El servicio `time`

De acuerdo con el RFC 868, un servidor que escuche en el puerto 37 proporcionará el servicio `time`, que consiste en enviar la fecha y hora del servidor, pero en un formato un tanto particular: en lugar de enviar una cadena de caracteres, como por ejemplo "20:31:56 del 13 de Noviembre 2005", lo que envía es un entero largo (es decir, de 32 bits) que contiene el número de segundos que han transcurrido desde el 1 de Enero de 1900.

1.1. Primer experimento: conectando con telnet

La máquina `sirio.edv.uniovi.es` tiene un servidor `time` escuchando por el protocolo TCP, por lo que podemos probarlo conectando con `telnet`. Veremos algo así:

```
$ telnet sirio.edv.uniovi.es 37
Trying 156.35.151.2...
Connected to sirio.
Escape character is '^]'.
Ç#9nConnection closed by foreign host.
```

Las tres primeras líneas las imprime el propio programa `telnet`, así como el mensje "`Connection closed by foreign host.`". Si observamos con atención la última línea, vemos las letras Ç#9n (serán otras cuando tú lo pruebes). ¿Qué significan?

Precisamente se trata de la respuesta dada por el servidor. En efecto, de acuerdo con el RFC 868, la respuesta del servidor consistirá en un entero de 32 bits, que necesitará por tanto cuatro bytes para ser representado. Estos cuatro bytes son los que recibe `telnet` y los interpreta (erróneamente) como cuatro caracteres. Casualmente, en mi experimento resultan ser cuatro caracteres imprimibles, pero en tu caso podrías no ver cuatro, porque alguno de ellos corresponda con un carácter de control o invisible.

Lo mejor para estudiar la respuesta de `telnet` es volcar su salida a un fichero para despues examinarla con la utilidad `hexdump`. Es decir:

```
$ telnet sirio.edv.uniovi.es 37 > respuesta
Connection closed by foreign host.
$ hexdump -C respuesta
00000000  54 72 79 69 6e 67 20 31  35 36 2e 33 35 2e 31 35  |Trying 156.35.15|
00000010  31 2e 32 2e 2e 2e 0a 43  6f 6e 6e 65 63 74 65 64  |1.2....Connected|
00000020  20 74 6f 20 73 69 72 69  6f 2e 65 64 76 2e 75 6e  | to sirio.edv.un|
00000030  69 6f 76 69 2e 65 73 2e  0a 45 73 63 61 70 65 20  |iovi.es..Escape |
00000040  63 68 61 72 61 63 74 65  72 20 69 73 20 27 5e 5d  |character is '^]|
00000050  27 2e 0a c7 23 3a 63                              |'..Ç#:c|
00000057
```

Como vemos, al redireccionar la salida estándar con el operador >, aún sigue apareciendo en pantalla el mensaje "`Connection closed by foreign host.`", debido a que este mensaje es enviado por `telnet` a la salida de error. En el fichero `respuesta` tenemos lo que `telnet` ha enviado a la salida estándar. Al examinarlo con `hexdump`, vemos los mensajes iniciales que `telnet` ha impreso, y la parte que nos interesa que son justamente los últimos cuatro bytes. Estos han cambiado, ya no son `Ç#9n`, sino `Ç#:c`. Es normal que cambien, puesto que dependen del paso del tiempo. Mirando el volcado hexadecimal vemos que el valor de estos cuatro bytes es `c7`, `23`, `3a` y `63`. Por tanto, el entero de 32 bits enviado por el servidor es `C7233A63`, en hexadecimal.

Si convertimos este número a base 10, por ejemplo usando el programa `bc` que viene con Linux, vemos que su valor es 3 340 974 691.

```
$ bc
ibase=16
C7233A63
3340974691
```

Este es el número de segundos transcurrido desde el 1 de enero de 1900, al menos según el reloj de la máquina `sirio`. Dividiendo sucesivamente por 3600, por 24, y por 365, podremos saber de forma aproximada el número de horas, de días, y de años transcurridos, y por tanto tener una estimación del año en que estamos. Operando con mucho más cuidado, a partir de ese número podremos obtener la fecha y hora exactas que tenía `sirio` cuando nos respondió. Naturalmente estos cálculos no son simples, pues deben tener en cuenta las diferentes longitudes de los meses, la existencia de años bisiestos, etc. Veremos que hay una función que nos puede hacer gran parte del trabajo.

1.2. Segundo experimento: escribiendo un cliente TCP

Escribe un cliente mínimo que se limite a conectar con el puerto 37 de la máquina `sirio` (cuya IP es `156.35.151.2`) y leer el entero que ésta máquina le envía, mostrándolo seguidamente por pantalla.

Ten en cuenta lo siguiente:

- El entero debe tener un tamaño de 32 bits. Por otro lado, los 32 bits son usados como parte del número, no hay bit de signo. Esto implica que la variable debe ser declarada en C como `unsigned long int`. Si incluyes `<sys/types.h>` puedes abreviar este tipo como `ulong`.
- Para recibir el entero debes usar `read`, pasándole la dirección de la variable, y el número de bytes que ocupa (cuatro).
- Para imprimir el valor del entero con `printf()`, la cadena de formato `%d` no es apropiada, ya que ésta es para enteros con signo. Debes usar `%lu` (la `l` indica "long" y la `u` "unsigned".

Ejecuta el programa un par de veces y anota los números que imprime. Usa una calculadora para restarlos. Debería darte aproximadamente el número de segundos transcurridos entre las dos ejecuciones. ¿Tiene sentido lo que obtienes? Si no es así ¿a qué crees que se debe? ¿cómo lo arreglarías?

1.3. Imprimiendo fecha y hora en formato legible

Una vez hayas solucionado los problemas anteriores, tu cliente imprimirá en pantalla un número que de alguna manera representa la fecha y hora actuales. Sin embargo sería mejor si en lugar del número de segundos transcurridos desde 1900 nos dijera directamente la fecha y hora en un formato más legible, como por ejemplo el que produce el comando `date`.

Es posible convertir el número que hemos recibido a un formato legible haciendo uso de la función `ctime()`. Esta función recibe un entero de 32 bits y devuelve una cadena de caracteres, con una representación textual de la fecha y hora. No obstante, si lees con atención la página de manual de `ctime()`, verás que hay una pequeña pega: el entero que espera `ctime()` es el número de segundos transcurridos desde el 1 de enero de 1970, y no desde 1900, que es el dato que el servidor nos envía.

Por suerte, conociendo el número de segundos transcurrido entre ambas fechas no es difícil la conversión. Basta restar al dato que nos envía el servidor esta constante, y ya tenemos un dato que le podemos pasar a `ctime()`. El valor de esta constante es `2208988800LU` (las letras "`LU`" al final del número son para indicar al compilador C que se trata de un `long unsigned`.

La función `ctime()` también puede ser usada para imprimir la hora que tiene la máquina local. Basta averiguar antes esta hora con la función `time()`. Esta función recibe opcionalmente un parámetro, que puede ser `NULL`, y nos devuelve el número de segundos transcurrido desde el 1 de enero de 1970 (observa que en este caso no es necesario restarle la constante). Si se le pasa un puntero distinto de `NULL`, el valor retornado será también almacenado en esa dirección.

Escribe un programa que consulte la hora local usando `time()` y después la hora de `sirio`, conectándose a su puerto 37. Seguidamente imprime ambas horas por pantalla en un formato legible con ayuda de `ctime()`. Ejecuta el programa y comprueba que salen horas parecidas.

2. Accediendo a los nodos por su nombre

En todos los ejemplos que hemos desarrollado hasta ahora, la dirección del servidor se especificaba mediante su IP. Esto es incómodo. Si lo que conocemos es el nombre del nodo pero no su IP, podemos utilizar la función `gethostbyname()` para averiguar esta última. En las transparencias de teoría hay ejemplos de cómo se hace esto.

Utiliza estos ejemplos para modificar el cliente que has programado en el punto anterior, de modo que ahora admita desde la línea de comandos el nombre de la máquina a la que se debe conectar para averiguar la hora. El puerto no es necesario especificarlo desde línea de comandos, ya que será siempre el 37. Un ejemplo de utilización de este cliente sería:

```
$ quehoraes sirio.edv.uniovi.es
Hora servidor: Wed Nov 16 11:31:17 2005
Hora local   : Wed Nov 16 11:32:14 2005
```

Prueba a poner `hora.uniovi.es` como nombre del nodo. Esta máquina mantiene un reloj muy exacto que puedes utilizar para poner en hora el tuyo.

3. Cliente UDP

El protocolo `time` también está definido para UDP. Leyendo el RFC 868 vemos que en este caso el cliente debe enviar al servidor un datagrama vacío (esto es, con 0 bytes), y el servidor enviará al cliente como respuesta el entero de 32 bits que representa el número de segundos transcurridos desde el inicio de 1900. ¿Por qué crees que el cliente debe enviar un datagrama vacío?

Escribe un cliente que implemente el protocolo UDP. Puedes tomar gran parte del código del cliente que ya tenías para TCP. Prueba el cliente con la máquina `sirio` (la máquina `hora.uniovi.es` no tiene servidor para UDP, sólo para TCP.

4. Ejercicios adicionales

4.1. El fin de los tiempos

Ya que el servidor usa un entero sin signo de 32 bits para contener la respuesta, hay una limitación inherente a este protocolo. El número más alto que cabe en ese entero es $(2^{32} - 1)$ por lo que a partir de una cierta fecha, el protocolo dejará de funcionar. ¿Cuál será esta fecha?

El mecanismo con el que Unix guarda la fecha y hora también está limitado por la misma razón, pero la fecha crítica será otra. En primer lugar UNIX usa números con signo para almacenar el tiempo, por lo que el máximo valor es $(2^{31} - 1)$, la mitad del caso anterior, por lo que la fecha crítica se acerca. Sin embargo, en lugar de comenzar a contar en 1900 lo hace en 1970, por lo que la fecha crítica se aleja. ¿En qué fecha y hora exactas dejará de funcionar?

> **Pista**: Puedes usar la función `ctime()` para que te imprima esa fecha, si le pasas el entero apropiado. Si incluyes `<limits.h>` en tu código, tendrás definidas un par de constantes llamadas `ULONG_MAX` y `LONG_MAX`, que contienen respectivamente el máximo valor que puede tomar un entero largo sin signo, o un entero largo con signo.

4.2. Portándolo todo a Windows

Reescribe los clientes anteriores para que compilen en Windows. Comprueba que compilan y se ejecutan correctamente. Recuerda que debes renombrarlos con la extensión `.CPP`, aunque el contenido sea código C y no C++.

4.3. Localización en Unix

Habrás notado que la función `ctime()` imprime la hora en inglés. La función `strftime()` proporciona mucha más flexibilidad. Por un lado, se adapta al idioma que el usuario elija, y por otro permite especificar con todo detalle cómo queremos mostrar la fecha y hora, desde algo tan compacto como: "20:00 2/10/2005" hasta algo tan extenso como "08:00:00pm del 2 de octubre de 2005".

Esta función, sin embargo, no admite como parámetro un entero largo representando el número de segundos, sino una estructura de tipo `tm` (definida en `time.h`). La función `localtime()` permite convertir el entero largo al tipo `struct tm`.

```
#include <time.h>
...

struct tm *hora_tm;
unsigned long int hora;

// Inicializar "hora" llamando a time() o pidiendosela
// a un servidor a traves del puerto 37

// Una vez inicializada se convierte así:
hora_tm=localtime(&hora);
```

Una vez tenemos la hora en la estructura `tm`, podemos convertirlo a una cadena de caracteres usando `strftime`. El tercer parámetro de esta función es una "cadena de formato". El concepto es similar al de las cadenas de formato usadas por `printf()`, pero en este caso con caracteres especiales para los campos de una fecha u hora. Por ejemplo, el especificador `%Y` representa el año con cuatro cifras, el especificador `%B` el nombre del mes con todas las letras, o `%b` el nombre del mes abreviado, etc. Para más detalles, consúltese la página del manual. En particular puede ser útil el especificador `%c` que representa la fecha y hora (todo junto), en el formato que se considere más adecuado para el idioma del usuario. Así, para un inglés producirá algo como: "Wed Nov 16 12:27:19 2005", mientras que para un español producirá "mié 16 nov 2005 12:27:19 CET"

El siguiente fragmento de código muestra cómo usar `strftime()`, suponiendo que la variable `hora_tm` es un `struct tm` correctamente inicializado.

```
#include<time.h>
...
 char buffer[200];

 // Imprimir en modo compacto
 strftime(buffer, sizeof(buffer),
          "Fecha y hora: %c", hora_tm);
 printf("%s\n", buffer);

 // Imprimir en modo libre
 strftime(buffer, sizeof(buffer),
          "Son las %Son las %X del %d de %B de %Y (%A)", hora_tm);
 printf("%s\n", buffer);
```

Finalmente, para que el programa reconozca el idioma preferido por el usuario, es necesario incluir al principio de la función `main()` las siguientes líneas de código:

```
#include <locale.h>
...
int main(...)
{
  char *locale;
  ...

  if (!(locale=setlocale(LC_TIME, ""))) {
```

```
    fprintf(stderr, "El locale no es valido.\n");
    exit(-1);
  }
  ...
}
```

La función `setlocale()` permite averiguar las opciones de *localización* que el usuario tiene activadas. Estas opciones permiten a un usuario especificar cosas como el lenguaje en que prefiere que le sean mostrados los mensajes o las páginas de *man*, la codificación de caractéres que usa su terminal (*ISO-8859* o *UTF-8*, por ejemplo), el orden alfabético de las letras de su alfabeto, el orden en que prefiere que se le muestren las fechas (día/mes/año, mes/dia/año, etc), si el separador de decimales debe ser un punto o una coma, etc. En definitiva, se trata de opciones que intentan que los programas se adapten a las variaciones culturales de los usuarios.

En particular, la variable `LC_TIME` controla cómo deben mostrarse las fechas y horas, y por tanto es la que interesa de cara a la función `strftime()`.

Ahora podemos probar a ejecutar el programa y veremos cómo las cadenas generadas por `strftime()` se adaptan a nuestra configuración local. El usuario puede cambiar esa configuración asignando diferentes valores a la variable de entorno `LC_TIME`, y puede averiguar qué opciones tiene en un momento dado usando el comando `locale`. Por ejemplo:

```
$ locale
LANG="es_ES@euro"
LC_CTYPE="es_ES@euro"
LC_NUMERIC="es_ES@euro"
LC_TIME="es_ES@euro"
LC_COLLATE="es_ES@euro"
LC_MONETARY="es_ES@euro"
LC_MESSAGES="es_ES@euro"
LC_PAPER="es_ES@euro"
LC_NAME="es_ES@euro"
LC_ADDRESS="es_ES@euro"
LC_TELEPHONE="es_ES@euro"
LC_MEASUREMENT="es_ES@euro"
LC_IDENTIFICATION="es_ES@euro"
LC_ALL=es_ES@euro
```

Esto indica que por defecto, usamos la opción `es_ES@euro` para todas las variables de localización (entre ellas `LC_TIME`). Este valor indica que nuestra preferencia es el español (`es`), en su dialecto hablado en España (`_ES`), y con la codificación de caracteres que incluye el euro (que será la ISO-8858-15).

Podemos cambiar nuestras opciones de localización, por ejemplo dándoles el valor "C":

```
$ export LC_ALL=C
```

En este caso el valor "C" significa "las opciones por defecto para el lenguaje C", que es lo mismo que decir "inglés". Si ejecutamos de nuevo el programa tras este cambio veremos que ahora los nombres de meses y días de la semana salen en inglés, y que en la forma "compacta", el mes sale antes del día. Dependiendo de la instalación de Unix podemos tener más localizaciones disponibles (francés, alemán, etc.) Podemos averiguar qué localizaciones concretas tiene instaladas nuestro sistema si ponemos:

```
$ locale -a
C
POSIX
es_ES.iso885915@euro
es_ES@euro
```

Comprobamos que en este caso (ejecutado en `gollum`), no hay muchas opciones disponibles. Las opciones `C` y `POSIX` son lo mismo, y se refieren al inglés americano. Las otras dos opciones también se refieren a lo mismo (son *alias* de la misma localización) y se refiere al español de España con el euro.