Sesión 3: Introducción a la programación con sockets

José Luis Díaz

Curso 2011-2012

1. Introducción: los RFC

Todos los protocolos, procedimientos y convenciones utilizados para construir la Internet están documentados y estandarizados en una serie de documentos numerados denominados RFCs. Las siglas vienen originalmente de *Request for Comments*, y se mantienen por razones históricas, pero lo cierto es que más que "peticiones de comentarios" son documentos finales que son usados como referencia para implementar las funcionalidades que describen.

Estos documentos están numerados, y cada uno de ellos define un aspecto del funcionamiento de Internet. Por ejemplo, el RFC 768 describe el protocolo UDP (datagramas), el RFC 1939 define el *Post Office Protocol* (POP3) para la recepción de mensajes de correo desde un servidor, etc.

Estos documentos son accesibles a través de Internet, por ejemplo en `http://www.rfc-editor.org/rfcsearch.html`. Están almacenados en un formato de texto plano (`.txt`, ASCII), y escritos en inglés. Aunque son documentos técnicos, no suelen ser muy difíciles de comprender pues están muy orientados a la práctica.

Los RFC's más frecuentemente usados están traducidos al español en `http://www.rfc-es.org/descargas.php`.

2. El protocolo `echo`

2.1. Definición: RFC 862

El protocolo `echo` es quizás el más sencillo de los protocolos posibles. Tal como define el RFC, un servidor *echo* debe limitarse a devolver al cliente los mensajes que éste le envíe.

Se reserva el número de puerto 7 para este cometido. Es decir, un cliente que se conecte al puerto 7 de una máquina, espera encontrar allí un servidor `echo`.

2.2. Usando telnet como cliente genérico

El comando `telnet` tiene dos posibles usos:

- `telnet` *máquina* Si se invoca con esta sintaxis, `telnet` intentará una conexión al puerto 25 de la máquina, donde encontrará un *servidor telnet*, que pedirá al cliente nombre y clave y seguidamente iniciará una *sesión interactiva*.

  Usado de este modo, `telnet` es equivalente a `ssh`, pero sin encriptación. Es decir, los mensajes que se intercambian el cliente y el servidor viajan por la red en *texto claro*, y podrían ser espiados. Es por esto que, siempre que la máquina disponga de un servidor `ssh`, se preferirá éste en lugar de `telnet`.

- `telnet` *máquina puerto*. Si se invoca con esta sintaxis, `telnet` intentará conectar al puerto especificado de la máquina especificada, usando el protocolo TCP. Si se logra la conexión, a partir de ese momento `telnet` mostrará por pantalla todos los mensajes que la máquina nos envíe, y a su vez enviará a esta máquina todo aquello que tecleemos.

  Dependiendo de qué versión del programa `telnet` estemos usando, puede ser que el envío de datos a la máquina ocurra cada vez que pulsemos una tecla, o tan sólo cuando pulsemos retorno de carro (y en este caso enviaría toda la línea).

Al usar `telnet` de la segunda forma, podremos conectarnos a cualquier servidor para "dialogar" con él. Por ejemplo, podemos conectarnos al puerto 7 de una máquina para establecer contacto con el servidor "echo".

Para salir de `telnet` es necesario pulsar la combinación de teclas `Control` y `]`, tras lo cual aparecerá un *prompt* que dice "`telnet>`", en el que deberemos escribir `quit` y pulsar el retorno de carro.

Podemos probar a hacer `telnet` a la máquina `sirio.edv.uniovi.es`, que tiene instalados algunos servidores. Por ejemplo, haciendo telnet al puerto número 13, nos devolverá la fecha y hora en formato legible (en forma de texto inglés) y después cerrará la conexión antes de dejarnos escribir nada. Este servicio se denomina *daytime* y está documentado en el RFC 867. También podemos conectar con el puerto 7, donde encontraremos un servidor `echo`.

2.3. Nuestro propio servidor `echo`

Vamos a programar un servidor `echo` que implemente lo que dice el estándar RFC 862, tanto para el protocolo TCP como para el UDP. Además haremos versiones para Windows y para Linux.

Para empezar, suministramos un código fuente ya casi listo para compilar en máquinas Unix. Puedes encontrarlo en `http://www.atc.uniovi.es/telematica/3sod/material/echo-basico.tar`. Este fichero es un "archivo tar", que es un formato que permite almacenar varios ficheros en un solo archivo, una idea similar a la de los formatos `zip` o `rar`, pero sin comprimir.

Para extraer los ficheros individuales del archivo, debes usar el comando:

```
tar xf echo-basico.tar
```

Encontrarás los ficheros `echo-tcp-basico.c` y `echo-udp-basico.c`. Debes editar ambos y asignar un valor válido a la constante `PUERTO`. Tras esto puedes compilar ambos usando `gcc`.

La versión TCP y UDP son similares en funcionamiento, aunque diferentes en su implementación. Simplemente esperan un mensaje del cliente, lo imprimen en pantalla para que podamos ver lo que ha llegado, y lo reenvían al cliente tal cual, como manda el RFC.

Para probar el servidor basta ejecutarlo y ver los mensajes que va imprimiendo, que serán los que reciba de los clientes. Obviamente necesitaremos que algún cliente se conecte, y ya que no hemos programado ningún cliente aún, podemos usar `telnet` para conectar con él servidor TCP. El servidor UDP no podremos probarlo hasta que no hayamos programado un cliente UDP.

Para que la prueba sea más divertida, podemos pedir a un compañero que conecte con nuestro servidor desde otra máquina. Para ello debemos decirle nuestro número de puerto.

2.3.1. Ahora todo en MAYUSCULAS

Abre el código fuente de `echo-tcp-basico.c` y estudialo con atención. Vas a modificarlo para que, antes de devolver al cliente el mensaje recibido, lo pase a mayúsculas.

Para esto tienes que iterar sobre cada carácter de la cadena, pasando a mayúsculas cada uno de ellos. La función para pasar un carácter a mayúsculas es `toupper()`. Para saber cuándo parar el bucle, tienes dos alternativas:

- La función `strlen()` devuelve cuántos caracteres tiene una cadena. Puedes hacer un bucle `for` que se repita ese número de veces.
- Puedes hacer un bucle `while` que compruebe si estás en el último carácter de la cadena. Esto se detecta porque el último carácter de una cadena en C es el ASCII nulo, que es un byte de valor cero[1].

Usando `telnet` como cliente, comprueba si recibes correctamente el eco, ahora en mayúsculas.

2.3.2. Un cliente para probar el servidor `echo` UDP

Como hemos dicho, `telnet` puede ser usado para probar el servidor TCP, pero no el UDP. Para poder probar éste, necesitaremos programar nuestro propio cliente.

Tomando como modelo el código fuente del servidor UDP que se suministra (`servidor-udp-basico.c`), y que deberás estudiar hasta comprenderlo bien, escribe un cliente UDP. Su mecánica sería la siguiente:

- Crea un socket UDP (puedes copiar la función `InicializarSocketUDP()` del servidor).

[1] Ojo, nos referimos al valor entero 0, y no al carácter '0'. Es decir, la comparación a realizar es `while (cadena[i]!=0)`

- Entra en un bucle que repita lo siguiente:
  - Imprime en pantalla un "prompt", por ejemplo "->", y lee lo que escriba el usuario
  - Si el texto que has leido es `"FIN"`, termina.
  - Si no, envía lo que el usuario ha escrito al servidor, usando la función `sendto()`. Observa que necesitas proporcionar a esta función una IP y un número de puerto. Usa los de la máquina en la que tengas funcionando tu servidor. Para averiguar la IP de una máquina puedes poner en la línea de comandos "`host` *`máquina`*".
  - Espera a que el servidor te envíe una respuesta, usando la función `recvfrom()`.
  - Imprime en pantalla esta respuesta, usando `printf("%s")`[2].

El bucle puedes programarlo dentro de un procedimiento que podrías llamar, por ejemplo, `EnviarYRecibir()`. Puedes necesitar alguna de las funciones siguientes (usa el comando `man` para averiguar qué `#includes` necesitas y más información sobre la sintaxis):

`scanf()` Esta función se usa para leer texto de la entrada estándar (es decir, lo que teclea el usuario) y convertirlo a alguno de los tipos de datos del C, típicamente enteros o reales.

En teoría también puede usarse para leer una cadena de caracteres (usando el especificador de formato `"%s"`), pero ocurre que en este caso sólo se leen caracteres hasta encontrar un espacio. Si se desea leer una línea de texto completa, aún cuando pueda contener espacios, es mejor usar una de las siguientes funciones.

`gets()` Esta función lee una cadena de la entrada estándar (es decir, lo que teclea el usuario), y la almacena en la dirección que recibe como parámetro. Esta cadena contendrá todo lo escrito por el usuario hasta que pulse retorno de carro, incluyendo los espacios (y el propio retorno de carro al final). Además, automáticamente añadirá un ASCII nulo tras dicho retorno de carro, para marcar el fin de cadena.

Es responsabilidad del programador reservar espacio suficiente para almacenar esta cadena. Ejemplo de uso:

```
char almacen[200]; // Hay sitio para 198 letras, más retorno de
                   // carro, más terminador (ASCII nulo)
printf("Escribe algo: ");
gets(almacen);
printf("Has escrito: |%s|\n");
```

Fíjate que si el usuario escribiera más de 198 letras tendríamos un problema, puesto que no cabrían en el espacio reservado. Ya que no podemos saber cuánto va a escribir el usuario, actualmente el uso de `gets()` se considera peligroso y un potencial problema de seguridad. Se recomienda mejor usar `fgets()` que se describe a continuación.

`fgets()` Su cometido es similar a `gets()`, pero permite especificar de qué fichero queremos leer la cadena y un tamaño máximo a leer[3]. Si el usuario tecleara más, el resto no sería leído (quedaría en un *buffer* y se leería en el próximo `fgets()`, o `scanf()`.

En el caso en que no queramos leer de fichero sino de la entrada estándar (el teclado), especificaremos `stdin` como descriptor de fichero. Ejemplo de uso:

```
char almacen[200]; // Hay sitio para 198 letras, más retorno de
                   // carro, más terminador (ASCII nulo)
printf("Escribe algo: ");
fgets(almacen, 199, stdin);  // Uno menos, por el terminador
printf("Has escrito: |%s|\n");
```

`strlen()` Esta función recibe un puntero a una cadena de caracteres y devuelve un entero indicando cuántos caracteres tiene la cadena. Su funcionamiento es muy simple: se limita a contar cuántos bytes hay hasta encontrar el primer byte igual a cero (que será el terminador). El propio terminador no es contado.

[2] A diferencia de `scanf()`, `printf()` imprime la cadena completa, y no se detiene en los espacios.

[3] En este tamaño se cuenta el retorno de carro, ya que este también es leído, pero no el terminador de cadena, ya que éste no es leido sino añadido por la función

Esta función puede serte útil para decirle a `sendto()` cuántos caracteres debe enviar al servidor. Ten en cuenta que si quieres enviar también el terminador, deberás especificar un carácter más. Estudia el código fuente del servidor para ver si necesita o no que le envíes ese terminador.

`strcmp()` Recibe dos punteros a dos cadenas y las compara alfabéticamente. Retorna -1 si la primera cadena es "menor", 0 si son iguales y 1 si la primera es "mayor".

Puedes usar esta función para detectar cuándo el usuario ha escrito "`FIN`". Sin embargo ten en cuenta que la cadena que el usuario haya tecleado probablemente tendrá un retorno de carro al final, por lo que deberías eliminarlo antes de hacer la comparación[4], o bien comparar con la cadena `"FIN\n"` que incluya el retorno de carro.

3. Comunicando con Windows

Podemos hacer una prueba de comunicación entre una máquina Windows y una Unix. Para ello lanza el servidor en la máquina Linux y después, desde un interfaz de comandos de Windows, usa `telnet` para conectar. Verás que el cliente `telnet` de Windows no se comporta exactamente igual que el de Linux, pero la comunicación debería funcionar.

3.1. Cliente UDP para Windows

- Copia a tu cuenta de Windows el cliente UDP que has escrito para Linux. Renómbralo poniéndole extensión `.cpp` (el compilador de Microsoft necesita esta extensión para compilar correctamente la sintaxis de este fichero).
- Modifícalo para que pueda compilar en Windows (utiliza las transparencias de teoría como guía).
- Compílalo. Recuerda que antes de intentar la compilación debes haber ejecutado `VCVARS32.BAT` en la ventana de la interfaz de comandos. Ya que además hay que especificar una biblioteca de sockets a la hora de compilar, debes usar la siguiente orden:

```
> CL /W3 CLIENTE-ECHO-UDP.CPP WS2_32.LIB
```

- Si consigues compilarlo sin errores, pruébalo. Puedes tener una ventana abierta en la que veas tu servidor (Linux) y un interfaz de comandos en el que ejecutes tu cliente (Windows). Prueba a enviar textos que contengan acentos.

[4] Eliminar el último carácter de una cadena es sencillo. Basta poner un byte de valor cero en su lugar.

4. Anexo: código fuente

4.1. echo-tcp-basico.c

```
1  // Implementación de un servidor que implementa  el servicio ECHO por
2  // el protocolo TCP (RCF 0862)
3  #include <sys/socket.h> // socket, send, recv...
4  #include <netinet/in.h> // sockaddr_in, htons, htonl, etc.
5  #include <arpa/inet.h>  // inet_addr
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <unistd.h>
9  #include <string.h>
10
11 // Definir esta constante apropiadamente. Por ejemplo, usar las
12 // últimas 4 ó 5 cifras del DNI, para evitar que dos alumnos asignen
13 // el mismo número de puerto.
14 #define PUERTO ????
15 #define MAX_LINEA 80 // Maxima longitud esperada para una lectura
16
17 // Esta función crea el socket de escucha, le asigna el número de
18 // puerto y le pone en modo "listen". Retorna el socket creado.
19 // El código de la función está más abajo (después del main())
20 int InicializarSocketEscucha(int puerto);
21
22 // Esta función hace una lectura del socket de datos, Si recibe cero,
23 // retorna cero. En otro caso, muestra por pantalla lo que ha recibido
24 // y lo envía de nuevo por el socket de datos, y retorna el número de
25 // caracteres enviados
26 int RecibirYEnviar(int socket_datos);
27
28 // Programa principal.
29 // Crea un socket de escucha y queda a la espera de un cliente. Todo
30 // lo que reciba, lo envia de nuevo al cliente. Cuando el cliente
31 // cierre, cierra el socket de datos y queda a la espera de un nuevo
32 // cliente.
33 int main(int argc, char *argv[])
34 {
35   int sockEscucha; // Socket por el que se esperan clientes
36   int sockDatos;   // Socket por el que se reciben datos de los
37                    // clientes y se envía la respuesta
38   struct sockaddr_in dir_cliente;
39   int long_dir_cliente=sizeof(dir_cliente);
40   int recibidos;
41
42
43   // Informar al usuario del número de puerto que se usará
44   printf("Servidor TCP: usando el puerto por defecto = % d\n", PUERTO);
45
46   // Eliminar el buffer de printf, para que los printfs tengan efecto
47   // inmediato y sea más fácil depurar el programa
48   // setvbuf(stdout, NULL, _IONBF, 0);
49
50   sockEscucha=InicializarSocketEscucha(PUERTO);
51   // Bucle infinito, se repite para cada cliente que llegue
52   while (1) {
53     // Esperar a que llegue un cliente
54     printf("Esperando por clientes...");
55     sockDatos=accept(sockEscucha, (struct sockaddr *) &dir_cliente,
56                      &long_dir_cliente);
57     // Imprimimos información sobre el cliente que se ha conectado
58     printf("Se ha conectado un cliente\n");
59     // Imprimir la IP y el puerto del cliente:
60     //    (A RELLENAR POR EL ALUMNO)
61
62     // Mientras recibamos datos, le devolvemos el eco
63     do {
64       recibidos=RecibirYEnviar(sockDatos);
65     } while (recibidos!=0);
66     // Cuando se dejan de recibir datos, cerramos el socket de datos y
67     // esperamos por otro cliente
68     printf("El cliente ha cerrado\n");
```

```
69      close(sockDatos);
70    }
71  }
72
73  //================================================================
74  // Código de las funciones auxiliares
75  int InicializarSocketEscucha(int puerto)
76  {
77    int ret;
78    struct sockaddr_in dir;
79    int socket_escucha;
80
81    // Crear el socket
82    socket_escucha=socket(PF_INET, SOCK_STREAM, 0);
83    if (socket_escucha==-1) {
84      perror("Al crear socket");
85      exit(-1);
86    }
87    // Asignarle el numero de puerto
88    dir.sin_family=AF_INET;
89    dir.sin_port=htons(puerto);
90    dir.sin_addr.s_addr=htonl(INADDR_ANY);
91    ret=bind(socket_escucha,(struct sockaddr *)  &dir, sizeof(dir));
92    if (ret==-1) {
93      perror("Al asignar direccion");
94      close(socket_escucha);
95      exit(-1);
96    }
97    // Ponerlo en modo escucha
98    ret=listen(socket_escucha, SOMAXCONN);
99    if (ret==-1) {
100     perror("Al poner en modo escucha");
101     close(socket_escucha);
102     exit(-1);
103   }
104   // El socket ya está preparado para hacer accept()
105   return socket_escucha;
106 }
107
108 //================================================================
109 // La función que hace "echo"
110 // Recibe por el socket de datos, muestra lo recibido y lo reenvía
111 int RecibirYEnviar(int socket_datos)
112 {
113   int leidos;
114   char buffer[MAX_LINEA];
115
116   // Recibir mensaje
117   leidos=read(socket_datos, buffer, MAX_LINEA);
118   if (leidos==0) return 0;
119
120   // Mostrar información por pantalla
121   printf("Recibido mensaje con %d caracteres\n", leidos);
122   buffer[leidos]=0;  // Hay que añadir un terminador a la cadena, para
123                      // poder imprimirla con %s
124   printf("Mensaje: %s\n", buffer);
125
126   // Devolver el mensaje
127   write(socket_datos, buffer, leidos);
128   // Observar que no escribo MAX_LINEA caracteres, sino solo los
129   // "leidos"
130   return leidos;
131 }
132
133
134
```

4.2. echo-udp-basico.c

```
1  // Implementación de un servidor para probar el servicio ECHO,
2  // protocolo UDP (RCF 0862)
3  #include <sys/socket.h> // socket, send, recv...
4  #include <netinet/in.h> // sockaddr_in, htons, htonl, etc.
5  #include <arpa/inet.h>  // inet_addr
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <unistd.h>
9  #include <string.h>
10
11 // Definir esta constante apropiadamente. Por ejemplo, usar las
12 // últimas 4 ó 5 cifras del DNI, para evitar que dos alumnos asignen
13 // el mismo número de puerto.
14 #define PUERTO ????
15 #define MAX_LINEA 80
16
17 // Esta función crea el socket de escucha, le asigna el número de
18 // puerto y le pone en modo "listen". Retorna el socket creado.
19 // El código de la función está más abajo (después del main())
20 int InicializarSocketUDP(int puerto);
21
22 // Esta función intenta recvfrom del socket. Si no hay datos, se
23 // bloqueará hasta que un ciente envíe algo, y entonces mostrará por
24 // pantalla lo que ha recibido y se lo reenviará al cliente.
25 void RecibirYEnviar(int socket_datos);
26
27 // Programa principal.
28 // Crea un socket UDP y entra en un bucle infinito en que llama a
29 // RecibirYEnviar una y otra vez.
30 int main(int argc, char *argv[])
31 {
32   int  socketUDP;
33
34   // Informar al usuario del número de puerto que se usará
35   printf("Servidor UDP: usando el puerto por defecto = % d\n", PUERTO);
36
37   // Eliminar el buffer de printf, para que los printfs tengan efecto
38   // inmediato y sea más fácil depurar el programa
39   // setvbuf(stdout, NULL, _IONBF, 0);
40
41   socketUDP=InicializarSocketUDP(PUERTO);
42
43   while (1)   {
44     printf("Esperando mensajes...");
45     RecibirYEnviar(socketUDP);
46   }
47 }
48
49 //============================================================
50 // Código de las funciones auxiliares
51 int InicializarSocketUDP(int puerto)
52 {
53   int ret;
54   struct sockaddr_in dir;
55   int socket_UDP;
56
57   // Crear el socket
58   socket_UDP=socket(PF_INET, SOCK_DGRAM, 0); // DGRAM
59   if (socket_UDP==-1) {
60     perror("Al crear socket");
61     exit(-1);
62   }
63   // Asignarle el numero de puerto
64   dir.sin_family=AF_INET;
65   dir.sin_port=htons(puerto);
66   dir.sin_addr.s_addr=htonl(INADDR_ANY);
67   ret=bind(socket_UDP,(struct sockaddr *)  &dir, sizeof(dir));
68   if (ret==-1) {
69     perror("Al asignar direccion");
70     close(socket_UDP);
71     exit(-1);
```

```
72    }
73    // NO SE HACE listen()
74    // El socket ya está preparado para recibir
75    return socket_UDP;
76  }
77
78  void RecibirYEnviar(int socket_udp)
79  {
80    struct sockaddr_in dir_cliente;
81    int long_dir_cliente=sizeof(dir_cliente);
82    int leidos;
83    char buffer[MAX_LINEA];
84
85    leidos=recvfrom(socket_udp, buffer, MAX_LINEA, 0,
86                    (struct sockaddr *)&dir_cliente, &long_dir_cliente);
87    if (leidos<0) {
88      perror("Al recibir");
89      close(socket_udp);  exit(-1);
90    }
91    if (leidos==0) {
92      printf("El cliente ha cerrado la conexión\n");
93    } else {
94      printf("Recibidos %d caracteres\n", leidos);
95      // poner terminador al buffer para poder mostrarlo con %s
96      buffer[leidos]=0;
97      printf("Mensaje: %s\n", buffer);
98      // Enviar de nuevo el mensaje al cliente
99      sendto(socket_udp, buffer, leidos, 0,
100             (struct sockaddr *) &dir_cliente, long_dir_cliente);
101   }
102 }
```

Índice