Introducción al uso de SOCKETS en Linux

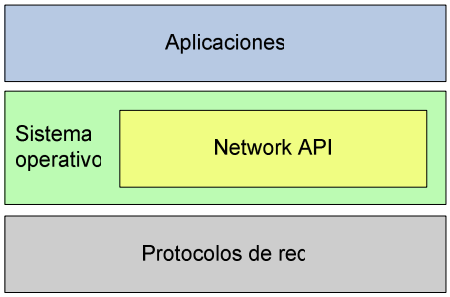
Lic. Leonardo de - Matteis

Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur

Interfaz de programación de aplicaciones

NAPI (Network Application Programming Interface)

Servicios provistos usualmente por el sistema operativo, los cuales brindan una interface entre componentes de software en la capa de aplicación y protocolos de red específicos.



2

APIs para comunicación vía TCP/IP

NAPI (Network Application Programming Interface)

- Los protocolos de transporte TCP/UDP no incluyen la definición de una API específica.

- Existen diversas APIs que se pueden utilizar con TCP/IP:
  - ✓ Sockets (BSD: Berkeley sockets)
  - TLI, XTI (AT&T UNIX System V)
  - Winsock (Microsoft, pero basado en BSD sockets)
  - MacTCP (Mac OS, Apple Computer)
  - Open Transport (Mac OS, Apple Computer)

3

Socket: definición

Representación abstracta de un punto de comunicación, que permite establecer un canal de comunicación entre dos rutinas o programas.

- Los podemos considerar como archivos, que se crean de manera especial.
- Los *sockets* trabajan con los servicios de entrada/salida de sistemas operativos del tipo Unix.
- Diversos programas corriendo en computadoras diferentes dentro de una red, pueden comunicarse a través del uso de *sockets*.
- Con las funciones write() y read() del lenguaje C, se pueden escribir y leer datos en el *socket*.

Sockets (BSD: Berkeley sockets)

- Primera versión disponible en el Unix BSD 4.2 (1983).
- API basada en librerías, para escribir programas en C.
- Permiten escribir rutinas para la intercomunicación entre procesos.
- Brindan generalidad:
  - soportan multiples familias de protocolos;
  - independencia en la representación de las direcciones.
- Utiliza la interface de programación de entrada/salida existente lo más posible.
- API standard para la utilización de *sockets* en redes.
- Otros lenguajes utilizan una API similar.

5

Tipos de sockets

Existen dos tipos de "canales de comunicación" o *sockets*, los **orientados a conexión** y los **no orientados a conexión**.

Orientados a conexión (TCP):

Dos programas deben conectarse entre ellos con un *socket* y hasta que no esté establecida correctamente la conexión, ninguno de los dos puede transmitir datos. (Protocolo TCP sobre IP).

Esto garantiza que todos los datos van a llegar de un programa al otro correctamente. Se utiliza cuando la información a transmitir es importante, y no se debe perder ningún dato.

Si uno de los programas está "ocupado" y no atiende la comunicación, el otro quedará bloqueado hasta que el primero lea o escriba los datos en el canal.

6

Tipos de sockets

Sin conexión (UDP):

No es necesario que los programas se conecten. Cualquiera de ellos puede transmitir datos en cualquier momento, independientemente de que el otro programa esté "escuchando" o no.

Para esto se utiliza el protocolo UDP sobre IP, y garantiza que los datos que lleguen sean correctos, pero **no garantiza que lleguen todos**.

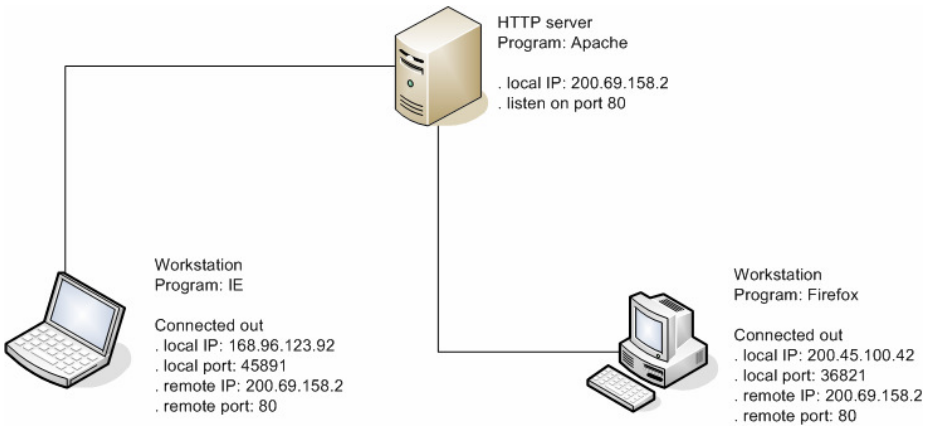
Programas: servidor y cliente

Servidor: es el programa que permanece pasivo a la espera de que alguien solicite conexión. Puede o no devolver datos.

Cliente: es el programa que solicita la conexión para enviar o solicitar datos al servidor.

7

Arquitectura cliente/servidor



8

Construcción del server

1) Creación del *socket*:

Función: **socket**()

Es una llamada al sistema, esta función devuelve un descriptor de archivo, al igual que la función **open**() al crear o abrir un archivo .

Durante la llamada se reservan los recursos necesarios para el punto de comunicación, pero no se especifica nada con respecto a la dirección asociada al mismo.

Librerías requeridas y parámetros en Linux:

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

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

DESCRIPTION
       Socket creates an endpoint for communication and returns a
       descriptor.
```

9

Construcción del server

1) Creación del *socket*:

Prototipo: int **socket** (int ***family***, int ***type***, int ***protocol***);

- **family:** especifica la familia del protocolo (PF_INET para TCP/IP).
- **type:** especifica el tipo de servicio (SOCK_STREAM, SOCK_DGRAM).
- **protocol:** especifica el protocolo (usualmente 0, lo cual significa el protocolo *por defecto* para la familia elegida).

Resultado:

La llamada al sistema **socket**() devuelve un descriptor (número de tipo *small integer*) o -1 si se produjo un error.

El descriptor de archivo luego será usado para asociarlo a una conexión de red.

10

Construcción del server

2) Asociar una dirección al *socket*:

función **bind**()

Llamada al sistema que permite asignar una dirección a un *socket* existente.

El sistema no atenderá a las conexiones de clientes, simplemente registra que cuando empiece a recibirlas le avisará a la aplicación.

En esta llamada se debe *indicar* el *número de servicio* sobre el que se quiere atender.

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

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

DESCRIPTION
       bind gives the socket sockfd the  local  address  my_addr.
       my_addr  is  addrlen  bytes  long.  Traditionally, this is
       called "assigning a name to a socket."  When a  socket  is
       created with socket(2), it exists in a name space (address
       family) but has no name assigned.
```

11

Construcción del server

2) Asociar una dirección al *socket*:

Prototipo: int **bind** (int ***sockfd***, const struct sockaddr ***myaddr***, int ***addrlen***);

- ***myaddr**: asigna la dirección especificada en la estructura del tipo **sockaddr**. En este parámetro suele utilizarse una estructura del tipo **sockaddr_in** (ejemplo en transparencia 20).
- **sockfd:** descriptor del *socket* involucrado.
- **addrlen:** tamaño de la estructura en ***myaddr**, es decir sizeof(myaddr).

Resultado:

La llamada al sistema **bind**() devuelve un 0, si se produjo un error devuelve -1.

12

Construcción del server

3) Atender conexiones:

función **listen**()

Avisar al sistema operativo que comience a atender la conexión de red.
El sistema registrará la conexión de cualquier cliente para transferirla a la aplicación cuando lo solicite.

Si llegan conexiones de clientes más rápido de lo que la aplicación es capaz de atender, el sistema almacena en una cola las mismas y se podrán ir obteniendo luego.

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

       int listen(int s, int backlog);

DESCRIPTION
       To accept connections, a socket is first created with
       socket(2), a willingness to accept incoming connections
       and a queue limit for incoming connections are specified
       with listen, and then the connections are accepted with
       accept(2). The listen call applies only to sockets of
       type SOCK_STREAM or SOCK_SEQPACKET.
```

13

Construcción del server

4) Aceptar conexiones:

función **accept**()

Pedir y aceptar las conexiones de clientes al sistema operativo. El sistema operativo entregará el siguiente cliente de la cola. Si no hay clientes se quedará bloqueada hasta que algún cliente se conecte.

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

       int accept(int s, struct sockaddr *addr, socklen_t
       *addrlen);

DESCRIPTION
       The accept function is used with connection-based socket
       types (SOCK_STREAM, SOCK_SEQPACKET and SOCK_RDM). It
       extracts the first connection request on the queue of
       pending connections, creates a new connected socket with
       mostly the same properties as s, and allocates a new file
       descriptor for the socket, which is returned. The newly
       created socket is no longer in the listening state. The
       original socket s is unaffected by this call. Note that
       any per file descriptor flags (everything that can be set
       with the F_SETFL fcntl, like non blocking or async state)
       are not inherited across an accept.
```

14

Construcción del server

5) Leer datos desde una conexión:

función **read**()

Recibir datos a través del descriptor del *socket*.

El cliente y el servidor deben conocer que datos esperan recibir, y cuales deben enviar, bajo un formato determinado.

```
SYNOPSIS
       #include <unistd.h>

       ssize_t read(int fd, void *buf, size_t count);

DESCRIPTION
       read()  attempts  to  read  up  to  count  bytes from file
       descriptor fd into the buffer starting at buf.

       If count is zero, read() returns zero  and  has  no  other
       results.   If  count is greater than SSIZE_MAX, the result
       is unspecified.
```

15

Construcción del server

6) Escribir datos en la conexión:

función **write**()

Escribir datos a través del descriptor del *socket*.

```
SYNOPSIS
       #include <unistd.h>

       ssize_t write(int fd, const void *buf, size_t count);

DESCRIPTION
       write  writes  up to count bytes to the file referenced by
       the file descriptor fd from the buffer  starting  at  buf.
       POSIX  requires that a read() which can be proved to occur
       after a write() has returned returns the new  data.   Note
       that not all file systems are POSIX conforming.
```

7) Cerrar la conexión:

función **close**()

Cierra el descriptor del *socket*.

16

Construcción del cliente

1) Creación del *socket*

función **socket**()

Esta función devuelve un descriptor de archivo, al igual que la función **open**() al crear o abrir un archivo .

2) Conectar con el servidor:

función **connect**()

Iniciar/solicitar una conexión con el servido. Dicha función quedará bloqueada hasta que el servidor acepte nuestra conexión.

Si no hay servidor que atienda, se obtendrá un código de error (-1).

Se debe especificar la dirección IP del servidor y el número de puerto (servicio) al que se desea conectar.

17

Construcción del cliente

3) Conectar con el servidor:

función **connect**()

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

       int  connect(int sockfd, const struct sockaddr *serv_addr,
       socklen_t addrlen);

DESCRIPTION
       The file descriptor sockfd must refer to a socket.  If the
       socket is of type SOCK_DGRAM then the serv_addr address is
       the address to which datagrams are sent  by  default,  and
       the  only  address  from which datagrams are received.  If
       the socket is of type SOCK_STREAM or SOCK_SEQPACKET,  this
       call attempts to make a connection to another socket.  The
       other socket  is  specified  by  serv_addr,  which  is  an
       address (of length addrlen) in the communications space of
       the socket.   Each  communications  space  interprets  the
       serv_addr parameter in its own way.
```

18

Construcción del cliente

5) Escribir datos en la conexión:

función **write**()

Escribir datos a través del descriptor del *socket*.

6) Leer datos desde una conexión:

función **read**()

Recibir datos a través del descriptor del *socket*.

El cliente y el servidor deben conocer que datos esperan recibir, y cuales deben enviar, bajo un formato determinado.

7) Cerrar la conexión:

función **close**()

Cierra el descriptor del *socket*.

19

Sinopsis

Estructuras y variables para el servidor:

```
extern  int errno;
extern  char *sys_errlist[];

int     sockfd;
struct sockaddr_in server;


/* Estructura con datos del server */

bzero(&server, sizeof(server));
server.sin_family      = AF_INET;
server.sin_addr.s_addr = INADDR_ANY;
server.sin_port        = htons((short) portnum);

//server.sin_addr = htonl(ipaddress); /* dirección IP específica */
```

20

Sinopsis

Algoritmo básico para el servidor:

```
/* Obtener un descriptor de archivo para el socket */
if ((sockfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
    perror ("error al querer abrir el socket\n");
    exit(1);
}
/* Asociar el programa al socket */
if (bind(sockfd, (struct sockaddr*)&server, sizeof(server))) {
    perror("no se pudo hacer el bind");
    exit(1);
}
/* Aceptación de conexiones en modo pasivo */
if (listen(sockfd, 5) < 0) {
    printf("No se puede oir en el port %d : %s\n", portnum,
    sys_errlist[errno]);
    exit(1);
}
    .
    .

close(sockfd);
```

21

Sinopsis

Estructuras y variables para el cliente:

```
int sockfd;
struct sockaddr_in server;
struct hostent *server_host;


/* Estructura con datos del server */

bzero(&server, sizeof(server));
server.sin_family      = AF_INET;
bcopy(server_host -> h_addr,
      &server.sin_addr,
      server_host -> h_length);
server.sin_port = htons((short) SERVER_PORT);
```

22

Sinopsis

Algoritmo básico para el cliente:

```
/* Obtener un descriptor de archivo para el socket */
if ((sockfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
    perror ("error al querer abrir el socket\n");
    exit(1);
}

/* connect to server, which should already be setup */
if (connect(sockfd, (struct sockaddr*) &server, sizeof(server)) < 0) {
    perror("error al querer establecer la conexión con el server");
    exit(1);
}

/ * obtener datos del host a través de su nombre */
if ((server_host = gethostbyname(SERVER_NAME)) == 0) {
    perror("unknown server name");
    exit(1);
}
    .
    .

close(sockfd);
```

23

Sinopsis

Generalidad:

```
/ * obtener datos del host a través de su nombre */
if ((server_host = gethostbyname(SERVER_NAME)) == 0) {
    perror("unknown server name");
    exit(1);
}

/* obtener datos del servicio a través de su nombre */
getservbyname()
```

Archivos de configuración involucrados:

```
/etc/protocols
/etc/hosts
/etc/services
```

24

Aclaraciones

Según la arquitectura subyacente los valores se almacenarán de diferentes maneras, es decir el ordenamiento de los bytes será distinto.

Cuando se utiliza el protocolo IP (Internet Protocol) el mismo define como estandard para la trasmisión de datos el formato: big-endian.

Los equipos basados en procesadores del tipo x86 (Intel) utilizan el formato: little-endian.

Si deseamos establecer una canal de comunicación entre dos programas en diferentes equipos utilizando *sockets*, la información transmitida debe ajustarse para ser recepcionada en forma correcta.

Para evitar problemas se utilizan las funciones de conversión. Dichas funciones permiten convertir los valores a transmitir.

Es decir: si enviamos enteros con write() y luego en otro punto de comunicación los leemos con read(), debemos convertirlos al formato adecuado.

25

Aclaraciones: Network byte order

Todos los valores (bits) almacenados en la estructura: **sockaddr_in** deben estar ordenados según alguna de las siguientes alrternativas:

- network byte order
- host byte order

Los siguientes campos involucran el uso de funciones para el ordenamiento de los bits de los valores almacenados:

- `sin_port`: número de puerto según protocolo UDP/TCP sobre IP.
- `sin_addr`: dirección IP.

26

Aclaraciones: Funciones de conversión

Las funciones: htonl, htons, ntohl y ntohs, permiten realizar conversiones para el ordenamiento de los bits según la red o el equipo.

```
h : host byte order
n : network byte order
s : short (16bit)
l : long (32bit)
```

Prototipos:

```
uint16_t htons(uint16_t hostshort);
uint16_t ntohs(uint16_t netshort);

uint32_t htonl(uint32_t hostlong);
uint32_t ntohl(uint32_t netlong);
```

27

Librerías necesarias

```
/usr/include/stdio.h
/usr/include/netdb.h
/usr/include/sys/errno.h
/usr/include/sys/types.h
/usr/include/sys/socket.h
/usr/include/netinet/in.h
```

Bibliografía

- *UNIX Systems Programming: Communication, Concurrency and Threads,* Kay A. Robbins, Steve Robbins - Sams, 2001
- *Linux Socket Programming* (1st edition), Sean Walton, Prentice Hall, 2003.

28