Redes de Computadoras
Manual de Prácticas

Andrés Marín López

Febrero de 1998

Índice General

1 Introducción

El objetivo de este manual de prácticas es dar una visión general sobre el diseño e implementación de clientes y servidores utilizando sockets de TCP/IP. Primero se explican conceptos relativos a los sockets y consideraciones de diseño de clientes y servidores. A continuación se ilustra el uso de la interfaz de sockets desde Java y C mediante algunos ejemplos. El alumno debe probar en el laboratorio los ejemplos que aparecen en el manual. Conforme avanza el manual, disminuye el detalle de los ejemplos y aumenta la tarea del lector. La sección 6 presenta la especificación de la **práctica obligatoria** que deben realizar los alumnos.
El presente manual se ha inspirado en un libro clásico [1], bibliografía muy útil para los desarrolladores de aplicaciones clientes/servidores que utilicen sockets tanto con BSD UNIX, System V o winsock.

2 El interfaz de sockets

Aunque TCP/IP no define un API (interfaz con los programas de aplicación), los estándares sugieren cuál es la funcionalidad esperada:

- Asignar recursos locales para la comunicación.
- Especificar puntos de comunicación locales y remotos.
- Iniciar o esperar una comunicación.
- Enviar o recibir datos.
- Determinar cuando llegan los datos.
- Generar datos urgentes.
- Manejar datos de entrada urgentes.
- Terminar una conexión adecuadamente.
- Gestionar la terminación de una conexión desde el lado remoto.
- Abortar la comunicación.

- Gestionar condiciones de error y comunicaciones abortadas.

- Liberar recursos locales cuando termina la comunicación.

El sistema operativo permite el uso de llamadas al sistema para transferir el control desde una aplicación al sistema operativo para utilizar los servicios que éste ofrece. Desde el punto de vista de la aplicación son llamadas a funciones que permiten gestionar recursos de bajo nivel como la entrada/salida. La gran diferencia es que el sistema operativo opera en un modo privilegiado que le permite acceder a todos los recursos, y la aplicación no tiene que leer ni modificar las estructuras del sistema operativo sino que las llamadas al sistema lo hacen por ella.
En UNIX se definen seis funciones básicas de entrada/salida. El interfaz de sockets utiliza estas funciones junto con algunas otras. El uso de estas funciones se describe a continuación.
Una aplicación llama a `open` para iniciar una entrada/salida, dicha operación devuelve un entero llamado *descriptor* que utilizará la aplicación para las demás operaciones. A continuación el programa puede utilizar las funciones de `read` o `write` para leer o escribir datos o `lseek` para posicionarse en un determinado lugar. Una vez la aplicación ha terminado sus lecturas y escrituras, llama a la función `close`. Por último la función `ioctl` permite enviar comandos de bajo nivel al dispositivo en uso. Por ejemplo, fijar parámetros en un dispositivo de red tales como si el interfaz debe escuchar en modo promiscuo o multicast, etc.

2.1 La llamada a socket

Los descriptores que obtienen las aplicaciones de una llamada a open dependen del tipo de recurso que estén abriendo. Una operación de apertura de un fichero devolverá un descriptor de fichero, mientras que una operación de apertura de un socket devolverá un descriptor del socket abierto. La llamada de apertura de un socket se denomina *socket*.
El sistema operativo mantiene las tablas de descriptores asignados a los procesos. En la tabla se referencian descriptores a distintos tipos de recursos, es decir, en la misma tabla se mantienen los descriptores de ficheros y los descriptores de sockets y corresponde a cada aplicación saber que tipo de recurso está asociado a cada descriptor. El sistema utiliza estas tablas pa-

ra referenciar las estructuras de datos que controlan el estado del recurso representado por el descriptor.
Cuando una aplicación abre un socket, el sistema operativo se encarga de inicializar las estructuras que gestionarán la comunicación, añade una referencia a dichas estructuras en la tabla de sockets del proceso y devuelve como descriptor el índice a la posición de la tabla en la que se almacenó dicha referencia como se muestra en la figura 1 en la página 4.
La llamada a socket tiene el siguiente prototipo en C:

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

Donde `domain` representa el dominio de comunicaciones en el que tendrá lugar la comunicación (familia de protocolos), `type` representa el tipo de socket que se desea abrir y `protocol` el tipo de protocolo a utilizar (vease `/etc/protocols`). La tabla 1 de la página 5 contiene todos los posibles valores de estos parámetros. En la tabla aparecen en **negrita** los valores que vamos a usar.

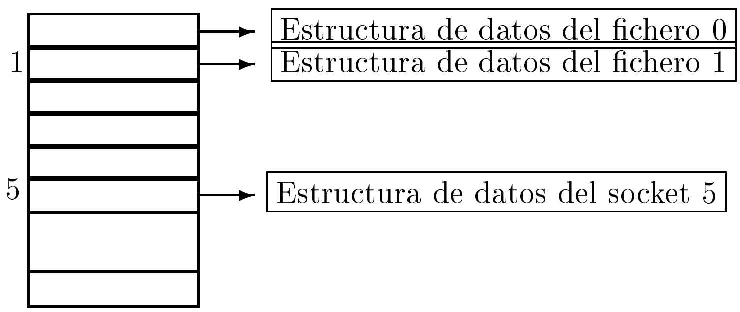

Figura 1: Representación de la tabla de descriptores de un proceso

La estructura de datos que mantiene el sistema operativo por cada socket abierto contiene indicación de la familia (`PF_INET` en el caso de TCP/IP), el servicio, las direcciones IP local y remota, los puertos local y remoto, etc.

Familias de sockets	
AF_UNIX	protocolos internos UNIX
AF_INET	protocolos Internet ARPA
AF_ISO	protocolos ISO
AF_NS	Protocolos de sistemas Xerox Network
AF_IMPLINK IMP	"host a IMP"nivel de enlace
Tipos de sockets	
SOCK_STREAM	stream fiable orientado a conexión
SOCK_DGRAM	datagramas, no fiable
SOCK_RAW	root (datagramas) acceso total
Protocolos de sockets	
ip	internet protocol, pseudo protocol number
icmp	internet control message protocol
igmp	internet group multicast protocol
ggp	gateway-gateway protocol
tcp	transmission control protocol
pup	PARC universal packet protocol
udp	user datagram protocol
raw	RAW IP interface

Tabla 1: Parámetros de los sockets BSD

En la llamada a open no se llenan todos estos campos, y la aplicación deberá realizar sucesivas llamadas al sistema antes de usar el socket.
Una vez creado un socket puede ser usado por un servidor para esperar conexiones entrantes (**socket pasivo**), o bien por un cliente para iniciar conexiones (**socket activo**). En cualquier caso, para realizar una comunicación, hay que conocer la dirección IP y el puerto local en los que se sitúa la otra parte de la comunicación. Para ello, el interfaz de sockets permite especificar las direcciones y los puertos (locales y remotos) que representan unívocamente la comunicación.
Aunque el interfaz de sockets permite distintas familias de direcciones, en este manual nos referiremos siempre a TCP/IP cuya familia de direcciones se denota con la constante `AF_INET`. Las aplicaciones que usan TCP/IP exclusivamente puede usar la estructura de direcciones `sockaddr_in` que detallamos

(en lenguaje C) en la figura 2 de la página 6:

```
struct sockaddr_in {
  u_char  sin_len;          /* longitud total */
  u_short sin_family;       /* familia de direcciones */
  u_short sin_port;         /* numero de puerto */
  struct  in_addr sin_addr;/* direccion IP(u_long) */
  char    sin_zero;         /* no usado (0) */
}
```

Tabla 2: Estructura de direcciones de sockets TCP/IP

Respecto a los constructores de sockets en Java, los veremos más adelante en la sección 4.1 de la página 14 y posteriores.

2.2 La llamada a connect

Una vez creado el socket, un cliente llama a `connect` para establecer una conexión remota con un servidor. El cliente debe especificar el punto de acceso remoto. Una vez conectado el socket, el cliente puede leer y escribir datos en él.
El prototipo de la llamada es el siguiente:

```
int connect(
    int sockfd, /* descriptor del socket */
    struct sockaddr *s_addr, /* direccion del otro extremo */
    int addrlen ); /* longitud de la direccion (en bytes) */
```

Si el socket es de tipo `SOCK_DGRAM` la llamada a socket establece la única dirección desde la que se recibirán y a la que se mandarán datagramas.

2.3 La llamada a write

Tanto los clientes como los servidores usan `write` para enviar datos a través del socket. Para escribir datos en un socket hay que especificar el buffer que contiene los datos a escribir asi como la longitud del buffer. `write` copia los datos en buffers del kernel y devuelve el control a la aplicación. Si todos los buffers del kernel están llenos, la aplicación queda bloqueada hasta que se libera el espacio suficiente. La llamada es la misma que se utiliza para escribir `count` datos de un buffer `buf` en un descriptor de fichero `fd`:

```
size_t write(int fd, const char *buf, size_t count);
```

`size_t` normalmente se define como:

```
typedef unsigned int size_t;
```

2.4 La llamada a read

Una vez conectado, el cliente escribe su petición y lee la respuesta que manda el servidor. Para leer datos de un socket hay que especificar un buffer en el que deben situarse los datos recibidos, asi como la longitud del mismo. La operación de lectura copia en el buffer del usuario los datos recibidos. Si no ha llegado ningun dato, la aplicación queda bloqueada hasta que se reciba alguno. Si llegan más datos que la longitud del buffer del usuario, `read` solo extrae del socket el numero suficiente para llenar el buffer del usuario. Si llegan menos datos de los que caben en el buffer, `read` copia todos los datos y devuelve el número de bytes recibidos.
También se puede usar `read` con sockets que usen UDP. `read` copia el datagrama recibido en el buffer del usuario. En caso de que el datagrama sea más grande que el buffer de usuario, se copian todos los posibles y se descarta el resto.
La llamada es la misma que se utiliza para leer `count` datos de un descriptor de fichero `fd` en un buffer `buf`:

```
size_t read(int fd, const char *buf, size_t count);
```

2.5 La llamada a close

Una vez que un cliente o un servidor terminan de usar un socket, llaman a `close` para liberarlo. Si solo hay un proceso usando el socket, `close` termina la conexión y libera el socket. Si hay varios procesos compartiendo un socket, `close` decrementa un contador de uso. El socket se libera cuando el contador llega a cero.

En determinados protocolos puede resultar conveniente que cualquiera de los extremos de la comunicación señalice que no tiene más datos que enviar. El interfaz de sockets permite usar llamadas a `shutdown` para este fin. El programa en el otro extremo (cliente o servidor) recibe una señal de fin de fichero.
La llamada es la misma que la que se utiliza para cerrar un fichero a partir del descriptor de fichero `fd`:

```
int close(int fd);
```

Si se intenta cerrar un socket en el cual la cola de envios pendientes no está vacía, la llamada bloqueará o no al proceso llamante dependiendo de una opción (`SO_LINGER`) que se le puede indicar al socket mediante la llamada `setsockopt`. Remitimos al lector interesado a la bibliografía o bien a la página del manual correspondiente **setsockopt(2)**.

2.6 La llamada a bind

Cuando el socket se crea, no tiene asignados los puntos de acceso local ni remoto. Cuando una aplicación llama a `bind`, fija el punto de acceso local (dirección y puerto) del socket. En general, los servidores usan `bind` para especificar el puerto conocido (*well-known port*) en el que esperan conexiones de los clientes.
La sintaxis de la llamada es la siguiente:

```
int bind(int  sockfd,  struct  sockaddr  *my_addr,  int
     addrlen);
```

2.7 La llamada a listen

Los servidores orientados a conexión llaman a `listen` para poner el socket en modo pasivo y prepararlo para aceptar llamadas entrantes. En general los servidores tienen un bucle en el que aceptan llamadas entrantes, las gestionan y vuelven a esperar la siguiente conexión. Durante el tiempo de gestión de la conexión se pueden perder nuevos intentos de conexiones. El servidor puede indicar al kernel que encole las conexiones recibidas por el socket. `listen` permite fijar el tamaño de la cola de peticiones, además de poner al socket en modo pasivo.
El prototipo de la llamada es el siguiente:

```
int listen(int sockfd,
           int backlog); /* longitud cola de peticiones */
```

2.8 La llamada a accept

Un servidor llama a `accept` para extraer la siguiente petición de conexión del socket especificado. `accept` crea un nuevo socket para cada nueva petición de conexión y devuelve el descriptor del nuevo socket. El servidor utiliza el nuevo socket únicamente para gestionar la nueva conexión, y el original para atender a nuevas peticiones. Una vez aceptada una conexión el servidor puede leer y escribir datos en el nuevo socket, y deberá cerrarlo cuando termine de utilizarlo.
El prototipo de la llamada es el siguiente:

```
int accept(int sockfd, struct sockaddr *addr, int *addrlen);
```

3 Uso de sockets en aplicaciones

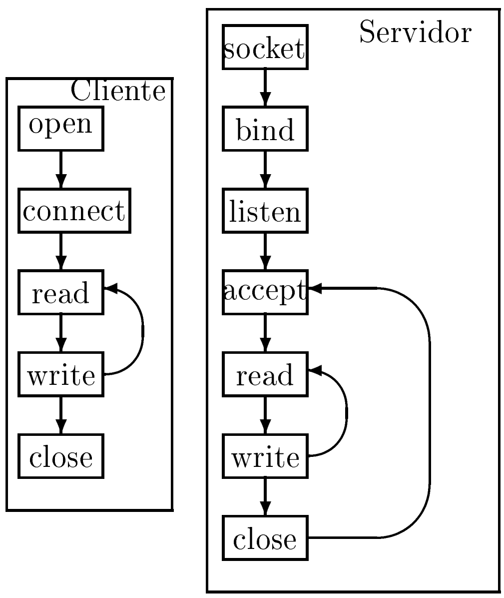

Figura 2: Uso de sockets en un programa

En esta sección vamos a estudiar la secuencia de llamadas que realizan los clientes y los servidores a la interfaz de sockets, ilustradas en la figura 2 de la página 9.
El cliente llama a `socket`, llama a `connect` para conectarse con el servidor (debe de conocer la dirección y el puerto en este punto), usa `read` y `write` para enviar y recibir datos del servidor, y por último llama a `close` para cerrar el socket.
El servidor llama a `socket` y llama a `bind` para especificar el punto de acceso local del servidor. A continuación llama a `listen` para especificar el tamaño de la cola de peticiones de conexión y a continuación entra en un bucle. En el bucle, el servidor llama a `accept` para quedarse a la espera de una llamada. Cuando se recibe una llamada, `accept` devuelve un descriptor de un nuevo socket a través del cual el servidor pasa a dar servicio (utilizando `read` y `write`) al cliente. Una vez atendido el cliente, llama a `close` para cerrar el socket devuelto por `accept` y vuelve al bucle principal en una nueva llamada a `accept`, a la espera de nuevas conexiones.
Si se quiere diseñar un servidor concurrente se creará un proceso hijo para dar servicio a cada petición. El servidor, tras la llamada a `listen`, entrará en un bucle en el cual:

1. llama a `accept`

2. crea un proceso hijo con los parámetros necesarios (al menos el identificador del socket devuelto por `accept`).

3.1 Formato de enteros

En TCP/IP se especifica que los enteros se representan con el byte más significativo al principio (representación big-endian). Sin embargo muchas arquitecturas muy populares (como las de Intel 80x86) utilizan otra representación (little-endian) con el byte más significativo al final. En la tabla 3 de la página 11 se dan las dos representaciones del entero 34.677.374.

Para que los programas se puedan migrar de una arquitectura a otra sin problemas, el sistema operativo ofrece las funciones de conversión de la representación usada en la red (a partir de ahora nos referiremos a ella como

	(2)	(17)	(34)	(126)
big endian	00000010	00010001	00100010	01111110

	(126)	(34)	(17)	(2)
little endian	01111110	00100010	00010001	00000010

Tabla 3: Formatos de representación de enteros

orden de red) a la de la máquina en la que corre el sistema operativo. Las funciones se describen en la tabla 4 de la página 11. **Si el programador utiliza estas funciones para convertir enteros del formato utilizado en la red al utilizado en la máquina, no tendrá problemas de portabilidad**.

	Conversión		Tipo de
Función	De	A	dato
`ntohs`	red	host	short
`ntohl`	red	host	long
`htons`	host	red	short
`htonl`	host	red	long

Tabla 4: Facilidades del S.O. para conversión de enteros

Este problema es transparente al programador de Java, dado que la representación de los enteros está fijada como big-endian y las funciones de las clases del paquete `java.net` ya lo tienen en cuenta.

3.2 `gethostbyname`, `getservbyname`, `getprotobyname`

La función `inet_addr()` convierte una dirección IP de formato con números y puntos al correspondiente formato binario y orden de bytes de la red. Si la entrada no es válida devuelve -1. Esta función está obsoleta y se prefiere usar `inet_aton()` que hace lo mismo pero almacena el valor convertido en una estructura de tipo `in_addr` y devuelve cero si la dirección no es válida. Los prototipos de las funciones y la estructura `in_addr` se muestran en la figura 3 de la página 12.

```
int inet_aton(const char *cp, struct in_addr *inp);

unsigned long int inet_addr(const char *cp);

struct in_addr {
  unsigned long int s_addr;
}
```

Figura 3: Prototipos y estructura para traducciones de direcciones en C

La función `gethostbyname()` devuelve una estructura de tipo `hostent` para el nombre del host especificado. Si el nombre no acaba en punto, también se busca el dominio actual y sus padres. El prototipo de la función es:

```
struct hostent *gethostbyname(const char *name);
```

La estructura `hostent` se define de la siguiente forma:

```
struct hostent {
  char    *h_name;
  char    **h_aliases;
  int     h_addrtype;
  int     h_length;
  char    **h_addr_list;
}
#define h_addr  h_addr_list[0]
```

Los miembros de la estructura `hostent` son los siguientes:

`h_name` Nombre oficial del host

`h_aliases` Array de nombres alternativos para el host (alias que pueden haberse definido para el host).

`h_addrtype` Tipo de dirección (de momento siempre `AF_INET`).

`h_length` longitud de la dirección en bytes.

`h_addr_list` Array de direcciones de red para el host en orden de red (un host puede tener más de una tarjeta de red, etc.)

`h_addr` primera dirección en el array `h_addr_list` (para compatibilidad hacia atrás).

La función`getservbyname()` devuelve una estructura `servent` con información de la línea del fichero `/etc/services` con el protocolo especificado. El prototipo de la función es el siguiente:

```
struct servent *getservbyname(const char *name,
                              const char *proto);
```

La estructura `servent` se define de la siguiente forma:

```
struct servent {
  char    *s_name;
  char    **s_aliases;
  int     s_port;
  char    *s_proto;
}
```

Los miembros de la estructura `servent` son los siguientes:

`s_name` Nombre oficial del servicio.

`s_aliases` Lista de nombres alternativos del servicio.

`s_port` Número del puerto del servicio en orden de red.

`s_proto` Nombre del protocolo a usar con este servicio.

La función `getprotobyname()` devuelve una estructura de tipo `protoent` con información de la línea del fichero `/etc/protocols` que coincide con el nombre del protocolo especificado.

```
struct protoent *getprotobyname(const char *name);
```

La estructura `protoent` se define de la siguiente forma:

```
struct protoent {
   char    *p_name;
   char    **p_aliases;
   int     p_proto;
}
```

Los miembros de la estructura `protoent` son los siguientes:

`p_name` nombre oficial del protocolo.

`p_aliases` lista de nombres alternativos del protocolo.

`p_proto` número del protocolo.

Las funciones `gethostbyname, getservbyname, getprotobyname` devuelven `NULL` en caso de que el nombre del host, servicio o protocolo sean incorrectos.

4 Un cliente de fecha y hora

Los estándares TCP/IP definen un protocolo en la RFC (Request For Comments) 868 que permite a un usuario obtener la fecha y hora en un formato legible por personas. El servicio se denomina `DAYTIME`.
El usuario utiliza un cliente de `DAYTIME` [1] que devuelve al usuario los datos que a su vez obtuvo del servidor. El resultado tiene esta forma:

```
Tue Jan 13 13:14:17 1998
```

El protocolo está disponible tanto para TCP como para UDP. En la versión para TCP, en cuanto el servidor detecta una conexión, construye un string con los datos, lo devuelve al cliente y cierra la conexión.

4.1 Implementación de un cliente TCP de `DAYTIME`

Una implementación en C del cliente TCP de `DAYTIME` la encontramos en [1] y la adjuntamos en la figura 4 de la página 15.

[1]El servicio se puede probar en unix con el comando `rdate`

```
void TCPdaytime(const char *host)
{
  char buf[LINELEN]; /* buffer para una linea de texto */
  int s, n;          /* socket, contador de lectura */
  s = connectTCP(host, "daytime");
  while((n = read(s, buf, LINELEN)) > 0) {
    buf[n] = '\0'; (void) fputs(buf, stdout);
  }
}

int connectTCP(const char *host, const char *service)
{
  return connectsock(host,service,"tcp");
}
```

Figura 4: Cliente TCP de `DAYTIME` en C

El cliente utiliza una función `connectTCP()` para crear el socket y conectarse con el servidor. `connectTCP()` llama a su vez a `connectsock()` y son funciones definidas para lograr una mayor modularidad y legibilidad en el código. En la figura 12 de la página 39 se presenta el código escrito por Comer y Stevens para `conectsock()`.
En la figura 5 de la página 16 se muestra una implementación en Java de un cliente TCP de `DAYTIME`. El interfaz de sockets en Java es mucho más simple que en C. Por tanto su uso es más sencillo pero la funcionalidad es menor (el acceso a las estructuras es más restringido). El paquete `java.net` ofrece una clase abstracta `SocketImpl` que permite crear sockets específicos para aplicaciones, sin embargo en la mayoría de los casos es suficiente con las implementaciones definidas en el paquete.
En la figura 5 de la página 16 se utiliza el constructor `Socket(String, int)` que crea un socket y lo conecta a la dirección y puerto especificados. El constructor está sobrecargado, de foma que se pueden crear sockets dando como parámetros las direcciones local y remota de tipo `InetAddress`, solo la dirección remota, etc. También existen constructores para crear sockets sin conectarlos, sobre los que se puede invocar posteriormente el método

```
public String TCPdaytime(String host)
{
  Socket s; /* socket */
  InputStream sin;

  try{
    s = new Socket(host, "daytime");
    sin =s.getInputStream();

    byte buf[LINELEN];
    int rec;
    while((rec= sin.read(buf)) != -1) {
      String texto= new String(buf,0,rec);
      System.out.println(texto);
    }
  }
  catch (UnknownHostException uh)
    { System.err.println("Host desconocido"); }
  catch (IOException io)
    { System.err.println("Error de I/O"); }
}
```

Figura 5: Cliente TCP de `DAYTIME` en Java

`connect()`.
Una vez creado y conectado el socket, se puede obtener el `InputStream` y leer directamente la respuesta del servidor con los métodos estándares definidos en el paqute `java.io`. En el ejemplo del cliente de `DAYTIME` se invoca el método `read(byte[])` y a continuación se convierte el array de `byte` en un `String` que se imprime por la salida estándar. Otra posibilidad hubiera sido crear un objeto de tipo `InputStreamReader` y con éste un `LineNumberReader` de la siguiente forma:

```
LineNumberReader lnrin=
  new LineNumberReader(
```

```
      new InputStreamReader(sin));
  system.out.println(lnrin.readLine());
```

En general resulta más cómodo crear este tipo de filtros con `Streams` cuando se necesita leer muchos datos del mismo.

4.2 Implementación de un cliente UDP de `DAYTIME`

```
#define UNIXEPOCH   2208988800 /* epoca UNIX(segundos UCT)*/
#define MSG   "Que hora es?\n"

void UDPdaytime(const char *host)
{
  time_t  now;       /* entero de 32 bits para el tiempo */
  int s, n;          /* socket, contador de lectura */
  s = connectUDP(host, "daytime");
  (void) write(s,MSG,strlen(MSG));
  if (n = read(s, (char*)&now,sizeof(now))) < 0)
    error("lectura no valida");
  now = ntohl((u_long)now);
  now -= UNIXEPOCH; /* convierte al formato de hora UNIX */
  printf("%s",ctime(&now));
  exit(0);
}

int connectUDP(const char *host, const char *service)
{
  return connectsock(host,service,"udp");
}
```

Figura 6: Cliente UDP de `DAYTIME` en C

La figura 6 de la página 17 contiene el código dado por [1] de una implementación de un cliente UDP de `DAYTIME`. La función `connectUDP()` se construye a partir de la más general `connectsock()`. Estas dos funciones junto con

`connectTCP()` pueden incorporarse en una biblioteca que facilite el uso de sockets para implementaciones de clientes e incremente la modularidad de los programas resultantes.
Una vez obtenido un descriptor del socket, se manda un datagrama, cuyo contenido es irrelevante. Este datagrama llega al servidor que lo descarta y manda la hora a la dirección de origen del datagrama. El cliente lee del socket la respuesta, la convierte al formato del host y la escribe (utilizando facilidades propias de UNIX como `ctime`).
En la figura 7 de la página 35 se da el código de una implementación de un cliente UDP de `DAYTIME` en Java. Se pueden observar las diferencias con respecto al cliente de TCP de la figura 5 de la página 16 y respecto al código C de la figura 6 de la página 17.

5 Clientes y servidores de eco

Los estándares TCP/IP definen un servicio de eco (`ECHO`) en [2] y [3] para TCP y UDP que devuelven los datos que se reciben de un cliente. Las aplicaciones de este servicio son muy variadas, y se pueden usar para probar protocolos nuevos o implementaciones nuevas, para ver si una máquina es alcanzable y si su estado es adecuado y para identificar problemas de encaminamiento en una red.
El usuario utiliza un cliente de `ECHO` que manda datos al servidor y a continuación los lee de nuevo. El servidor a su vez lee datos de la conexión y los escribe de vuelta.
Las figuras 8 de la página 36 y 9 de la página 36 muestran las implementaciones de [1] en C de clientes TCP y UDP de `ECHO`. Las figuras 10 de la página 37 y 11 de la página 38 muestran las implementaciones en Java de los mismos clientes.
A la hora de realizar una implementación de un servidor, hay que reflexionar sobre varias cuestiones de diseño:

- Número de peticiones que el servidor va a aceptar.
- Política de aceptaciones de clientes.
- Si los clientes van a ser servidos de forma secuencial o concurrente.
- Orientados(no orientados) a conexión.

- Si se van a llevar estadísticas de uso o acceso al servidor.
- Acción a tomar en caso de fallo del cliente.

El número de peticiones que se van a aceptar, además de fijar un tamaño de cola a la hora de crear el socket del servidor, impone los recursos que van a ser necesarios en el servidor (tamaño de tablas y memorias, etc.)
Si se decide seguir una política de aceptaciones, esto conlleva hacer una serie de comprobaciones y anotaciones antes de servir a un cliente. Estas consideraciones son muy importantes en determinados servicios comerciales, ya que este es el punto en el que se incian las actividades de contabilidad de clientes a efectos de tarificación de servicios. Frecuentemente, estos servicios requieren tener en cuenta el tipo de seguridad que se da a los usuarios: ninguna, seguridad básica basada en logins y passwds, cifrado de conexiones, autentificación de clientes y servidores, no repudiación de mensajes, etc.
La consideración acerca de la concurrencia o secuencialidad del servidor es posiblemente la que más drásticamente influye en el diseño. El modelo de servidor secuencial es menos complejo pero lleva a servidores de peores prestaciones, puesto que ocasionan retrasos en los clientes y pueden convertirse en el cuello de botella del sistema. Los servidores concurrentes son más complejos de diseñar pero sus prestaciones son mejores. Elegir un servidor concurrente no implica necesariamente una implementación concurrente (con varios procesos ejecutando en paralelo), y las esperas en operaciones de I/O pueden utilizarse para dar servicio concurrente a distintos clientes.
Los servidores orientados a conexión obtienen de la red un servicio fiable que se encarga de la retransmisión de paquetes cuando esta es necesaria y de entregarlos en orden. Son en general más sencillos de programar. Su principal desventaja es que necesitan un socket por cada cliente al que dan servicio. Además si un cliente cae, el socket y los recursos destinados a ese cliente quedan inutilizados. El problema se complica si los clientes se caen a menudo, pues el servidor puede quedarse sin recursos.
Los servidores no orientados a conexión pueden utilizar un mismo socket UDP para dar servicios a muchos clientes. A cambio, el servidor tiene que encargarse de ordenar los paquetes recibidos y de implementar una estrategia de retransmisión. En caso de que estemos programando un servidor que vaya a prestar servicio en internet, la estrategia de retransmisión puede ser bastante complicada pues tiene que adaptarse a un entorno en el que enlaces, retardos y rutas varían dinámicamente.

A continuación se muestra la implementación definida en [1] de un servidor TCP de `echo`. El servidor usa procesos concurrentes para dar servicio concurrente a múltiples clientes. El sevidor usa la función `passiveTCP()` (ver figura 13 de la página 40) para crear el socket del servidor.

```
/* TCPechod.c - main, TCPechod */

#include <sys/types.h>
#include <sys/signal.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/wait.h>
#include <sys/errno.h>
#include <netinet/in.h>

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define QLEN  5  /* tamano maximo de la cola de conexiones */
#define BUFSIZE 4096

extern int errno;

void reaper(int);
int TCPechod(int fd);
int errexit(const char *format, ...);
int passivesock(const char *service,
                    const char *transport, int qlen);

int
main(int argc, char *argv[])
{
  char   *service = "echo"; /* nombre servicio o num puerto */
  struct sockaddr_in fsin;  /* direccion de un cliente */
```

```
int    alen;  /* longitud de la direccion del cliente */
int    msock; /* socket maestro del servidor */
int    ssock; /* socket esclavo del servidor */

switch (argc) {
case 1:
  break;
case 2:
  service = argv[1];
  break;
default:
  errexit("uso: TCPechod [port]\n");
}

msock = passivesock(service, "tcp", QLEN);

(void) signal(SIGCHLD, reaper);

while (1) {
   alen = sizeof(fsin);
   ssock = accept(msock, (struct sockaddr *)&fsin, &alen);
   if (ssock < 0) {
     if (errno == EINTR)
       continue;
     errexit("accept: %s\n", strerror(errno));
   }
   switch (fork()) {
     case 0:   /* hijo */
       (void) close(msock);
       exit(TCPechod(ssock));
     default: /* padre */
       (void) close(ssock);
       break;
     case -1:
       errexit("fork: %s\n", strerror(errno));
  }
}
```

```
}

int
TCPechod(int fd)
{
  char  buf[BUFSIZ];
  int   cc;

  while (cc = read(fd, buf, sizeof buf)) {
    if (cc < 0)
      errexit("echo read: %s\n", strerror(errno));
      if (write(fd, buf, cc) < 0)
        errexit("echo write: %s\n", strerror(errno));
  }
  return 0;
}

void
reaper(int sig)
{
  int  status;

  while (wait3(&status, WNOHANG, (struct rusage *)0) >= 0)
    /* vacio */;
}
```

El proceso maestro comienza ejecutando `main()`. Tras comprobar sus argumentos, llama a `passivesock()` para crear el socket maestro en el puerto del servicio `ECHO` y a continuación entra en un bucle infinito. Dentro del bucle espera a recibir llamadas entrantes bloqueándose en la llamada a `accept()`. Cuando vuelve de una llamada, obtiene un descriptor de un nuevo socket esclavo que utilizará para atender la nueva petición. El servidor llama a `fork()` para crear un proceso hijo (`fork() == 0`). A continuación el servidor devuelve su descriptor al socket esclavo y vuelve a ejecutar el bucle.
El proceso hijo cierra su descriptor al socket maestro y llama a `TCPechod()` con el descriptor al socket esclavo que devolvió accept al servidor. El proceso termina (pasa a un estado `zombie`) cuando `TCPechod()` termina. Cuando

el proceso hijo termina, UNIX manda al proceso padre la señal `SIGCHLD` indicando la terminación del proceso hijo. El servidor mediante la llamada a `signal(SIGCHLD,reaper)` informa al sistema operativo de que debe ejecutar la función `reaper()` cuando reciba la señal `SIGCHLD`. La función `reaper()` lo que hace es llamar a `wait3()` que se encarga de completar la terminación de un proceso que llamó a `exit()`. El argumento `WNOHANG` hace que la llamada a `wait3` no sea bloqueante.
Veamos a continuación como sería el codigo en Java del mismo servidor:

```
class echoserver{
  ServerSocket maestro;
  Socket esclavo;

  public void arrancar(int puerto) throws Exception{

    maestro=new ServerSocket(puerto,5);
    while (true){
      esclavo=maestro.accept();
      Hijo hijo=new conexion(esclavo, this);
      hijo.start();
    }
  }
}

class Hijo extends Thread{
  private Socket canal;

  public Hijo(Socket s){
    canal=s;
  }
  public void run(){
    /* lee datos del canal y los escribe de nuevo
       se deja como ejercicio al lector.          */
  }
}
```

Se puede observar que la implementación C y Java son muy similares. En la implementación Java, el servidor crea el socket maestro y espera a recibir llamadas entrantes. Por cada llamada crea un nuevo `Thread` hijo pasando como parámetro el descriptor del socket esclavo en el cual espera el cliente del servicio. Del `Thread` hijo se muestra el constructor y el esqueleto del método `run()` que se deja como ejercicio al alumno.

6 Integración de servicios

A menudo en las RFCs se define que un servicio se puede dar en TCP y en UDP. También de forma bastante habitual se permite dar en un mismo puerto ambos servicios. Para optimizar los recursos, se suele utilizar un mismo servidor para dar ambos servicios y el servidor acepta tanto conexiones TCP como paquetes UDP al puerto en el que se ofrece el servicio. Otra ventaja es que el servicio suele utilizar el mismo código independientemente de si se da en TCP o en UDP, con lo cual si se varía unicamente la comunicación con sockets podemos tener un mismo programa para las dos modalidades. Esto es mucho más cómodo de mantener cuando se hacen nuevas revisiones y modificaciones del protocolo.
El último ejercicio que se plantea en este manual es hacer clientes de distintos servicios utilizando tanto TCP como UDP y un servidor único para todos los servicios y modalidades. El servidor escuchará en el puerto:

```
#define INTSERVPORT 2050
```

Los clientes solicitan al servidor un determinado servicio de entre los varios posibles. A continuación el servidor contesta con un número de puerto y máquina en la que será atendido el servicio en concreto. Los servicios existentes son los siguientes:

- servicio de eco
- servicio de autentificación
- servicio de localización de recursos
- servicios diversos

Los mensajes que mandan los clientes contienen únicamente un string con el nombre del servicio (`ECO`, `AUT`, `LOC`). La respuesta del servidor contiene dos enteros: uno corto indiando el puerto y uno largo indicando la dirección IP (generalmente la del propio servidor) en la que será atendida su petición.

6.1 Servicio de eco

El servicio de eco se incorpora a efectos de gestión del servidor y de depuración del programa. Para la descripción del servicio consultar la sección 5 de la página 18 de este manual.

6.2 Servicio de autentificación

El servicio de autentificación permite a un cliente obtener un identificador mediante el cual el proveedor del servicio garantiza la identidad de un usuario y da fe de ello a un tercero.
En este manual se va a plantear una de las estrategias que se pueden utilizar para dar un servicio de estas características. La estrategia que vamos a describir se basa en el uso de funciones de hash y concretamente en MD5. MD5 es un algoritmo que toma como entrada un mensaje de longitud arbitraria y que produce como salida una *huella digital* (o *message digest*) de 128 bits. Es computacionalmente imposible obtener dos mensajes que den como resultado la misma huella. El código fuente para el cálculo de MD5 que aparece en [4] está disponible en el laboratorio en el directorio

`ftp://itserv1.it.uc3m.es/pub/rc/md5.tgz`

En nuestro sistema cada uno de los usuarios del servicio tiene una frase secreta (`frase_usu`). Las frases además de secretas son únicas y conocidas por el servidor (se dan por teléfono u otros medios).
Cuando un usuario quiere un servicio de un servidor X, manda los siguientes parámetros de identificación a X: un string `nombre` (el nombre del usuario), un string identificador de la transacción en curso `num_op` (suministrado por X) y un tercer parámetro `md_usu`. `md_usu` se obtiene aplicando MD5 a la concatenación de

`nombre:num_op:frase_usu`

y es el *message digest* de la transacción realizada por el cliente.
X puede requerir del servidor de autentificación que certifique que `md_usu` es auténtica. X manda un mensaje de petición de autentificación indicando `nombre` y `num_op` a lo que el servidor responderá con el message digest del usuario `md_usu` si no ha habido ninguna anomalía.
El formato de los dos tipos de mensajes (petición y respuesta) del servicio de autentificación se indica en la tabla 5 de la página 26.

Primitiva	Parámetros
Petición	string - nombre usuario string - identificador transacción string - md usuario
Respuesta	string - md usuario

Tabla 5: Formato de mensajes del servicio de autentificación

6.3 Servicio de localización de recursos

El objeto del servicio de localización de recursos es el de disponer de agentes que gestionan las altas, bajas, consultas y modificaciones de los distintos agentes y los distintos servicios que ofrecen. A los agentes del servicio de localización de recursos los denominaremos *brokers*. Cuando un cliente necesita de un determinado servicio consulta a su broker los servidores que hay disponibles para ese servicio.

Por ejemplo: Juan quiere compilar un programa en C++ para una arquitectura motorola 68030. Juan selecciona un menú en su entorno gráfico, especifica los requisitos de la compilación y espera que el trabajo se realice de forma transparente. Para ello se invoca a un cliente que interroga al broker qué servidores hay disponibles para la compilación requerida. El broker consulta en sus tablas y da una respuesta mediante la cual el cliente puede invocar al agente que gestiona el acceso al compilador de C++ buscado.

Podemos imaginar muchos posibles escenarios: un nuevo agente de otro compilador cruzado de C++ se quiere dar de alta en el broker, o se quiere dar de baja, o quiere ampliar sus servicios a compilación cruzada de Fortran, etc.

El broker también debe de llevar la contabilidad de sus servicios, para lo cual tiene que identificar a sus clientes y asignar un identificador a cada transacción realizada.

Un broker puede a su vez requerir del servicio de otros brokers cuando se le pida un recurso y no tenga en sus tablas ningún servidor conocido.

Un agente puede dar de alta un servicio en un broker. Para ello necesitará comunicarle al broker varias cosas:

- Nombre del servicio (`string servicio`).

- Puerto del servicio (`string puerto`). El puerto puede ser o bien un string tal y como aparece en `/etc/services`, o un número codificado como un string que se recupera con `atoi` (o con el `método java.lang.String.parseInt`).

- Protocolo del servicio (`string proto`) TCP/UDP.

- Nombre del servidor (`string servidor`).

- Descripción general (`string descripcion`) en ASCII relativa al servicio ofrecido: Nombre, características técnicas del servicio, precio, disponibilidad, etc. Esta información está destinada a los posibles usuarios del servicio por lo que puede ser de utilidad mandarla codificada en HTML.

- *Message digest* del servidor (`string md`) construido como:

```
MD5(servicio:puerto:proto:servidor:descripcion:passwd)
```

donde `passwd` es la frase secreta del servidor conocida por el servidor de autentificación. El broker y cualquier cliente pueden solicitar del servidor de autentificación la verificación del message digest.
El broker devolverá un índice al servicio y un código de respuesta (ver tabla 6 de la página 28) al agente. El agente puede entonces actualizar o dar de baja el servicio indicando en su petición el índice devuelto por el broker.
Los clientes de usuarios finales piden al broker información de los servicios ofrecidos. Puesto que el broker tiene indexados todos los servicios, la forma lógica de pedir información sería a través de dichos índices. Por ejemplo: un broker tiene apuntados cien servicios. Un cliente puede pedir el listado básico de todos los servicios (índice y nombre del servicio), o el listado completo de los servicios 20 al 30, o el del servicio 23.[2]
En la tabla 7 de la página 29 se muestran todos los posibles mensajes de peticiones que pueden mandar los clientes y servidores finales a un broker. Todos los mensajes tienen un campo inicial de un byte en que se da el tipo de mensaje.

[2]Nota: Como extensión de la práctica, las tablas del broker se podrían consultar por servidores, nombre de servicio, etc.

Código	Significado
200	OK (siguen los datos)
201	Creado (sigue el identificador)
202	Baja o Modificación Aceptada
304	No Modificado
400	Petición Inválida
401	No autorizado
403	Prohibido
404	No encontrado
500	Error interno del servidor
501	No Implementado
503	Servicio No Disponible

Tabla 6: Códigos de respuesta del broker

Los desarrolladores en C deben notar que cuando un campo se indica que contiene un string, se supone la convención C para los strings, es decir, una cadena de carácteres que contenga el carácter `'\0'` como delimitador de final.
Los mensajes de **alta de un servicio** tienen un campo de longitud que es un entero largo y dice el número de bytes del campo de datos que se manda a continuación. Por último tienen un campo de message digest que, como se indicó anteriormente, se construye como:

```
MD5(servicio:puerto:proto:servidor:descripcion:passwd)
```

El campo de datos de un mensaje de alta se forma como se indica en la tabla 7 de la página 29, siendo el identificador del servicio el string nulo (`'\0'`). El último campo contiene la descripción del servicio y puede contener cualquier carácter (incluyendo `'\0'`). Su tamaño se determina a partir del campo de longitud y de los otros campos de datos.
El mensaje **modificación de un servicio** es muy similar al de altas, salvo que el identificador de servicio es el que devolvió el broker al dar de alta el servicio. El message digest se construye como:

```
MD5(ident:servicio:puerto:proto:servidor:descripcion:passwd)
```

Alta de un servicio:

Tipo (1 byte)	longitud long	datos char*	md string
0x0	No. de bytes de datos		

Modificación de un servicio:

Tipo (1 byte)	longitud long	datos char*	md string
0x3	No. de bytes de datos		

Baja de un servicio:

Tipo	Ident	md
0x7	string	string

Consulta de servicios:

Tipo	Ident 1	Ident 2
0x5	string desde	string hasta

Datos:

ident	servicio	puerto	proto	servidor	descripción
string	\0 si no hay cambios				

Tabla 7: Formato de mensajes del servicio de localización de recursos

El mensaje de baja de un servicio no tiene campo de datos. Está compuesto por el identificador del servicio y el message digest del servidor construido como:

```
MD5(ident:passwd)
```

Los mensajes de **consulta de servicios** están formados por dos identificadores que son dos strings. La codificación se da en la tabla 8 de la página 30.

Los mensajes de respuesta del broker se especifican en la tabla 9 de la página 31. Todos ellos contienen en primer lugar uno de los código de respuesta

`desde`	`hasta`	Listado requerido	Detalle
==\0	==\0 +	Total	Básico
==\0	p1	- p1	Básico
p0	==\0	p0 -	Básico
p0	p1	p0 - p1	Básico
p0	p0	p0	Completo

Tabla 8: Semántica de mensajes de consulta

indicados en la tabla 6 de la página 28. Los códigos 3xx, 4xx y 5xx son códigos de error y puede devolverlos el broker como respuesta a casi todas las peticiones, en caso de devolver un código de error se contempla la posibilidad de enviar un campo de longitud y unos datos auxiliares. El código de éxito 200 se reserva para responder a las consultas que tienen éxito. A continuación del código se manda la longitud de los datos y por último los datos. El código 201 se reserva para responder a las altas que tienen éxito, y el 202 para las modificaciones o bajas que tengan éxito, en cualquiera de estos tres casos (alta, modificación o baja) antes de aceptar el servicio hay que comprobar el digest con el servidor de autentificación. Una vez comprobado, a continuación del código se manda el identificador asignado al servicio. Si el digest no es válido, se manda el código 400.
En el caso de consultas no se considera la necesidad de enviar el digest del mensaje. Existen dos tipos de listados: básico y completo. Ambos casos se envían como un string (`char*`). El listado básico consta de secuencias –identificador de servicio (`ident`), nombre de servicio, puerto, protocolo y servidor– separadas por los carácteres `\r\n`. El listado completo incluye la descripción del servicio. Ambos tipos de datos están considerados en la figura 9 de la página 31.

6.4 Servicios diversos

Los otros servicios se dejan a la elección e imaginación del lector. Estos servicios deben de construirse a partir de software ya escrito y disponible en UNIX, o bien no deben de ser muy complicados. Los servicios pueden ser finales (como un agente buscador de precios mínimos) o intermediarios

Formatos de respuestas del broker:

código	datos		md
200	long	(char*) datos	**No hay**
201	string ident		MD5(codigo:ident:passwd)
202	string ident		MD5(codigo:ident:passwd)

Datos (listado básico):

ident	servicio	puerto	proto	servidor	sig.
string	string	string	string	string	`\r\n`

Datos (listado completo):

ident	servicio	puerto	proto	servidor	descripción
string	string	string	string	string	string

Tabla 9: Formato de mensajes de respuesta del broker

(como un agente anunciador de servicios).

7 Depuración de servidores

Para hacer la depuración de servidores, se recomienda el uso de un navegador de HTML como Netscape. Por ejemplo si hemos realizado una implementación de un servidor de `DAYTIME` que está corriendo en la máquina `tuba` en el puerto `13`, podemos monitorizar el funcionamiento del seridor pidiendo la dirección `http://tuba:13` al navegador. Como el resultado es texto ASCII, en la pantalla del navegador aparecerá la hora dada por el servidor.

8 Normas de la práctica

La práctica se realizará en grupos de libre elección. El número de alumnos por grupo será de un mínimo de tres y un máximo de cinco. En primer lugar se realizarán dos sesiones introductorias en las que se harán los ejem-

plos desarrollados en las primeras secciones de este manual y a continuación se realizará la implementación, prueba y documentación de los clientes y servidores descritos en la sección 6 de la página 24.
Los grupos deberán dividirse el trabajo de la implementación de los distintos clientes y servidores. Se recomienda que se realice la identificación de interfaces de forma conjunta para que todo el grupo participe de las decisiones iniciales del diseño. A continuación se sugiere especificar las interfaces y dividir el trabajo restante: implementación de los distintos módulos (u objetos si se opta por la implementación en Java), y diseño e implementación del plan de pruebas que debe de quedar reflejado en la práctica.
Las prácticas se corregirán en el laboratorio. Cada grupo expondrá el trabajo realizado y se realizarán preguntas a todos los componentes del grupo. Para la evaluación se tendrán en cuenta los siguientes criterios en el siguiente orden:

1. claridad del código: estilo, comentarios, etc.
2. corrección del código: funcionalidad correcta
3. plan de pruebas: identificación de casos de pruebas, implementación y entorno de ejecución de pruebas, cobertura de las pruebas.
4. eficiencia del código
5. extensión del código: servicios "extra" realizados (ver sección 6.4 de la página 30) y ampliaciones a las consultas de servicios (ver nota 2 de la página 27).
6. documentación (se tendrán en cuenta especialmente la brevedad y la claridad de la documentación adjuntada).
7. calificación (secreta) que cada miembro del grupo hace de la práctica y del resto de los componentes del grupo.

Los alumnos deberán entregar la práctica en la cuenta del grupo según el siguiente árbol de directorios:

```
practica/src <- ficheros fuente (incluir Makefile)
practica/bin <- binarios
practica/test <- casos de prueba
practica/doc <- documentacion (breve y clara)
```

Referencias

[1] Douglas E. Comer, David L. Stevens *Internetworking with TCP/IP Client-server programming and applications*. Vol. 3, Segunda edición, Prentice Hall 1996 `http://www.cs.purdue.edu/homes/comer/books.html`

[2] C. Partridge *IP Echo Host Service* IETF RFC número 2075, 01/08/1997 Standard

[3] J. Postel *Echo Protocol* IETF RFC número 862, 05/01/1983 Experimental

[4] Ronald L. Rivest *The MD5 Message-Digest Algorithm* IETF RFC número 1321, Abril 1992 Informational

```
public string TCPdaytime(String host)
{
  DatagramSocket s; /* socket */
  DatagramPacket d;   /* datagrama */
  byte buf[LINELEN+1];

  try{
   s = new DatagramSocket(13);
     /* 13 es el puerto de daytime */
   d = new DatagramPacket(buf,buf.length,
                          getByName(host),
                          "que hora es?");
   s.send(d);
   s.receive(d);
   String texto=new String(d.getData(),0,d.getLength());
   System.out.println(texto);
  }
  catch (UnknownHostException uh)
    { System.err.println("Host desconocido"); }
  catch (IOException io)
    { System.err.println("Error de I/O"); }
}
```

Figura 7: Cliente UDP de DAYTIME en Java

```
void TCPecho(const char *host)
{
  char buf[LINELEN+1]; /* buffer para una linea de texto */
  int s, n;            /* socket, contador de lectura */
  int outc, inpc;      /* caracteres enviados/recibidos */
  s = connectTCP(host, "echo");
  while(fgets(buf, sizeof(buf),stdin)){
    buf[LINELEN] = '\0';
    outc = strlen(buf);
    (void) write(s, buf, outc);

    for (inchars = 0; inpc < outc; inpc += n)
      if ((n = read(s, &buf[inpc], outc - inpc) < 0)
        error("lectura del socket no valida %s",
              strerror(errno));
    fputs(buf,stdout);
  }
}
```

Figura 8: Cliente TCP de ECHO en C

```
void UDPecho(const char *host)
{
  char buf[LINELEN+1]; /* buffer para una linea de texto */
  int s, n;            /* socket, contador de lectura */
  s = connectUDP(host, "echo");
  while(fgets(buf, sizeof(buf),stdin)){
    buf[LINELEN] = '\0';
    n = strlen(buf);
    (void) write(s, buf, n);

    if (read(s, buf, n) < 0)
        error("lectura del socket no valida %s",
              strerror(errno));
    fputs(buf,stdout);
  }
}
```

Figura 9: Cliente UDP de ECHO en C

```
public string TCPecho(String host)
{
  Socket s; /* socket */
  InputStream sin;
  OutputStream sout;

  try{
    s = new Socket(host, "echo");
    sin =s.getInputStream();
    sout =s.getOutputStream();

    byte buf[LINELEN+1];
    int outb, inpb;

    while(system.in.read(buf,sizeof(buf))){
      sout.write(buf);
      sin.read(buf);
    }
  }
  catch (UnknownHostException uh)
    { System.err.println("Host desconocido"); }
  catch (IOException io)
    { System.err.println("Error de I/O"); }
}
```

Figura 10: Cliente TCP de ECHO en Java

```
public string UDPecho(String host)
{
  DatagramSocket s; /* socket UDP */
  DatagramPacket din;   /* datagrama a recibir*/
  DatagramPacket dout;  /* datagrama a enviar*/
  byte buf[LINELEN+1];

  try{
    s = new DatagramSocket("echo");
    system.in.read(buf,buf.length);
    dout = new DatagramPacket(buf,buf.length
                              getByName(host),"echo");
    do{
      dout.setData(buf);
      s.send(dout);
      s.receive(din);
      system.out.println(din.getData());
    }while (system.in.read(buf,buf.length))
    }
  }
  catch (UnknownHostException uh)
    { System.err.println("Host desconocido"); }
  catch (IOException io)
    { System.err.println("Error de I/O"); }
}
```

Figura 11: Cliente UDP de ECHO en Java

```
int connectsock(const char *host, const char *service,
                const char *transport)
{
  struct hostent *phe;
  struct servent *pse;
  struct protoent *ppe;
  struct sockaddr_in sin;
  int s, type;

  memset(&sin, 0, sizeof(sin));
  sin.sin_family = AF_INET;

  /* Traduce nombre de servicio a numero de puerto */
  if (pse= getservbyname(service, transport))
    sin.sin_port = pse->s_port;
  else
    if ((sin.sin_port= htons((u_short)atoi(service)))==0)
      error(" servicio %s", service);

  /* Traduce nombre de host a direccion IP */
  if (phe = gethostbyname(host))
    memcpy(&sin.sin_addr, phe->h_addr, phe->h_length);
  else
    if ((sin.sin_addr.s_addr = inet_addr(host)) == INADDR_NONE)
    error(" host %s", host);

  /* Traduce protocolo de transporte a numero de protocolo */
  if ((ppe = getprotobyname(transport)) == 0)
    error(" protocolo %s", transport);

  /* Escoger el tipo de socket en funcion del protocolo */
  if (strcmp(transport, "udp") == 0)
    type = SOCK_DGRAM;
  else
    type = SOCK_STREAM;

   /* Asignar un socket */
   s= socket(PF_INET,type,ppe->p_proto);
   if (s < 0)
     error("socket:%s",strerror(errno));

   /* Conectar el socket */
   if (connect(s,(struct sockaddr*)&sin, sizeof(sin)) < 0)
     error("no conecto con %s.%s: %s",
            host,service,strerror(errno));
  return s;
}
```

Figura 12: `connectsock` en C

```
int
passivesock(const char *service, const char *transport, int qlen)
/*
 * Argumentos:
 *      servicio   - servicio asociado con el puerto deseado
 *      transporte - protocolo de transporte a usar (tcp o udp)
 *      qlen       - maxima longitud de la cola de peticiones
 *                   del servidor
 */
{
struct servent *pse; /* puntero a informacion de servicio */
struct protoent *ppe; /* puntero a informacion de protocolo */
struct sockaddr_in sin; /* direccion Internet */
int s, type; /* descriptor y tipo de socket*/

memset(&sin, 0, sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;

    /* Traduce nombre de servicio a numero de puerto */
if ( pse = getservbyname(service, transport) )
sin.sin_port = htons(ntohs((u_short)pse->s_port)
+ portbase);
else if ( (sin.sin_port = htons((u_short)atoi(service))) == 0 )
errexit("no encuentro servicio \"{}%s\"{}\n", service);

    /* Traduce nombre de protocolo a numero de protocolo */
if ( (ppe = getprotobyname(transport)) == 0)
errexit("no encuentro protocolo \"{}%s\"{}\n", transport);

    /* Usa protocolo para elegir el tipo de socket */
if (strcmp(transport, "udp") == 0)
type = SOCK_DGRAM;
else
type = SOCK_STREAM;

    /* Asigna un socket */
s = socket(PF_INET, type, ppe->p_proto);
if (s < 0)
errexit("can't create socket: %s\n", strerror(errno));

    /* Bind */
if (bind(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
errexit("no puedo hacer bind al servicio %s puerto:%s\n",
                        service, strerror(errno));
if (type == SOCK_STREAM && listen(s, qlen) < 0)
errexit("no puedo escuchar el servicio %s puerto: %s\n",
                        service,strerror(errno));
return s;
}
```

Figura 13: passivesock() en C