Manual de sockets
Redes de Ordenadores
Ingeniero de Telecomunicación

Andrés Marín López

Leganés, Abril 2002

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.

El presente manual se ha inspirado en un libro clásico [2], 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 para 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.

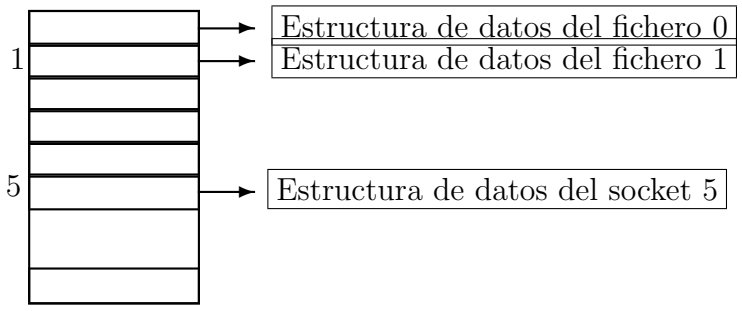

Figure 1: Representación de la tabla de descriptores de un proceso

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

Table 1: Parámetros de los sockets BSD

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. 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) */
}
```

Table 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 15 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 así 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, así como la longitud del mismo. La operación de lectura copia en el buffer del usuario los datos recibidos. Si no ha llegado ningún 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 envíos 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

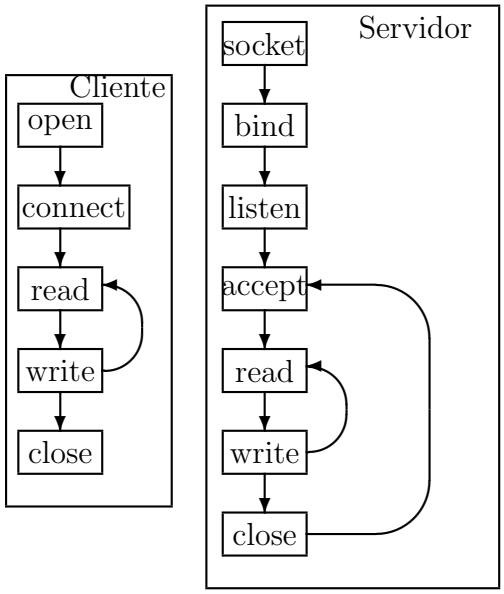

Figure 2: Uso de sockets en un programa

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 10.

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.

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

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

Table 3: Formatos de representación de enteros

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 *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 12. **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**.

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.

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

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

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;
}
```

Figure 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 [2] y la adjuntamos en la figura 4 de la página 16.

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. `connectsock()` es la función escrita por Comer y Stevens para crear un socket y conectarlo.

En la figura 5 de la página 17 se muestra una implementación en Java de un cliente TCP de `DAYTIME`. El interfaz de sockets en Java es mucho más

[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");
}
```

Figure 4: Cliente TCP de `DAYTIME` en C

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 17 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 forma 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 `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 paquete `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 posibili-

```
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"); }
}
```

Figure 5: Cliente TCP de `DAYTIME` en Java

dad 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");
}
```

Figure 6: Cliente UDP de DAYTIME en C

La figura 6 de la página 18 contiene el código dado por [2] 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 19 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 17 y respecto al código C de la figura 6 de la página 18.

```
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"); }
}
```

Figure 7: Cliente UDP de `DAYTIME` en Java

5 Clientes y servidores de eco

```
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);
  }
}
```

Figure 8: Cliente TCP de `ECHO` en C

Los estándares TCP/IP definen un servicio de eco (`ECHO`) en [3] y [4] 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.

```
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);
  }
}
```

Figure 9: Cliente UDP de `ECHO` en C

Las figuras 8 de la página 20 y 9 de la página 21 muestran las implementaciones de [2] en C de clientes TCP y UDP de `ECHO`. Las figuras 10 de la página 28 y 11 de la página 29 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 inician 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 [2] de un servidor TCP de `echo`. El servidor usa procesos concurrentes para dar servicio concurrente a múltiples clientes. El servidor usa la función `passiveTCP()` 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 utiliza el descriptor al socket esclavo que devolvió `accept` al servidor para llamar a llama a `TCPechod()`. 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 código 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.

References

[1] W. Richard Stevens. TCP/IP Illustrated, Vol I The Protocols. Addison Wesley Professional Computing Series, 1994.

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

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

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

```
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"); }
}
```

Figure 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"); }
}
```

Figure 11: Cliente UDP de `ECHO` en Java