Introducción a Sistemas Operativos: La red

Clips xxx
Francisco J Ballesteros

1. Sockets

La red no existía cuando hicieron UNIX. Cuando posteriormente las máquinas empezaron a interconectarse (antes de la llegada de Skynet) nuevas abstracciones aparecieron para poder usar la red en UNIX. Igual sucedió con los terminales gráficos.

El desmadre que puede apreciarse en los interfaces (y en gran parte de las implementaciones) para usar tanto la red como los gráficos en UNIX se debe en parte a que ni Ken Thompson ni Dennis Ritchie estaban ahí cuando los añadieron a UNIX. Mientras tanto, en Bell Labs añadieron interfaces mas razonables para ambas cosas (primero en su versión de UNIX y luego en Plan), pero por desgracia las versiones de los otros se extendieron antes y tenemos que vivir con ello.

La principal abstracción para utilizar la red se conoce como **socket**. Es como un descriptor de fichero (por un extremo) y el otro extremo es la red, con lo que para nosotros un socket es como un calcetín (un tubo con un sólo extemo). De ahí el nombre.

Una vez tenemos creado y configurado un socket podemos utilizarlo para leer y escribir. Siendo un socket, cuando leemos recibimos datos de la red y cuando escribimos enviamos datos a la red. Si utilizamos un protocolo orientado a conexión entonces puede que varios writes se envíen como un sólo mensaje de red, o puede que parte de un write. Si utilizamos un protocolo orientado a datagramas entonces cada write envía un datagrama. Esta es la idea. El interfaz es algo más complicado, aunque no tanto.

Los sockets pueden utilizarse tanto para TCP, como para UDP, como para otros protocolos que no son TCP/IP (que no son de internet). De hecho, existen los llamados *unix domain sockets* que funcionan sólo dentro de una máquina. En la actualidad lo mas práctico es utilizar siempre TCP/IP en lugar de cualquier otro tipo de sockets.

2. Un cliente

Veamos cómo programar un **cliente**, llamado así puesto que hace peticiones a un **servidor**, llamado así puesto que sirve las peticiones de un cliente. Lo que debemos hacer es:

- Crear un socket, en este caso para usar TCP, luego
- construir la dirección a que queremos llamar, y por último
- establecer la conexión a dicha dirección.

Una vez tengamos esto hecho, obtendremos un descriptor de fichero conectado al servidor y podremos usar `read` y `write` para hablar con el otro extremo de la red. Casi como si fuese un pipe full-duplex (bidireccional).

Vamos a implementar una función llamada `dial` a la que daremos un string con la dirección del servidor y un entero con el puerto en el servidor al que queremos conectarnos. Aunque ya has estudiado redes de ordenadores, recordamos que los puertos sirven para hablar con procesos dentro de una máquina y son enteros que no difieren mucho de un apartado de correos.

```
int
dial(char *host, int port)
{
    struct hostent *hent;
    int sfd;
    struct sockaddr_in addr;



    hent = gethostbyname(host);
    if (hent == NULL) {
        return -1;
    }
    memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_port = htons(port);
    memmove(&addr.sin_addr.s_addr, hent->h_addr, hent->h_length);



    sfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sfd < 0) {
        return -1;
    }
    fprintf(stderr, "dialing %s:%d...\n", host, port);
    if (connect(sfd, (struct sockaddr*)&addr, sizeof(addr)) < 0) {
        close(sfd);
        return -1;
    }
    return sfd;
}
```

Todo el código al principio de la función llama a `gethostbyname` para obtener una dirección IP para el nombre del host al que queremos conectarnos y construye una dirección en la variable `addr`. Ten en cuenta que los sockets usan redes dispares y las direcciones deben soportar las distintas redes. En Plan 9 utilizan strings como tipo de datos, lo que resulta más natural, pero en UNIX es preciso hacer conversiones de tipo y mover los bytes casi a mano. Cuidado así mismo con poner los bytes en el formato de red, como recordarás de tus estudios de redes de ordenadores.

El socket está dentro del kernel. En esta función `sfd` es el descriptor del socket y eso es todo lo que necesitamos. Lo creamos llamando a

```
sfd = socket(AF_INET, SOCK_STREAM, 0);
```

para crear un socket de TCP/IP (`AF_INET`) que hable TCP (`SOCK_STREAM`).

Una vez creado podemos llamar a `connect` para conectarnos a la dirección que pasamos como argumento. En ese momento, nuestro socket abstrae toda la pila de TCP/IP que tiene debajo para hablar con el servidor en el otro extremo de la red.

Con este código, es posible escribir un cliente para, por ejemplo, conectarse a un servidor, escribir una línea de comandos y leer la salida de la ejecución de dicho comando en el servidor.

[cli.c]:

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <netdb.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <err.h>

int
dial(char *host, int port)
{
    ...como se muestra antes...
}


int
main(int argc, char *argv[])
{
    int fd, port, nr;
    char buf[16*1024];

    if (argc != 4) {
        fprintf(stderr, "usage: %s addr port cmd\n", argv[0]);
        exit(1);
    }
    port = atoi(argv[2]);


    fd = dial(argv[1], port);
    if (fd < 0) {
        err(1, "dial %s:%d", argv[1], port);
    }
    snprintf(buf, sizeof buf, "%s\n", argv[3]);
    buf[sizeof buf - 1] = 0;
    if (write(fd, buf, strlen(buf)) != strlen(buf)) {
        close(fd);
        err(1, "write");
    }
```

```
    for(;;) {
        nr = read(fd, buf, sizeof buf-1);
        if (nr < 0) {
            close(fd);
            err(1, "read");
        }
        if (nr == 0) {
            break;
        }
        if (write(1, buf, nr) != nr) {
            close(fd);
            err(1, "write stdout");
        }
    }
    close(fd);
}
```

3. Un servidor

Para programar un servidor necesitamos poder atender llamadas procedentes de clientes, lo que podemos hacer con un socket. Este socket es preciso configurarlo con la dirección en que queremos escuchar (que incluye el puerto en que escuchamos). Dicho socket se queda escuchando hasta que un cliente se conecta. Además, cada llamada que se acepta se procesa en otro socket distinto para permitir que el socket original continúe escuchando. La siguiente función devuelve un socket que está escuchando en el puerto de TCP que se indica (en cualquier dirección que posea la máquina para dicho protocolo).

```
int
listentcp(int port)
{
    int sfd;
    struct sockaddr_in addr;

    memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_port = htons(port);

    sfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sfd < 0) {
        return -1;
    }
    if (bind(sfd, (struct sockaddr*)&addr, sizeof(addr)) < 0) {
        close(sfd);
        return -1;
    }
    if (listen(sfd, 30) < 0) {
        close(sfd);
        return -1;
    }
    return sfd;
}
```

Al principio construimos una dirección para configurar el socket, que guarda el puerto en que queremos escuchar. Usamos una dirección IP que indica que nos vale cualquier dirección IP, lo que en realidad quiere

decir que queremos escuchar en todas las que tenga el sistema.

Posteriormente creamos el socket como en el código del cliente, pero, esta vez hemos de llamar a `bind` para darle una dirección al socket. En este punto podemos llamar a `listen` para ponerlo realmente a escuchar. Tras la llamada, UNIX acepta peticiones de conexión (realmente las recibe sin rechazarlas) y mantiene un máximo de 30 en cola (según indica el segundo argumento) mientras nuestro programa se toma su tiempo para llamar a `accept`.

Con esta llamada, tenemos el socket escuchando pero no estamos realmente aceptando peticiones de conexión. Esto es, estamos escuchando pero no *descolgando* el teléfono. La siguiente función "descuelga" cuando llega una llamada pero aceptándola en otra "extensión telefónica" (en otro socket) para que nuestro socket pueda seguir escuchando llamadas.

```
int
acceptcall(int listenfd, char caddr[], int ncaddr)
{
    int fd, port;
    struct sockaddr_in addr;
    socklen_t alen;
    char *cs;


    memset(&addr, 0, sizeof(addr));
    alen = sizeof(addr);
    fd = accept(listenfd, (struct sockaddr*)&addr, &alen);
    if (fd < 0) {
        return -1;
    }
    cs = inet_ntoa(addr.sin_addr);
    port = ntohs(addr.sin_port);
    snprintf(caddr, ncaddr, "%s#%d", cs, port);
    caddr[ncaddr-1] = 0;
    return fd;
}
```

La llamada a `accept` devuelve *otro* socket, cuyo descriptor es *fd*, tras esperar una llamada en el socket `listenfd` que procede del *listen*, y aceptarla. Además, dicha llamada deja en `addr` la dirección del cliente que se ha conectado, lo que aprovechamos para hacer que la función deje en `caddr[]` un string con una cadena que corresponda a dicha dirección.

Ya tenemos todo lo necesario para programar el servidor. Ahora basta ponerlo a escuchar y, para cada llamada, ejecutar un shell que ejecute el comando y escriba la respuesta desde el servidor.

[xsrv.c]:

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <netdb.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>
#include <err.h>

int
listentcp(int port)
{
    ...como se muestra antes...
}
int
acceptcall(int listenfd, char caddr[], int ncaddr)
{
    ...como se muestra antes...
}

int
main(int argc, char *argv[])
{
    int lfd, cfd, infd, port, nr;
    char buf[16*1024];

    if (argc != 2) {
        fprintf(stderr, "usage: %s port\n", argv[0]);
        exit(1);
    }
    port = atoi(argv[1]);
    lfd = listentcp(port);
    if (lfd < 0) {
        err(1, "listen *:%d", port);
    }


    for(;;) {
        cfd = acceptcall(lfd, buf, sizeof buf);
        fprintf(stderr, "call from %s\n", buf);
        switch(fork()) {
        case -1:
            close(cfd);
            close(lfd);
            err(1, "fork");
            break;
```

```
        case 0:
            close(lfd);
            // should really read line by line
            nr = read(cfd, buf, sizeof buf-2);
            if (nr < 0) {
                close(cfd);
                err(1, "read");
            }
            if (nr == 0) {
                close(cfd);
            }
            buf[nr++] = '\n';
            buf[nr] = 0;
            fprintf(stderr, "exec %s", buf);
            infd = open("/dev/null", O_RDONLY);
            if (infd < 0) {
                close(cfd);
                err(1, "open");
            }
            dup2(infd, 0);
            close(infd);
            dup2(cfd, 1);
            dup2(cfd, 2);
            close(cfd);
            execl("/bin/sh", "sh", "-c", buf, NULL);
            exit(1);


        default:
            close(cfd);
        }
    }
    exit(1);
}
```

4. Usando el cliente y el servidor

Podemos ejecutar el servidor en un shell

```
unix$ xsrv 4000
```

Y llamarlo con un cliente ejecutado en otro

```
unix$ cli localhost 4000 ls
dialing localhost:4000...
cli
xsrv
cli.c
cli.o
xsrv.c
unix$ cli localhost 4000 date
dialing localhost:4000...
Wed Aug 31 19:40:05 CEST 2016
unix$
```

Esto puede verse en la ventana del servidor mientras tanto...

```
unix$ xsrv 4000
call from 127.0.0.1#62961
exec ls

call from 127.0.0.1#62962
exec date
```

Con lo que hemos visto y el manual tienes todo lo necesario para programar servidores y clientes. Las herramientas que hemos aprendido a utilizar durante el curso, como verás, han resultado útiles para hacer nuestro trabajo.

Además, en el futuro, basta con utilizar `dial`, `listentcp` y `acceptcall` en lugar de tener que utilizar directamente el interfaz de sockets. Acóstumbrate a implementar abstracciones como la que proporcionan estas tres funciones: *los sockets como llamadas de teléfono*. Te resultará cómodo para evitar repetir el trabajo que tienes que hacer una y otra vez.