

Unix: Programación en Redes

Programación en Redes

- Para rellenar la estructura:

```
struct sockaddr_in sa;
struct hostent *hp;
char myname[MAXHOSTNAME+1];

bzero(&sa, sizeof(struct sockaddr_in));
gethostname(myname, MAXHOSTNAME);
hp = gethostbyname(myname);
if (hp == NULL)
    return -1;
sa.sin_family = AF_INET;
sa.sin_port = htons(portnum);
bcopy(hp->h_addr_list[0], &sa.sin_addr.s_addr,
      hp->h_length);
```

Programación en Redes

- La estructura hostent:

```
struct hostent{
    char *h_name;      /* official name of host */
    char **h_aliases;  /* alias list */
    int h_addrtype;    /* host address type */
    int h_length;      /* length of address */
    char **h_addr_list; /* list of addresses */
}
```

- “h_addrtype” es generalmente AF_INET

Programación en Redes

- Para obtener información de la red en donde se está ejecutando, se utilizan las funciones:

```
struct hostent *gethostbyaddr(char *addr,
                              int len, int type);
struct hostent *gethostbyname(char *name);
```

- Para “gethostbyaddr” el único tipo válido es AF_INET
- “gethostbyname”, buscará la dirección IP asociada al nombre, llamando las funciones necesarias para “resolver” el nombre

Programación en Redes

- Para “resolver” nombre:
 - Se consultarán archivos de configuración (/etc/hosts)
 - Información de los servicios de red (NIS)
 - DNS
- Similarmente, se puede obtener información de los servicios con las funciones:

```
struct servent *getservbyname(char *name,
                              char *proto);
struct servent *getservbyport(int port,
                              char *proto);
```

Programación en Redes

- Para las funciones anteriores, el parámetro “proto”, será:
 - “tcp”, para conexiones SOCK_STREAM TCP
 - “udp”, para SOCK_DGRAM datagramas
- La estructura “servent”:

```
struct servent {
    char    *s_name;           /* nombre oficial del
                               servicio */
    char    **s_aliases;       /* lista de alias */
    int     s_port;            /* número de puerto */
    char    *s_proto;          /* protocolo a usar */
}
```

Programación en Redes

- Para obtener el nombre del host actual existe la función:

```
int gethostname(char *name, size_t len);
```

- Escribe el nombre del host actual en “name”, con largo máximo “len”
- Retorna 0 en caso de éxito y -1 en caso de error

Programación en Redes

- Una vez que se tiene creado el socket y asociado a una dirección (bind) es necesario recibir conexiones:
 - Para aceptar conexiones entrantes en un socket, el programa servidor debe crear una cola para mantener estos requerimientos:

```
int listen(int s, int backlog);
```

- Comienza a esperar conexiones, hasta un límite de “backlog” conexiones pendientes
- Las conexiones que sobrepasen este límite, serán rechazadas

Programación en Redes

- El mecanismo provisto por “listen” permite mantener conexiones entrantes pendientes mientras el programa servidor está ocupado
- Para aceptar una conexión, se utiliza la función:

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

- Saca la primera solicitud pendiente de conexión para el socket s
- Crea un nuevo socket para comunicarse con el cliente y lo retorna

Programación en Redes

- El nuevo socket tendrá el mismo tipo que el socket servidor que está haciendo listen
- El socket “s” previamente debe tener una dirección asociada con “bind” y una cola de conexiones con “listen”
- La dirección del cliente que está llamando será almacenada en la estructura apuntada por “addr”
- Si no hay conexiones pendientes, la llamada a “accept” **bloqueará** la ejecución hasta que llegue un nuevo cliente
- Se puede utilizar el flag O_NONBLOCK para evitar el bloqueo

Programación en Redes

- Para que el programa no se bloquee:

```
int flags = fcntl(socket, F_GETFL, 0);
fcntl(socket, F_SETFL, O_NONBLOCK|flags);
```

- con F_GETFL, se obtienen los flags del socket
- con F_SETFL, se asignan todos los flags del socket más el de no bloqueo

- Para realizar conexiones se utiliza:

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

Programación en Redes

- Parámetros de la función “connect”:
 - sockfd: socket local, obtenido previamente llamando a socket
 - serv_addr: socket del servidor al cual se desea conectar
 - addrlen: tamaño de la estructura del socket al cual nos deseamos conectar
- En caso de éxito (se logró conexión) retornará 0 y -1 en caso de error. Posibles errores:
 - EBADF: sockfd es inválido
 - EALREADY: Ya existe una conexión en progreso
 - ETIMEDOUT: Se acabó el tiempo para intentar la conexión
 - ECONNREFUSED: El servidor rechazó la conexión

Programación en Redes

- Si no se establece la conexión inmediatamente, "connect" bloqueará la ejecución por un tiempo
- Una vez que ese tiempo se cumpla, la conexión se abortará
- Igual que en la llamada a "accept", se puede indicar que no se bloquee al momento de conectarse:
 - Se debe asignar O_NONBLOCK en el fd correspondiente
- Las conexiones se pueden terminar cerrando el socket llamando a "close(fd)"

Programación en Redes

- Ejemplo: client.c y server.c
- Como funciona el cliente:
 - Usa la estructura "sockaddr_in" para especificar una dirección AF_INET
 - Trata de conectarse a un servidor en el host con IP: 127.0.0.1
 - Se utiliza la función "inet_addr" para convertir la dirección a un formato de dirección del socket
- Como funciona el servidor:
 - Crea un AF_INET socket domain, y espera conexiones
 - El socket está asociado al puerto dado

Programación en Redes

- Como funciona el servidor:
 - La dirección especificada determina que computadores podrían conectarse
 - Especificando la dirección 127.0.0.1 (igual en el cliente) las comunicaciones se restringen a la máquina local
 - Si se desea que el servidor se comunique con clientes remotos, se deben especificar direcciones adecuadas

Programación en Redes

- Con el comando "netstat" podemos ver las conexiones existentes
 - Muestra las conexiones cliente/servidor esperando ser cerradas
 - Las conexiones se cierran después de un pequeño timeout
- ¿Por que el puerto del servidor es distinto al que dimos anteriormente?
 - Los números de puerto y las direcciones son comunicadas en los sockets como números binarios
 - Computadores distintos pueden usar distintos ordenamientos para los enteros

Programación en Redes

- Para que computadores de distintos tipos puedan comunicarse sin problemas, con enteros multibytes sobre una red:
 - Es necesario definir un orden de comunicación
 - Los clientes y servidores deben convertir desde su representación interna a “network order” antes de transmitir
 - Para eso existen las funciones:

```
unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int netshort);
```

Programación en Redes

- Estas funciones permiten convertir enteros de 16 y 32 bit del formato nativo al estándar
- Los nombres son abreviaciones, 'host to network long' es htonl
- Para los computadores que utilicen el mismo orden que el de la red, estas operaciones no tienen efecto
- Independiente del orden utilizado por una máquina, es altamente recomendable utilizar estas funciones:
 - Los programas cliente y servidor funcionarán sin problemas en distintas arquitecturas

Programación en Redes

- Los sistemas Unix que proveen un número determinado de servicios lo hacen a través de un “super-servidor”:
 - Internet Daemon (inetd), escucha conexiones en varios puertos a la vez
 - Cuando un cliente se conecta a un servicio, inetd ejecuta el programa servidor apropiado
 - No hay necesidad de tener servidores funcionando todo el tiempo
 - Generalmente por medio de un archivo de configuración (/etc/inetd.conf) se indica que se debe hacer para cada servicio

Múltiples Clientes

- Hasta el momento, cuando se establece una conexión:
 - Se crea un nuevo socket en el servidor y el original sigue esperando nuevas conexiones
 - Si el servidor no puede atender inmediatamente las conexiones, estas quedarán en una cola
- El hecho de que el socket original se mantenga disponible y que los podamos usar como file descriptor:
 - Nos entrega las herramientas para poder atender múltiples clientes al mismo tiempo

Múltiples Clientes

- Si el servidor ejecuta una llamada a “fork”, se creará una segunda copia de si mismo:
 - El socket abierto será heredado a su proceso hijo
 - El hijo se puede comunicar con el cliente, mientras el padre se preocupa de seguir aceptando futuras conexiones
 - Ejemplo: server2.c
- Como funciona:
 - El programa servidor crea un proceso hijo para manejar cada nueva conexión
 - El proceso hijo espera 5 segundos para finalizar:
 - Simula algún procesamiento, conexión a base de datos, etcétera

Múltiples Clientes

- Como funciona el servidor:
 - El programa servidor utiliza fork para manejar múltiples clientes
 - En una aplicación de bases de datos, no es la mejor solución: el programa será muy grande y complicado de manejar el acceso coordinado de múltiples servidores
 - Se necesita un servidor que maneje múltiples clientes sin bloquearse, esperando a nuevos clientes
 - La solución es manejar múltiples file descriptors a la vez, pero sin las limitantes de los sockets

Múltiples Clientes

- Comúnmente una aplicación Unix necesita saber el estado de un número de entradas para determinar la siguiente acción a ejecutar
 - Ejemplo: Un programa de comunicaciones (terminal emulador) necesitará leer del teclado y del puerto serial al mismo tiempo
 - Se podría implementar ciclos infinitos de lectura de todas las entradas, pero es muy costoso en términos de CPU
- La función “select” nos permite esperar a una entrada en un número de file descriptors a la vez
 - Un terminal se podría bloquear hasta que algo llegue
 - Un servidor podría tener múltiples clientes esperando un requerimiento en varios sockets a la vez

Múltiples Clientes

- La función “select” trabaja con una estructura “fd_set” la cual refleja un grupo de file descriptors abiertos
- Existen macros para manejar este grupo:

```
FD_ZERO(fd_set *set);
FD_CLR(int fd, fd_set *set);
FD_SET(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
```

- Inicializa, borra o añade un fd al set, mira si un fd pertenece al set

Múltiples Clientes

- El número máximo de file descriptor en una estructura fd_set está definido por la constante: FD_SETSIZE
- La función select puede usar un valor de timeout para no bloquearse infinitamente:
 - Utiliza una estructura timeval:

```
struct timeval{
    time_t tv_sec;    /* segundos */
    long   tv_usec;   /* micro-segundos */
}
```

Múltiples Clientes

- La llamada a select:

```
int select(int n, fd_set *readfds,
           fd_set *writefds, fd_set *exceptfds,
           struct timeval *timeout);
```

- n indica el número de file descriptors que serán testeados
- select retornará si:
 - cualquier fd de “readfds” está listo para lectura
 - cualquier fd de “writefds” está listo para escritura
 - cualquier fd de “exceptfds” tiene una condición de error

Múltiples Clientes

- Si ninguna de las condiciones anteriores se cumple, retornará en un tiempo “timeout”
- Si el timeout es NULL y no hay actividad en los fds, se bloqueará eternamente
- Cuando select retorna, los sets de fds estarán modificados para indicar cual fd está listo para lectura, escritura o tiene error
 - Utilizar FD_ISSET para probar los fds y saber cual es el que necesita atención
- La llamada a select retornará el número total de fds en el set modificado, -1 en caso de error

Múltiples Clientes

- Ejemplo: select.c
- El programa servidor podría utilizar “select” para manejar múltiples clientes simultáneamente
- El servidor puede usar select para el socket que espera nuevos clientes como el socket de conexión con cada cliente
 - Se debe utilizar de la manera apropiada FD_ISSET (recorrer los sockets en búsqueda de actividad)
 - Ejemplo: server3.c