UNIVERSIDAD DE EL SALVADOR
FACULTAD DE INGENIERIA Y ARQUITECTURA
ESCUELA DE INGENIERIA ELECTRICA

"APLICACION DE MODELOS DE INTERCONEXION Y COMUNICACION DE DATOS"

PRESENTADO POR:

JUAN MANUEL GUERRERO SÁNCHEZ.
WALTER OVIDIO SÁNCHEZ CAMPOS

PARA OPTAR AL TITULO DE
INGENIERO ELECTRICISTA

CIUDAD UNIVERSITARIA, SEPTIEMBRE DE 2000.

UNIVERSIDAD DE EL SALVADOR

RECTORA :
Dra. María Isabel Rodríguez.

SECRETARIA GENERAL :
Licda. Lidia Margarita Muñoz Vela

FACULTAD DE INGENIERIA Y ARQUITECTURA

DECANO :
Ing. Alvaro Antonio Aguilar Orantes

SECRETARIO :
Ing. Saúl Alfonso Granados

ESCUELA DE INGENIERIA ELECTRICA

DIRECTOR :
Ing. Ricardo Alfredo Colorado Eméstica.

UNIVERSIDAD DE EL SALVADOR
FACULTAD DE INGENIERIA Y ARQUITECTURA
ESCUELA DE INGENIERIA ELECTRICA

Trabajo de Graduación previo a la opción de:
INGENIERO ELECTRICISTA

Título :

"APLICACION DE MODELOS DE INTERCONEXION Y COMUNICACION DE DATOS"

Presentado por :

JUAN MANUEL GUERRERO SÁNCHEZ.
WALTER OVIDIO SÁNCHEZ CAMPOS

Trabajo de Graduación aprobado por :

Coordinador :

Ing. José Roberto Ramos.

Asesor :

Ing. Carlos Eugenio Martínez Cruz.

San Salvador, Septiembre de 2000.

Trabajo de Graduación aprobado por:

Coordinador y asesor :

Ing. José Roberto Ramos.

Asesor :

Ing. Carlos Eugenio Martínez Cruz.

ACTA DE CONSTANCIA DE NOTA Y DEFENSA FINAL

En esta fecha, 25 de septiembre de 2000 en el local de Sala de Lectura de la Escuela de Ingeniería Eléctrica, a las diecisiete horas y treinta minutos, en presencia de las siguientes autoridades de la Escuela de Ingeniería Eléctrica de la Universidad de El Salvador:

Firma:

1- Ing. Ricardo Alfredo Colorado
Director

ESCUELA DE INGENIERIA ELECTRICA
FACULTAD DE INGENIERIA
Y ARQUITECTURA
Universidad de El Salvador

Y, con el Honorable Jurado de Evaluación integrado por las personas siguientes:

Firma:

1- Ing. Julio Quijano

2- Ing. Werner David Meléndez

Se efectuó la defensa final reglamentaria del Trabajo de Graduación:

"Aplicación de Modelos de Interconexión y Comunicación de Datos"

A cargo de los Bachilleres:

GUERRERO SANCHEZ, JUAN MANUEL
SANCHEZ CAMPOS, WALTER OVIDIO

Habiendo obtenido el presente Trabajo una nota final, global de: 9.2

(NUEVE PUNTO DOS)

AGRADECIMIENTOS

Dedico este trabajo a:

Don Antonio Guerrero.
(Q.E.P.D.)

Agradezco a:

Dios Todopoderoso por iluminar mi camino.

Mis padres: Sonia y Sócrates por haberme ayudado y apoyado en todo momento.

Mis abuelos Angelita, Adelita y Juan Manuel.

Mis hermanos.

A todos mis amigos por su apoyo.

Todas aquellas personas que de una u otra manera me ayudaron a llevar a feliz término el presente trabajo.

Manuel Guerrero.

AGRADECIMIENTOS

Dedico este trabajo a:

Jesucristo mi Salvador Personal.

Agradezco a:

Dios Todopoderoso por guiar mi pasos por buen camino.

Mis padres: Mirian Sánchez y José Campos por haberme ayudado y apoyado en todo momento.

A todos mis amigos por su apoyo.

Todas aquellas personas que de una u otra manera me ayudaron a llevar a feliz término el presente trabajo.

Ovidio Campos.

Tabla de Contenidos

Indice de Tablas

Indice de Figuras

PREFACIO

En los tiempos actuales donde la comunicación de datos y las redes de comunicación tienen un gran auge, es necesario que los ingenieros electricistas con orientación telemática adquieran conocimientos concernientes al funcionamiento de las comunicaciones en sistemas operativos tipo UNIX/LINUX, ya que estos sistemas operativos multiusuario/multitarea poseen excelentes características orientadas a las comunicaciones. Los ingenieros electricistas también deben ser capaces de desarrollar y entender a fondo el funcionamiento de las aplicaciones de comunicaciones de datos cliente-servidor ampliamente utilizadas en estos sistemas operativos.

En base a lo anteriormente mencionado, el presente texto tiene como objetivo fundamental que el lector comprenda el funcionamiento de las comunicaciones de datos en los sistemas operativos UNIX/LINUX, los modelos de interconexión y comunicación de datos siguientes: Sockets, el estándar de representación de datos XDR y las llamadas a procedimientos remotos (RPC).

Como un objetivo adicional está el presentar guías de laboratorio que traten de forma práctica los diferentes tópicos cubiertos en cada uno de los capítulos del presente documento y que por medio de ejemplos prácticos y tareas propuestas que puedan ser desarrolladas en un entorno de red, ayuden una mejor comprensión de los tópicos presentados y a la vez sirvan de apoyo para asignaturas futuras a impartirse en la Escuela de Ingeniería Eléctrica, afín a las comunicaciones de datos.

RESUMEN

El texto se ha dividido en tres capítulos principales, anexando guías de laboratorios y ejercicios propuestos para los diversos tópicos de los capítulos, una descripción resumida de los protocolos utilizados a lo largo del documento para una mejor referencia (Protocolos IP, TCP, UDP e ICMP).

El primer capítulo introduce el modelo de programación de aplicaciones distribuidas basado en sockets. Esta interfaz, que nació con la versión 4.3 del UNIX de Berkeley, se ha mostrado tan sencillo, universal y práctico para intercomunicar procesos, que es el que prácticamente incorpora todos los sistemas operativos basados en UNIX/LINUX como estándar de comunicaciones.

El segundo capítulo presenta el estándar de representación de datos XDR, que permite aislar los procesos de comunicaciones de la arquitectura interna de la máquina y, por tanto, facilitar aún más al programador de aplicaciones en red la portabilidad de sus programas a otras arquitecturas. Este estándar es un subconjunto de la llamada a procedimientos remotos (RPC).

El tercer capítulo, por su parte, describe la llamada a procedimientos remotos (RPC) diseñada por Sun Microsystems. Se cubren los Niveles Alto, Medio y Bajo de programación con RPC. Uno de los intereses mas notorios de este capítulo es explicar como eximir al programador de la manipulación de los sockets para la mayoría de las aplicaciones utilizando la programación RPC, ya que con ayuda del compilador `rpcgen`, la creación de aplicaciones distribuidas con RPC es reducida en complejidad ya que este compilador crea los esqueletos necesarios de la aplicación deseada.

Como requisitos previos, de cara aprovechar al máximo el tiempo y el esfuerzo invertido en la lectura de los tópicos que se cubren y al desarrollo de las guías de laboratorios, se recomienda que lector disponga de un sólido conocimiento previo de programación en lenguaje C ANSI (haciendo mayor énfasis en punteros y estructuras), conceptos generales de comunicación de datos tales como: Redes LAN y WAN, Protocolos IP, TCP, UDP e ICMP así como conocimientos generales de sistemas operativos tipo UNIX/LINUX.

CAPITULO
I

SOCKETS

Introducción

Para que las comunicaciones de datos sean efectivas, se hace necesario ir más allá de los niveles físico, de enlace de datos, de red e incluso de transporte, puesto que a pesar de que los anteriores facilitan la comunicación entre dos o más terminales, no representan por sí solos un esquema completo de comunicación. Es necesario incluir también las capas de nivel superior: sesión presentación y aplicación, los cuales se encuentran inequívocamente ligados a la programación.

Este capítulo comprende los mecanismos básicos para establecer una comunicación entre dos ordenadores orientada al transporte de datos entre aplicaciones distribuidas. Introduce al concepto de socket, que es la unidad básica en lo que a programación distribuida se refiere. Asimismo, se proporcionan las bases para poder iniciar la programación en red a través de aplicaciones que usan el modelo cliente-servidor.

Como complemento, se tratan los principales archivos de configuración utilizados por sistemas operativos como UNIX y Linux que se relacionan en mayor o menor grado con las comunicaciones entre ordenadores, así como las estructuras utilizadas para accesar a su contenido.

Los sockets proveen una interfaz lo suficientemente general como para permitir la construcción de aplicaciones basadas en red. De ahí su gran aceptación al momento de crear aplicaciones distribuidas.

1.0. El Modelo Cliente-Servidor

En los principios de la informática, las aplicaciones dentro de una red eran almacenadas en grandes ordenadores y cargados en su memoria para ser ejecutados en terminales no inteligentes. Actualmente esto ha ido cambiando hasta llegar al punto en que dichas aplicaciones se distribuyen en al menos dos partes, ejecutándose cada una en un procesador diferente.

Una de esas partes se encuentra en un proceso llamado *cliente*, que emite peticiones, y la otra en un proceso llamado *servidor*, que escucha y atiende dichas peticiones. Este modelo es muy usado en comunicaciones tanto orientadas a la conexión como no orientadas a conexión.

Una ventaja de este modelo respecto al usado anteriormente es que tanto el cliente como el servidor son procesos independientes, lo que implica:

- *Transparencia en la localización de los procesos:* Tanto el proceso cliente como el proceso servidor pueden estar en cualquier lugar de la red.
- *Transparencia en cuanto al tipo de máquina y sistema operativo:* Los procesos clientes o servidores pueden entenderse sin importar el sistema operativo o el tipo de máquina.
- *Facilidad de crecimiento:* Resulta sumamente fácil agregar más ordenadores o estaciones de trabajo como clientes o servidores.

- *Mayor Fiabilidad:* La facilidad de crecimiento permite tener servidores redundantes para que en caso uno falle, otro tome su lugar, evitando la interrupción de la actividad de todo el sistema.

1.0.1. Asociaciones y Sockets

La comunicación entre procesos cliente y servidor se puede dar a través de uno o varios canales, por lo que se hace necesario identificar a cada canal por sus características particulares. Al conjunto de parámetros o coordenadas que identifican de forma única un canal de comunicación entre dos procesos se le llama *asociación*.

Una asociación se compone de 5 parámetros:

1. **Tipo de Protocolo:** Se refiere al protocolo usado a nivel de capa de transporte.
2. **Dirección local:** Número de red (Dirección IP) de máquina local.
3. **Proceso Local:** Proceso que se ejecuta en la máquina local.
4. **Dirección remota:** Número de red (Dirección IP) de máquina remota.
5. **Proceso remoto:** Proceso que se ejecuta en la máquina remota.

Por ejemplo, una estación que hace las veces de cliente intenta conectarse a un servidor cuya dirección es 194.12.220.5 usando FTP (el uso de FTP indica de manera implícita que se usará TCP a nivel de capa de transporte). El puerto donde se sabe se localiza el proceso servidor (puerto remoto) al que se conectará es el 1509. La dirección IP local es 205.4.8.10, mientras que el sistema operativo escoge para esta conexión el puerto 2574. Con estos datos, la asociación quedaría formada de la siguiente manera:

```
{ Protocolo, Dirección Local, Proceso Local, Dirección Remota, Proceso Remoto } =
                    { TCP ,194.12.220.5,1509 , 205.4.8.10,2574 }
                              (Servidor)              (Cliente)
```

Un proceso trabaja con un canal de comunicación de forma muy similar a como un archivo es definido por un descriptor del mismo. Un descriptor de canal es conocido como ***socket*** (a veces conocido también como *"conector lógico"* o *"zócalo"*). Cuando se desea enviar o recibir datos, se escribe o lee el socket correspondiente. El interfaz socket está disponible a partir de la versión 4.1cBSD[1]

Además de los sockets para el desarrollo de aplicaciones referentes a programación distribuida, existe la Interfaz de capa de Transporte (TLI: *Transport Layer interfaz*). Sin embargo, esta otra interfaz no es tan difundida como los sockets y por lo tanto, es soportada por un menor número de plataformas.[2]

No puede haber 2 canales exactamente iguales entre 2 procesos, ya que no podría diferenciar a cuál de ellos se envía o reciben datos, por lo que el sistema operativo no lo permite generando un mensaje de error. Una variación en uno de los cinco parámetros que forman la asociación es suficiente para definir un canal diferente.

La información de estos 5 parámetros se distribuye entre las capas 3 (capa de red) y 4 (capa de transporte) del modelo OSI:

- En la cabecera del protocolo de red (generalmente IP) suele estar contenida la información sobre el protocolo de capa de transporte usado y las direcciones de red.
- La cabecera del protocolo de capa de transporte (generalmente TCP o UDP) contiene los números de puerto.

Cada protocolo tiene asignados 65535 puertos (2^{16} -1).

1 Berkeley Software Distribution.
2 La mayoría de distribuciones de Linux disponibles actualmente no soportan TLI.

1.1. Esquema de Funcionamiento

A continuación se muestran las partes básicas que componen a una aplicación cliente-servidor, tanto para un servicio no orientado a la conexión (por ejemplo, UDP) como orientado a la conexión (por ejemplo, TCP).

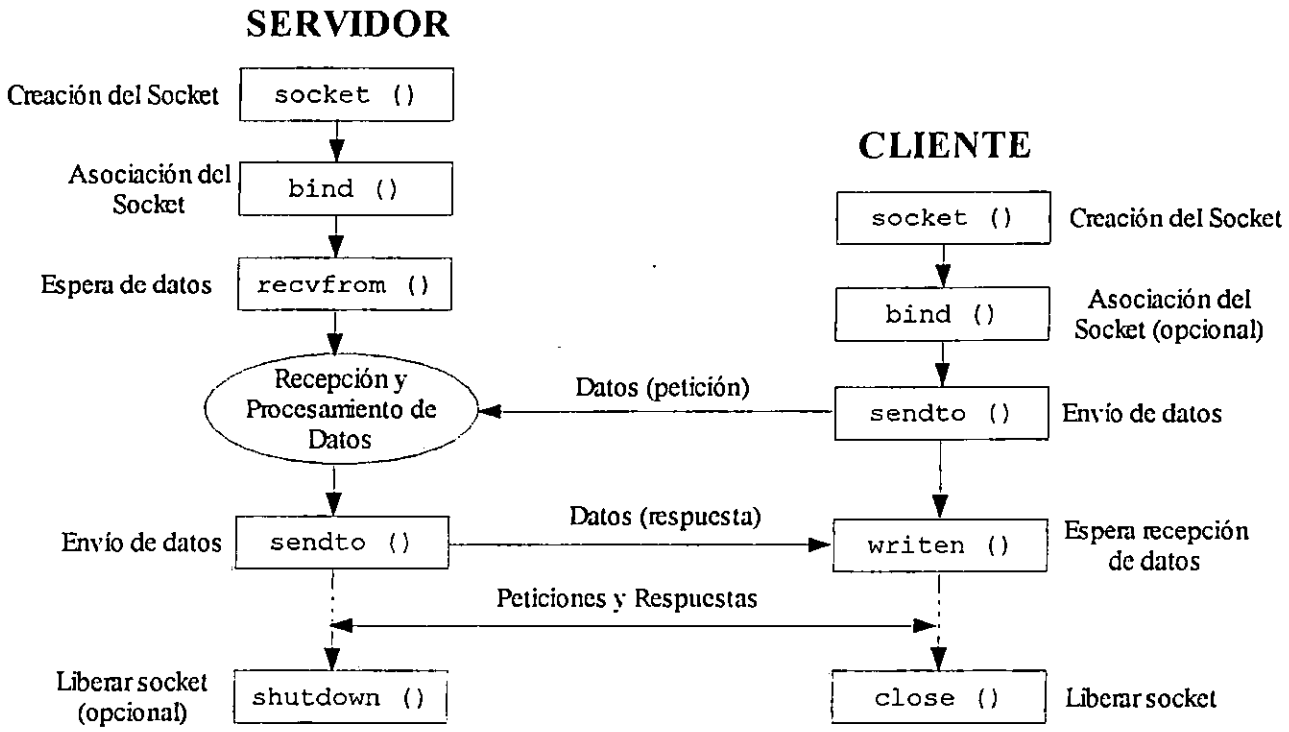


Figura 1: Aplicación Cliente-Servidor orientada a la NO conexión.

En un servicio no orientado a la conexión, los datos son simplemente enviados a través de la estructura de transmisión hacia el destino. No hay garantía de entrega de los datos y la única forma de saber si fueron recibidos es que el receptor envíe un nuevo mensaje haciéndolo saber.

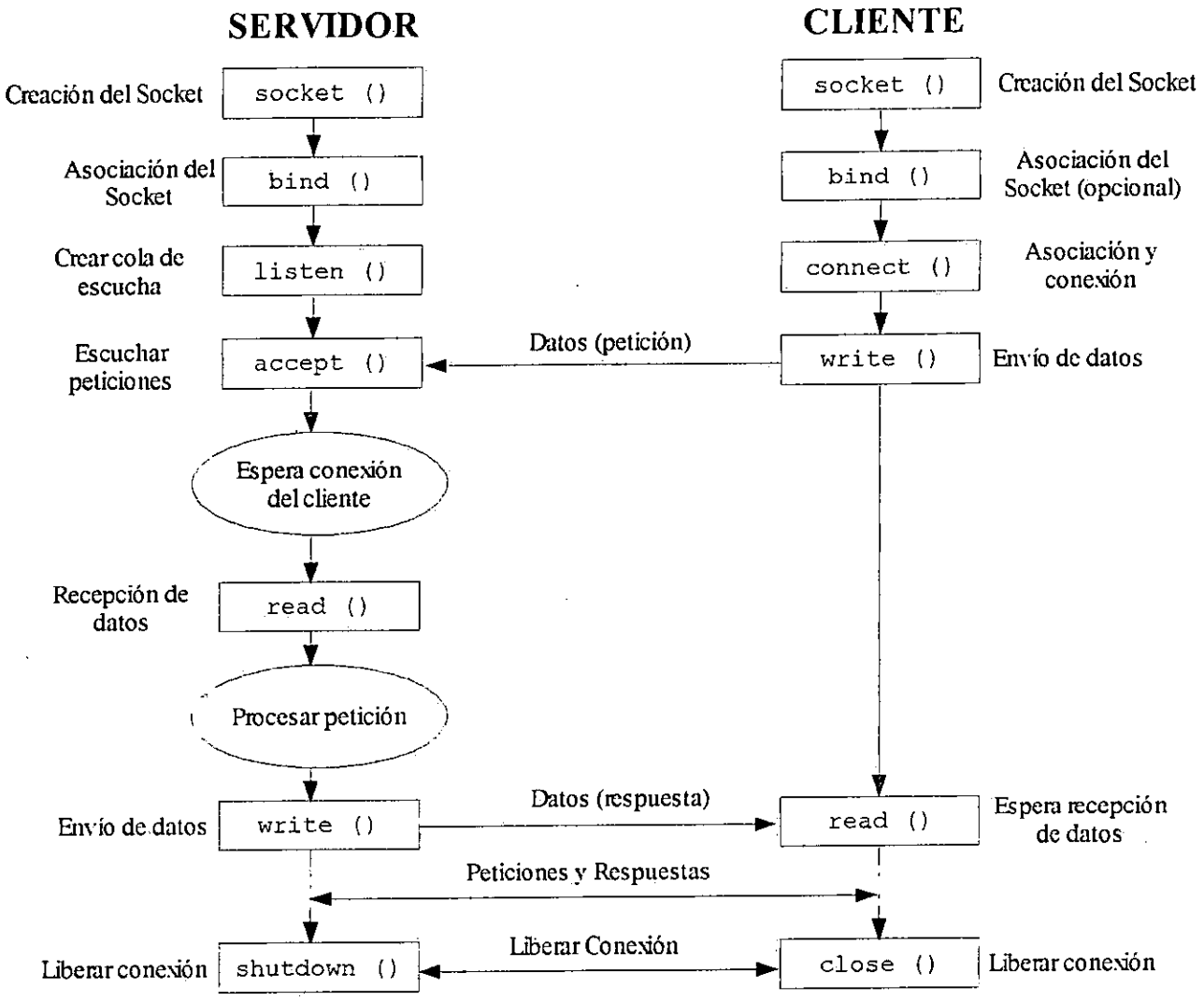


Figura 2: Aplicación Cliente-Servidor orientada a la conexión.

En un servicio orientado a la conexión, el proceso de comunicación de divide en 3 partes:

1. Establecimiento de la conexión.
2. Uso de la conexión, por medio del intercambio de datos de peticiones y respuestas.
3. Liberación de la conexión.

En este modelo, hay garantía de entrega de datos o notificación en caso de que no pueda lograrse dicha entrega.

1.2. Estructura de Dirección Socket

Para establecer las comunicaciones entre procesos utilizando sockets, es necesario definir primero la *familia* o *dominio* del mismo. Este especifica el formato de las direcciones que se podrán dar a los sockets y los diferentes protocolos soportados por la comunicación entre sockets. Todo dominio posee su propia estructura especialmente creada conocida como *estructura de dirección socket*.

La mayoría de las funciones relacionadas con sockets requieren un puntero a una estructura de dirección socket como argumento. Cada conjunto de protocolos (dominio) define su propia estructura de dirección socket. Los nombres de dichas estructuras inician con las letras `sockaddr_` y finalizan con un sufijo único para cada conjunto de protocolos. En el caso de UNIX/LINUX, estas estructuras pueden ser pasadas tanto de un proceso al kernel (núcleo) como del kernel a un proceso.

1.2.1. Estructura de Dirección Socket Genérica

Las estructuras de dirección socket son pasadas siempre por referencia[3] cuando se pasan como argumento a cualquiera de las funciones relacionadas con sockets. Sin embargo, la estructura puede variar dependiendo del conjunto de protocolos usados ya que por cada dominio existente, se tiene una estructura de dirección socket específica para ese dominio. Con esto se tendría el inconveniente de necesitar un conjunto de funciones dedicadas a manejar solamente un tipo de estructura de dirección socket, lo que resultaría impráctico. Es por ello lo que dichas funciones deben ser capaces de manejar un formato estándar en lugar de adaptarse a un tipo de estructura de dirección socket específica.

Dominio	*Estructura de Dirección Socket Asociada*
IPV4 (AF_INET)	`sockaddr_in`
IPV4 (AF_INET6)	`sockaddr_in6`
UNIX (AF_UNIX / AF_LOCAL)	`sockaddr_un`
Apple Talk (AF_APPLETALK)	`sockaddr_at`
Xerox NS (AF_NS)	`sockaddr_ns`

Tabla 1: *Ejemplos de dominios y sus respectivas estructuras de dirección socket asociadas.*

Para solucionar este problema, en 1982 se creó una estructura de dirección socket genérica definida en el archivo de cabecera `<sys/socket.h>`[4]. Actualmente, la mayoría de distribuciones define esta estructura de la siguiente forma:

```
struct sockaddr
{         short int    sa_family;      // Familia de la dirección (2 bytes)
          char         sa_data[14];    // Dirección dependiente del protocolo (14 bytes).
}
```

Como puede apreciarse, esta estructura posee una longitud fija de 16 bytes.

Todas las funciones que usan el direccionamiento a un socket deberán utilizar esta estructura, es decir, se necesita hacer una conversión de la estructura específica (que depende del conjunto de protocolos usados) a la estructura genérica en la propia llamada a la función. Desde el punto de vista del programador, la conversión a un tipo de estructura de dirección socket genérica es la única utilidad de esta estructura.

Para ilustrar el uso de la estructura de dirección socket genérica, nos auxiliamos del siguiente ejemplo.

---

3 Esto se refiere a que se pasa un puntero a una variable en lugar de pasar dicha variable.

4 En los sistemas operativos UNIX y Linux, los archivos de cabecera o de inclusión se encuentran generalmente en el directorio `/usr/include`.

Ejemplo:

Considere la función definida a continuación:

```
void conectar (struct sockaddr *dir)
```

Esta función, como todas las que pasan como argumentos estructuras de dirección socket, toman como parámetro de entrada el tipo de estructura genérica. Ahora considere la variable llamada `dirección_servidor`, la cual es una estructura de dirección socket definida según un conjunto de protocolos X. Si se hace una llamada a dicha función de la forma:

```
conectar (&dirección_servidor)
```

Al compilar el programa, se generaría un mensaje de advertencia indicándonos la incompatibilidad entre el tipo de estructura esperada y el ingresado a la función. Esto puede corregirse mediante la conversión de la variable a una estructura de dirección socket genérica, que es lo que la función `conectar` espera como entrada, así:

```
conectar ((struct sockaddr *)&dirección_servidor)
```

Esta conversión será necesaria en todas las funciones que hagan referencia a una estructura de dirección socket.

1.2.2. Estructura de Dirección Socket IPV4

Una estructura de dirección socket, conocida también como *"Estructura de Dirección Socket Internet"*, utiliza la estructura `sockaddr_in` definida por el archivo de cabecera `<netinet/in.h>`. Su definición es la siguiente:

```
struct sockaddr_in
{     short int          sin_family;     // Familia de Dominio (2 bytes)
      unsigned short int sin_port;       // Número de puerto de capa 4 , con ordenamiento
                                         // de bytes según el formato de red. (2 bytes).
      struct in_addr     sin_addr;       // Dirección IPV4 de 32 bits, con ordenamiento de
                                         // bytes según el formato de red (4 bytes).
      char               sin_zero[8];    // Sin uso (8 bytes).
}
```

El miembro `sin_addr` es a su vez otra estructura que está constituida de la forma siguiente:

```
struct in_addr
{
      unsigned long     s_addr;          // Dirección IPV4 de 32 bits, con ordenamiento de
                                         // bytes según el formato de red
}
```

Algunas distribuciones incluyen como primer miembro de la estructura `sockaddr_in` un valor tipo entero de 8 bits llamado `sin_len`, que fué agregado junto con el soporte para protocolos OSI. No todos los vendedores soportan un campo de longitud para las estructuras de dirección socket. El estándar Posix.1g[5], que actualmente es uno de los más usados, no necesita de este miembro ya que no hace falta en ningún momento darle un valor alguno ni tampoco examinarlo. Es usado en el kernel (núcleo) por las rutinas que manejan estructuras de dirección socket de varias familias de protocolos.

El estándar Posix.1g requiere solamente de 3 miembros: `sin_family`, `sin_addr` y `sin_port`. Es aceptable para una implementación compatible con Posix definir miembros adicionales, lo cual también es normal para una estructura de dirección socket Internet. La gran mayoría de las implementaciones agregan el miembro `sin_zero`[6].

5 POSIX: Portable Operating System Interface.

El miembro `sin_family` indica el tipo de dominio o familia de la estructura. Es usado para cuando esta estructura sea convertida a un tipo genérico, las funciones relacionadas con sockets aún puedan identificar el tipo de estructura de dirección socket que era originalmente. Para el caso de IPV4, su valor es `AF_INET`.

Tanto las direcciones IP como TCP/UDP deben ser guardadas dentro de la estructura siguiendo el orden de formato de red.

Por convención, toda estructura de dirección socket es puesta a cero (0) antes de poner datos en ella.

1.3. Ordenamiento de Bytes

Debido a que en una red pueden haber ordenadores de cualquier tipo comunicándose entre si, surge el problema de que diferentes fabricantes manejan diferentes representaciones de datos al interior de sus equipos. Para ilustrar el problema, considere un entero de 16 bits (2 bytes). Algunos ordenadores guardarán en memoria este dato almacenando el byte menos significativo (LSB) en la posición inicial de memoria que ocupará el dato, mientras que otros guardarán en la primera dirección de memoria el byte más significativo (MSB). A este tipo de ordenamiento se le conoce como *little-endian"* y *"big-endian"* respectivamente. Estos términos hacen referencia a cual de los bytes se almacena en la primera dirección de memoria.

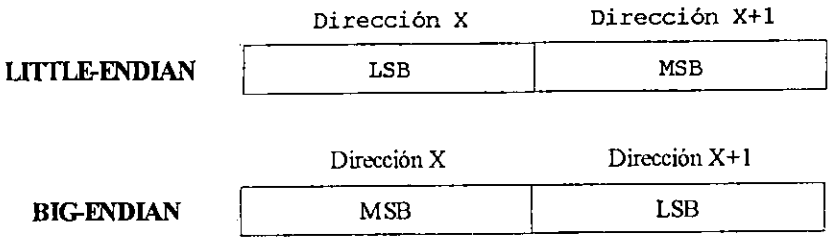


Figura 3: Representación de datos Little-endian y Big-endian.

Los protocolos de red especifican un ordenamiento de bytes específico, por lo que es necesario realizar conversión entre la representación de datos del ordenador anfitrión (Host Byte Order) y la representación de datos de la red (Network Byte Order). Por ello se hace necesario el uso de las siguientes funciones, definidas en el archivo de cabecera `<netinet/in.h>`:

```
unsigned short int      htons (unsigned short int valor_16_bits_anfitrión)
unsigned long int       htonl (unsigned long int valor_32_bits_anfitrión)
unsigned short int      ntohs (unsigned short int valor_16_bits_red)
unsigned long int       ntohl (unsigned long int valor_32_bits_red)
```

NOTA: Las palabras en cursiva representan los argumentos de la función.

htons (Host to Network Short): Convierte un entero corto en formato del anfitrión a formato de red.
htonl (Host to Network Long): Convierte un entero largo en formato del anfitrión a formato de red.
ntohs (Network to Host Short): Convierte un entero corto en formato del red a formato de anfitrión.
ntohl (Network to Host Long): Convierte un entero largo en formato del red a formato de anfitrión.

Cuando se usan estas funciones, no necesitamos preocuparnos por el formato de ordenamiento tanto del anfitrión como de la red. La conversión es hecha ajustándose a los requerimientos de la red o del ordenador anfitrión, según el caso.

6 El miembro `sin_zero` se introduce con el fin de que la longitud de la estructura `sockaddr_in` iguale a la estructura de dirección genérica `sockaddr`.

1.4. Manipulación de Bytes

Para manejar datos como estructuras de dirección socket, se necesitan funciones de cadena que no interpreten de modo especial el código cero (0) y además permitan trabajar con cadenas que no terminen con un carácter nulo. El estándar 4.2 de BSD define las siguientes funciones definidas en el archivo de cabecera <strings.h>:

```
void    bzero (void *destino, int longitud)
void    bcopy (const void *fuente, void *destino, int n)
int     bcmp (const void *cadena1, const void *cadena2, int n)
```

bzero: Rellena la cadena destino, cuyo tamaño está definido en longitud, con el código cero (0). Esta función se usa para inicializar las estructuras de dirección socket antes de colocar datos en ellas.

bcopy: Copia n bytes de la cadena fuente en la cadena destino.

bcmp: Compara los primeros n bytes de las cadenas fuente y destino. Devuelve 0 si son iguales y un número diferente de cero si no lo son, generalmente indicando el primer byte donde no hubo coincidencia.

1.5. Conversión de Direcciones

Las direcciones de Internet vienen dadas en un formato fácilmente comprensible por los humanos, pero relativamente difícil de interpretar para un ordenador. Existen funciones que convierten direcciones de Internet entre cadenas ASCII y valores binarios en ordenamiento de red. Estas funciones están definidas en el archivo de cabecera <arpa/inet.h> y son:

```
int             inet_aton (const char *cadena, in_addr *dirección)
unsigned long   inet_addr (const char *cadena)
char           *inet_ntoa (struct in_addr estructura_dir)
int             inet_pton (int familia, const char *cadena, void *dirección)
const char     *inet_ntop (int familia, const void *dirección, char *cadena,
                           int tamaño)
```

inet_aton: Convierte una cadena numérica con formato de dirección IPV4 (xxx.xxx.xxx.xxx) a un entero de 32 bits ordenado en formato de red. Devuelve 1 si la cadena es válida y 0 en caso de error.

inet_addr: Realiza exactamente la misma conversión que inet_aton, pero devuelve el entero de 32 bits ordenado en formato de red si la conversión tiene éxito o la constante INADDR_NONE en caso de error.

inet_ntoa: Convierte un entero de 32 bits ordenado en formato de red a cadena numérica en formato IPV4. La función devuelve un puntero a la cadena destino.

inet_pton: Convierte una dirección de red en formato de presentación (ASCII) a numérico con ordenamiento de red. Devuelve 1 si la función tiene éxito, 0 si el formato de la cadena numérica no es correcto según la familia especificada y -1 en caso de error. Esta función es ampliamente usada ya que soporta los formatos tanto IPV4 como IPV6. El argumento familia puede tomar los valores **AF_INET** (para IPV4) o **AF_INET6** (para IPV6). El resultado se guarda en dirección.

inet_ntop: Realiza la conversión opuesta a inet_pton, de formato numérico con ordenamiento de red a cadena ASCII cuyo formato sigue a la familia seleccionada. Si tiene éxito devuelve un puntero a la cadena resultado. Si falla devuelve NULL. Generalmente el puntero devuelto es el que corresponde al argumento dirección. por lo que en este argumento se coloca la cadena que servirá para guardar el resultado de la función. El parámetro tamaño deberá ser la longitud de dicha cadena.

Estas dos últimas funciones son nuevas, y soportan tanto direcciones IPV4 como IPV6. Es por ello que no aparecen en muchas distribuciones de UNIX o LINUX con excepción de aquellas más recientes.

Sabemos que las cadenas que representan direcciones IP tienen longitudes máximas establecidas (16 bytes para IPV4 y 46 bytes para IPV6). Para ayudar a definir el tamaño de las cadenas que contendrán estos parámetros, estas longitudes están definidas en el archivo de cabecera <netinet/in.h> del siguiente modo:

```
#define     INET_ADDRSTRLEN   16     // Para usar en combinación con AF_INET (IPV4)
#define     INET6_ADDRSTRLEN  46     // Para usar en combinación con AF_INET6 (IPV6)
```

1.6. La Interfaz Loopback

En la mayoría de casos, para poner a prueba cualquier programa diseñado para ejecutarse en red no es estrictamente necesario encontrarse en un ambiente de red. Este puede ser simulado en un solo ordenador por medio de las direcciones de *Loopback*. Cuando el primer campo de una dirección IP (en la Versión 4) vale 127, esta dirección se usa como una dirección de Loopback del sistema local. En un gran número de sistemas, esta dirección está definida en el archivo `/etc/hosts` y su valor generalmente es de 127.0.0.1 aunque, como ya se dijo, cualquier dirección del tipo 127.xxx.xxx.xxx es igualmente válida con excepción de las direcciones primera y última de este rango (127.0.0.0 y 127.255.255.255).

1.7. Asignación de Puertos

Para llevar a cabo una conexión entre un cliente y un servidor es necesario, además de proporcionar la dirección de red (IP) también el número de puerto al que el cliente se conectará al servidor. Es por eso que se hace necesario conocer la distribución de los puertos, la cual está dada de la siguiente forma:

Uso Previsto	*Puerto*
Puertos Reservados	1 -1023
Puertos Asignados por el Sistema	1024 - 5000
Puertos Libres para los Usuarios	5001 - 65535

***Tabla 1:** Asignación de puertos según la versión 4.3 de BSD.*

En los casos donde el sistema operativo es quien escoge el número de puerto, las direcciones de entre las cuales se toma la dirección comprenden el rango 1024 a 5000.

Los puertos reservados, comprendidos entre 1 y 1023 pueden ser utilizados por el programador, siempre y cuando al momento de ejecutar los respectivos programas se tengan permisos de *superusuario* (root).

1.8. Operaciones Elementales con Sockets

Existen muchas operaciones que pueden hacerse con un socket. Sin embargo, para poder comunicar dos o más ordenadores en una red, basta con saber manejar algunas funciones elementales de socket.

1.8.1. Creación de un Socket

Para llevar a cabo la entrada/salida entre dispositivos de una red, lo primero que debe hacerse es crear un socket. Esto se logra con el llamado a la función `socket`, la cual está definida en el archivo de cabecera `<sys/socket.h>` de la siguiente forma:

```
int   socket (int familia, int tipo, int protocolo)
```

Valor Devuelto: Si tiene éxito, devuelve un entero positivo que indica el número de descriptor (por eso suele ser llamado *descriptor del socket*), -1 en caso de error.

familia: Este término se conoce también como *Dominio* del Socket. Especifica el formato de direcciones que pueden dársele al socket y los protocolos soportados por las comunicaciones a través del mismo. Según el valor de este parámetro, así será la estructura de dirección socket a relacionar. Algunos de sus valores más comunes son:

Familia	*Descripción*
AF_INET	Protocolos IPV4
AF_INET6	Protocolos IPV6
AF_LOCAL	Protocolos del dominio UNIX
AF_ROUTE	Sockets de Ruteo
AF_KEY	Key Sockets
AF_CCITT	Protocolos X.25, CCITT, etc.
AF_NS	Xerox
AF_ISO	Protocolos ISO

Tabla 2: Constantes que representan Familias de Protocolos.

Además de los que aparecen en la tabla, hay otros valores que puede tomar el parámetro `familia`, pero se usan con menor frecuencia. En el presente documento solamente estudiaremos los sockets creados con la constante AF_INET, siendo la constante que se relaciona con `sockaddr_in` (IPV4).

tipo: Este parámetro suele tomar uno de 3 valores (son los valores más comunes, per no los únicos):

SOCK_DGRAM: Se usa en sockets destinados a la comunicación con protocolos no orientados a la conexión. El protocolo equivalente a `SOCK_DGRAM` es UDP. Se recomienda para aplicaciones donde lo que importa es la velocidad y simplicidad, sabiendo que el medio es altamente confiable.

SOCK_STREAM: Este valor se usa para crear sockets que permitan comunicaciones bajo protocolos orientados a la conexión, como TCP. Se recomienda para aplicaciones donde lo que importa es la fiabilidad, a costa de una mayor complicación en la programación.

SOCK_RAW: Sockets "crudos". Permite el acceso a protocolos de más bajo nivel.

protocolo: Especifica un tipo de protocolo particular. Usualmente se la da el valor de cero (0) de modo que el sistema elige entonces un protocolo por defecto asociado al tipo y dominio del socket creado. Sin embargo, en las ocasiones en que se hace necesario darle un valor diferente de cero a este argumento (suele ser el caso de los sockets "crudos"), es común usar alguna de las siguientes constantes, definidas en el archivo de cabecera `<netinet/in.h>` (no se muestran todos los posibles valores, solamente los más comunes):

Constante	*Valor*	*Significado*
IPPROTO_IP	0	Asignación a juicio del kernel (núcleo)
IPPROTO_ICMP	1	Protocolo de Control de Mensajes de Internet (ICMP)
IPPROTO_IGMP	2	Protocolo de Manejo de Grupos de Internet (IGMP)
IPPROTO_TCP	6	Protocolo de Control de Transmisión (TCP)
IPPROTO_UDP	17	Protocolo de Datagramas de Usuario (UDP)
IPPROTO_MTP	92	Protocolo de Transporte Multidifusión (MTP)
IPPROTO_COMP	108	Protocolo de Compresión de Cabecera.
IPPROTO_RAW	255	Paquetes IP "crudos"

Tabla 3: Valores más comunes para el campo `protocolo`.

No todas las combinaciones de los posibles valores de `familia` y `tipo` son válidos. A continuación se muestran las combinaciones válidas:

	AF_INET	AF_INET6	AF_LOCAL	AF_ROUTE	AF_KEY
SOCK_STREAM	TCP	TCP	X		
SOCK_DGRAM	UDP	UDP	X		
SOCK_RAW	IPV4	IPV6		X	X

***Tabla 4:** Combinaciones de FAMILIA y TIPO para la función socket.*

(Las casillas marcadas con X indican otras combinaciones válidas pero que no poseen un significado especial).

Debe aclararse que la llamada a la función `socket` genera solamente un extremo del canal de comunicación, por lo que la llamada a esta función debe ser hecha tanto en el cliente como en el servidor.

1.8.2. Enlace de un Socket

Cuando se crea un socket con la llamada al sistema `socket`, este es generado sin serle asignado a un puerto. La función `bind` permite asociar una dirección de protocolo local a un socket. Cuando se usa IP, la dirección de protocolo está compuesta por la dirección IP y la dirección TCP o UDP correspondiente al número de puerto. Esta función esta definida en el archivo de cabecera `<sys/socket.h>` de la siguiente forma:

```
int   bind (int sockfd, const struct sockaddr *dirección, int long_dir)
```

Valor Devuelto: En caso de éxito, devuelve cero y -1 si hay un error.

sockfd: Descriptor de archivo. Este parámetro es el valor devuelto por la función `socket`.

dirección: Este parámetro es un puntero a una estructura de dirección genérica (`sockaddr`) que contiene información como la familia de protocolos, tipo de protocolo de nivel 4 y la dirección de capa 3.

long_dir: Define la longitud de la estructura tipo `sockaddr` contenida en `dirección`.

Todo servidor debe registrar su puerto antes de empezar a atender solicitudes. Esto se logra con `bind`, quien informa al sistema operativo que ese puerto pertenece a dicho servidor (proceso servidor). por lo que todos los datos que se reciban o envíen serán para él. Esto ocurre independientemente si el servidor es orientado o no a conexión.

Normalmente, un cliente no suele enlazar por medio de `bind` una dirección IP a su socket. En este caso, es el sistema es quien asigna un puerto efímero (temporal) para el socket en las llamadas a `connect` o `listen`. Es normal para un proceso cliente que el kernel le asigne un número de puerto a menos que la aplicación requiera de un puerto reservado. Sin embargo, es muy raro que un servidor deje al kernel la tarea de escoger sus puertos ya que los servidores se caracterizan por sus puertos bien definidos.

Para la mayoría de máquinas que trabajan con un solo interfaz de red, es decir, con una única dirección IP, el parámetro `sin_addr` de la estructura de dirección (IPV4) suele contener la dirección IP con la cual se comunicará. Sin embargo, hay máquinas que actúan como pasarela *(gateway)* o bien son servidores que necesitan atender varias peticiones a la vez, por lo que dispondrá de más de un interfaz de red, de modo que si queremos que atienda un servicio, no podemos hacer que llame a `bind` cada vez que necesita enlazar la dirección IP de cada interfaz. En lugar de ello, se usa la constante `INADDR_ANY`, con lo que le indicamos al sistema que acepte peticiones provenientes de cualquier interfaz. Esta constante esta definida en el archivo de cabecera `<netinet/in.h>`.

Si especificamos un valor de cero (0) en la llamada a `bind`, le indicamos al sistema operativo que escoja el número de puerto. El número escogido estará comprendido entre 1024 y 5000.

1.8.3. Supresión de un Socket

Una de las formas de cerrar un socket es por medio de la función `close`. Esta función se define en el archivo de cabecera `<unistd.h>` y tiene el formato siguiente:

```
int    close (int sockfd)
```

Valor Devuelto: Cero (0) si la llamada a la función tiene éxito. -1 en caso de error.

sockfd: Entero que identifica al descriptor de archivo. Este parámetro suele ser el valor devuelto por la llamada a `socket`.

La acción por defecto de esta función es marcar el socket como cerrado y regresar al proceso inmediatamente. El socket queda inutilizado. El kernel intenta enviar los datos pendientes en cola, aún después de regresar de la función.

1.8.4. Cerrado de una Conexión

Además de la función `close`, una conexión puede cerrarse por medio de la función `shutdown`. La diferencia con esta función es que puede seleccionarse qué operaciones del socket finalizar (transmisión o recepción). Esta función se encuentra definida en el archivo de cabecera `<sys/socket.h>` y tiene el formato siguiente:

```
int    shutdown (int sockfd, int cómo)
```

Valor Devuelto: Cero (0) si la llamada a la función es exitosa, -1 en caso de error.

sockfd: Entero que representa al descriptor devuelto por la llamada a la función `socket`.

cómo: Permite controlar el proceso de desconexión. Los valores que puede tomar son:

`SHUT_WR:`	Cerrar la parte de la conexión referente a la escritura. Los datos pendientes en el buffer de escritura son enviados y se inicia la secuencia de cierre del socket
`SHUT_RD:`	Cerrar la parte de la conexión referente a la lectura. No pueden recibirse más datos en el socket y cualquier dato en el buffer de recepción es descartado.
`SHUT_RDWR:`	Cierra las conexiones para transmisión y recepción. Esto es el equivalente a llamar a la función `shutdown` 2 veces. Una con valor SHUT_RD y otra con valor de SHUT_WR.

1.9. Comunicación Usando Sockets

1.9.1. Comunicación Orientada a la No Conexión

Cuando se usa comunicación no orientada a la conexión, como es el caso de UDP, no se tiene la confiabilidad ni el control de flujo que se puede obtener con TCP, pero aún así se usa este modelo en ciertas aplicaciones dado su mayor velocidad y simplicidad. En una comunicación no orientada a la conexión, el cliente no establece una conexión con el servidor. En su lugar, simplemente se limita a enviar datagramas al servidor. De forma similar, el servidor no establece una conexión con el cliente, sino que simplemente espera a recibir datos del mismo.

Algunos ejemplos de casos en los que usar un protocolo no orientado a la conexión (como UDP) tiene más sentido que usar alguno orientado a la conexión (como TCP) son:

DNS (Domain Name System)
NFS (Network File System)
SNMP (Simple Network Management Protocol)

1.9.1.1. Transferencia de Datos No Orientados a la Conexión

Las funciones más elementales para transferir datos no orientados a la conexión (envío y recepción) están definidas en el archivo de cabecera `<sys/socket.h>` y son las siguientes:

```
int     recvfrom (int sockfd, void *buffer, int n, int banderas,
                  struct sockaddr *origen, int *long_dir)

int     sendto (int sockfd, void *buffer, int n, int banderas,
                 struct sockaddr *destino, int long_dir)
```

Valor Devuelto: Ambos devuelven el número de bytes leídos (para `recvfrom`) o escritos (para `sendto`) si la operación tuvo éxito. Se devuelve -1 en caso de error.

sockfd: Entero que identifica al descriptor de archivo. Este parámetro suele ser el valor devuelto por la llamada a `socket`.

buffer: Puntero al buffer de transmisión/recepción. Su valor depende de la aplicación.

n: Número de bytes a leer o escribir.

banderas: Contiene las banderas de opción. Generalmente este parámetro vale cero (0), especialmente cuando se usa UDP.

origen/destino: Es una estructura de dirección que apunta a la dirección de protocolo de origen/destino (dirección IP + número de puerto). Necesitamos especificar este valor en `sendto`, pero en `recvfrom` la función es quien llena esta estructura de dirección socket con la dirección de protocolo de quien envió el datagrama. Lo anterior significa que en la llamada a `sendto`, `destino` es un *valor*, mientras que en `recvfrom`, `origen` es un *valor-resultado*. Se le llama así porque la función devuelve un resultado en dicha variable.

long_dir: Especifica el tamaño de la estructura de dirección socket definida en `origen/destino`. Para la función `recvfrom`, este parámetro es un *valor-resultado* (al igual que `origen`) en el que la función escribe el número de bytes guardados en la estructura de dirección socket . Para `sendto`, este parámetro es un valor (al igual que `destino`). Observe que para la función `recvfrom`, este parámetro es un valor entero, mientras que para `sendto`, es un puntero a un entero.

Es posible escribir un datagrama de longitud cero (en su área de datos). En el caso de UDP, esto resulta en un datagrama IP conteniendo la cabecera IP (20 bytes para IPV4 y 40 para IPV6) y la cabecera UDP sin datos (8 bytes). Esto significa que un valor de retorno cero de `recvfrom` no implica necesariamente un error, ya que no quiere decir que se ha "cerrado" la conexión, pues en UDP no existe tal cosa.

En el caso de que no estemos interesados en conocer la dirección de origen de los datos (`recvfrom`), a los parámetros `origen` y `long_dir` se les pasa un valor nulo (NULL). Esto es común en el cliente, por ejemplo, cuando al recibir respuesta de una petición suya no le interesa averiguar de dónde vienen los datos, pues ya sabe que su origen es el servidor al que le solicitó la petición inicial.

Cuando se trabaja con cadenas de caracteres, se debe tener en cuenta que las funciones que leen y escriben cadenas (explicadas anteriormente) dejan los resultados en un arreglo (array) de longitud fija. Suponga el caso en que se envía una cadena hasta el primer caracter nulo (NULL): al recibir la cadena, las funciones de entrada de red la almacenan sin rellenar el resto del arreglo con caracteres nulos (NULL). Cuando se trata de la primera cadena recibida, no hay problemas, pero si la segunda cadena es menor, se visualizará la segunda cadena y el resto de la primera hasta encontrar un carácter nulo, como en el siguiente ejemplo:

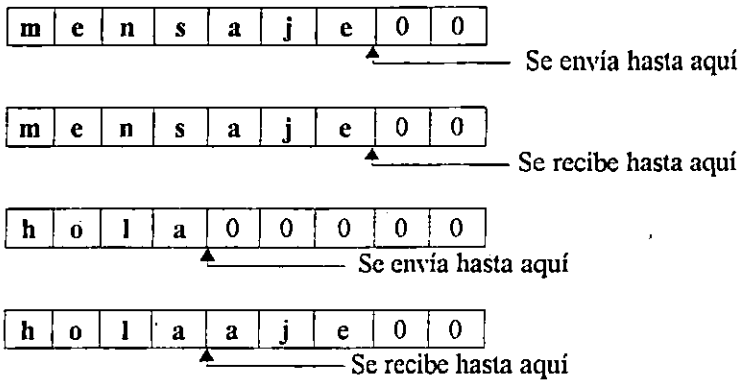


Figura 4: Ejemplo de recepción de cadenas que no terminan en caracter nulo.

La solución a este problema es, después de recibir una cadena, terminarla con un carácter nulo (NULL).

1.9.2. Comunicación Orientada a la Conexión

El el caso de la comunicación orientada a la conexión, cada extremo (el cliente y el servidor) debe encargarse de los pasos orientados al establecimiento y liberación de la conexión, tanto antes como después de la transferencia de datos.

1.9.2.1. Espera de Conexión en el Servidor

Después de crear el canal de comunicación con `socket` y asignarle una dirección de protocolo con `bind`, el siguiente paso es crear una cola de espera para las peticiones de conexión que se hacen al servidor. Esto se logra con la función `listen`. Dicha función es llamada solamente en el servidor.

Cuando se llama a esta función, un socket creado usando la función `socket` (el cual se asume es un socket activo), es convertido en un socket *pasivo*. Esto significa que el kernel aceptará solicitudes de conexión dirigidas a él en lugar de ser el socket quien intenta realizar conexiones.

La función `listen` está definida en el archivo de cabecera `<sys/socket.h>` y tiene el siguiente formato:

```
int   listen (int sockfd, int cola)
```

Valor Devuelto: Cero, si la llamada a la función es exitosa. -1 en caso de error.

sockfd: Entero que identifica al descriptor del socket. El valor dado a este parámetro generalmente es el número devuelto por la llamada a `socket`.

cola: Define el tamaño de la cola de espera. Normalmente vale 5, aunque este valor es muy pequeño considerando la cantidad de conexiones simultáneas que usualmente se le solicitan a un servidor, en especial si se trata de un servidor Web. No es recomendable especificar un valor de cero, ya que diferentes implementaciones de UNIX y LINUX interpretan esto de forma distinta.

1.9.2.2. Aceptación de Conexión en el Servidor

Una vez que hay peticiones de conexión con el servidor en la cola, es necesario aceptar dichas conexiones por medio de la función `accept`. Dicha función extrae la primera conexión pendiente en la cola creada con `listen`. En caso de no existir ninguna conexión en la cola, la función se bloquea hasta que aparezca una. En este caso se dice que el proceso es puesto "a dormir".

La función `accept` está definida en el archivo de cabecera `<sys/socket.h>`. Su formato es:

```
int   accept (int sockfd, struct sockaddr *dir_cliente, int *long_dir)
```

Valor Devuelto: Entero positivo que indica el número de descriptor. -1 en caso de error.

sockfd: Entero que representa al descriptor devuelto por la llamada a la función `socket`.

dir_cliente: Estructura de dirección socket que identifica la dirección de protocolo del proceso cliente. Este parámetro es un argumento de valor-resultado. Es importante recalcar que a menos que el elemento de la estructura de dirección socket `s_addr` sea igual a la constante INADDR_ANY, que le indica al servidor aceptar conexiones desde cualquier dirección IP, dicho servidor solamente atenderá conexiones provenientes de la dirección IP que se especifique en la mencionada variable, rechazando aquellas conexiones cuya dirección de origen no coincida con el valor especificado.

long_dir: Longitud de la estructura de dirección `dir_cliente`. Este parámetro es un argumento de valor-resultado.

La llamada a `accept` crea un nuevo socket cuyo valor de descriptor es el entero devuelto por la llamada. Este socket está conectado con un cliente cuya dirección dejará el sistema operativo en la estructura apuntada por `dir_cliente`. Asimismo, el tamaño de esta estructura devuelta por la llamada a la función aparecerá definida en `long_dir`. Si no nos interesa saber la dirección de protocolo del cliente, le damos a estos 2 parámetros el valor nulo (NULL).

El nuevo socket creado puede ser usado para enviar y recibir datos, mientras que el primer socket (el creado antes de la llamada a la función) sigue escuchando nuevas peticiones y enviándolas a la cola.

Un inconveniente de esta función es que no puede rechazar una conexión de un cliente no deseado. Para solucionar este problema, lo que se hace generalmente es aceptar la conexión, analizar su precedencia y cerrarla inmediatamente si no pertenece a un cliente deseado.

1.9.2.3. Establecimiento de la Conexión en el Cliente

En un cliente orientado a la conexión, solamente es necesario crear un socket mediante la llamada a la función `socket`. La llamada a la función `connect` permite la conexión entre el cliente y el servidor, paso necesario antes de la transferencia de datos. Esta función está definida en el archivo de cabecera `<sys/socket.h>`. Su formato es:

```
int   connect (int sockfd, const struct sockaddr *dir_servidor, int long_dir)
```

Valor Devuelto: Cero si la llamada a la función tiene éxito. -1 en caso de error.

sockfd: Entero que representa al descriptor devuelto por la llamada a la función `socket`.

dir_servidor: Estructura de dirección socket que identifica la dirección de protocolo del proceso servidor. Este parámetro es un argumento de valor-resultado.

long_dir: Longitud de la estructura de dirección `dir_servidor`.

En el caso de que la llamada a esta función falle, el socket no puede ser usado y debe ser cerrado. No es posible llamar a la función `connect` de nuevo en el socket. Si se desea intentar nuevamente la conexión, debe volverse antes a crear el socket por medio de la llamada a la función `socket`.

1.9.2.4. Envío y Recepción de Datos Orientados a la Conexión

Para la recepción y envío de datos orientados a la conexión, se usan las funciones básicas `read` y `write` respectivamente. Estas funciones aparecen en el archivo de cabecera `<unistd.h>` y su formato es:

```
int   read (int sockfd, char *mensaje, int longitud)
int   write (int sockfd, char *mensaje, int longitud)
```

Valor Devuelto: Estas funciones devuelven el número de bytes procesados (enviados o recibidos respectivamente) en caso de éxito. -1 en caso de error.

sockfd: Entero que representa al descriptor devuelto por la llamada a la función `socket`.

mensaje: La cadena a transmitir o recibir. Esta cadena no termina necesariamente con el carácter nulo.

longitud: Longitud de la cadena a transmitir o recibir.

La llamada a `write` no es bloqueante, ya que la función devuelve el control cuando se copian los datos del usuario al núcleo. Por otro lado, la llamada a `read` es bloqueante hasta que recibe datos.

Alternativamente, existen otras funciones que pueden desempañar el mismo trabajo que `read` y `write`. Estas funciones son `recv` (equivalente a `read`) y `send` (equivalente a `write`). Se encuentran definidas en el archivo de cabecera `<sys/socket.h>` y su formatos es:

```
int   recv (int sockfd, void *mensaje, int longitud, int banderas)
int   send (int sockfd, const void *mensaje, int longitud, int banderas)
```

Valor Devuelto: Estas funciones devuelven el número de bytes procesados (enviados o recibidos respectivamente) en caso de éxito. -1 en caso de error.

Como puede observarse, los primeros 3 parámetros son exactamente los mismos que con sus contrapartes `read` y `write`.

banderas: Este argumento indica las opciones de las funciones `send` y `recv`. Cuando no interesa especificar alguna opción, su valor es cero (0). Cuando se desea incluir más de una opción, estas se concatenan por medio del operando lógico OR. Los posibles valores de opciones son:

Bandera	*Descripción*	*Usada en recv*	*Usada en send*
MSG_DONTROUTE	Ignorar búsqueda en tabla de ruteo	NO	SI
MSG_DONTWAIT	Hace la función no bloqueante	SI	SI
MSG_OOB	Enviar o recibir datos fuera de banda	SI	SI
MSG_PEEK	"Espiar" mensaje entrante	SI	NO
MSG_WAITALL	Esperar a todos los datos	SI	NO

***Tabla 5:** Valores posibles que puede tomar el argumento `banderas` para las funciones `send` y `recv`.*

MSG_DONTROUTE: Esta bandera dice al kernel (núcleo) que el destino se halla en una red local, por lo que no es necesario buscar en la tabla de ruteo. Esto para una sola operación.

MSG_DONTWAIT: Especifica lectura/escritura no bloqueante para una sola operación independientemente de los valores de otras banderas del socket.

MSG_OOB: Cuando se usa con `send`, indica que se enviarán datos fuera de banda[7] (alta prioridad). Con `recv` indica que se leerán datos fuera de banda en lugar de datos con prioridad normal.

MSG_PEEK: Con esta bandera, es posible observar los datos disponibles en el buffer de recepción sin que el sistema descarte esos datos después de realizar la lectura.

MSG_WAITALL: Con esta bandera se le dice al kernel que no regrese de una operación de lectura hasta que se hayan leído el número de bytes especificados por la función. Aún así, la función puede devolver menos bytes de los solicitados en caso de:
-) Captura de señal.
-) Finalización de la conexión.
-) Error en el socket.

1.10. Tipos de Servidores

Dependiendo de la forma en que manejan la comunicación, los procesos servidores pueden clasificarse en 2 tipos básicos:
- Orientados a la No Conexión.
- Orientados a la Conexión.

Asimismo, en función de su capacidad para atender clientes, se pueden dividir en:
- Iterativos
- Concurrentes.

Los servidores iterativos son aquellos que siempre escuchan peticiones en un puerto conocido, siendo las peticiones de los clientes atendidas por un mismo proceso servidor. Si otras peticiones llegan cuando se atiende a otro cliente, el sistema operativo los guarda en una cola para atenderlos después de finalizar con la petición actual. Este tipo de servidores se usa cuando el tiempo de procesamiento de una petición sea bajo.

Un servidor concurrente estará escuchando peticiones en un puerto conocido para atender las peticiones de conexión de los clientes. Al llegar una solicitud de conexión, el proceso (proceso padre) crea una copia de sí mismo (proceso hijo) para atender la petición, siendo este último quien realmente se conecta con el cliente. Mientras el proceso hijo se encarga de la conexión con el cliente, el proceso servidor padre escucha nuevas peticiones de servicio. Los procesos servidores hijo finalizan cuando lo hacen las conexiones de los clientes.

7 TCP permite solamente el envío de 1 byte (8 bits) como datos fuera de banda por operación.

1.10.1. Servidor Iterativo No Orientado a Conexión

Los servidores iterativos no conectivos son fáciles de diseñar: programar, depurar y modificar. Son los más empleados para dar una rápida respuesta a las peticiones.

El servidor iterativo no conectivo siempre estará escuchando peticiones en un puerto conocido y todas las peticiones de los distintos clientes serán atendidas por el mismo proceso servidor. Si llegan peticiones mientras se atiende a otro cliente, el sistema operativo los guarda en una cola y después son atendidas.

El algoritmo de este servidor es muy sencillo: después de llamar a `socket()` y `bind()`, espera las peticiones con `recvfrom()` y devuelve respuesta con `sendto()` como se muestra en la siguiente figura.

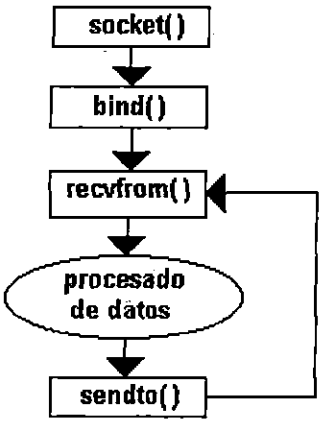


Figura 5: Servidor iterativo orientado a la NO conexión.

Este tipo de servidores se emplearán cuando el tiempo de procesamiento de una petición sea baja. Si además queremos dar una respuesta lo más rápida posible, utilizaremos un protocolo no orientado a conexión, que es mas rápido ya que no se pierde tiempo en establecer la conexión (el envío de paquetes es inmediato), las cabeceras son mas pequeñas y el tiempo de procesamiento (errores, asentimientos, control de flujo, etc) asociado al protocolo es menor.

Un servidor iterativo no orientado a la conexión se usa cuando el servicio solicitado puede ser atendido en un tiempo corto, por lo que no interesa que haya muchos procesos servidores a la espera de conexiones con los clientes. Ejemplo: echo, time, finger, etc.

1.10.2. Servidor Iterativo Orientado a Conexión

En el servidor iterativo orientado a conexión, abrimos un socket en un puerto conocido por todos los clientes esperando a que lleguen peticiones. Al llegar una petición, creamos un socket nuevo para atender al cliente y nos conectamos con él mediante un puerto temporal asignado por el sistema. Mientras se atiende a un cliente por el socket inicial, el sistema operativo almacena las nuevas peticiones de conexión que lleguan. Cuando termina un cliente, se cierra el socket actual y se abre uno nuevo si hay peticiones pendientes, que se procesan por medio de `accept()`.

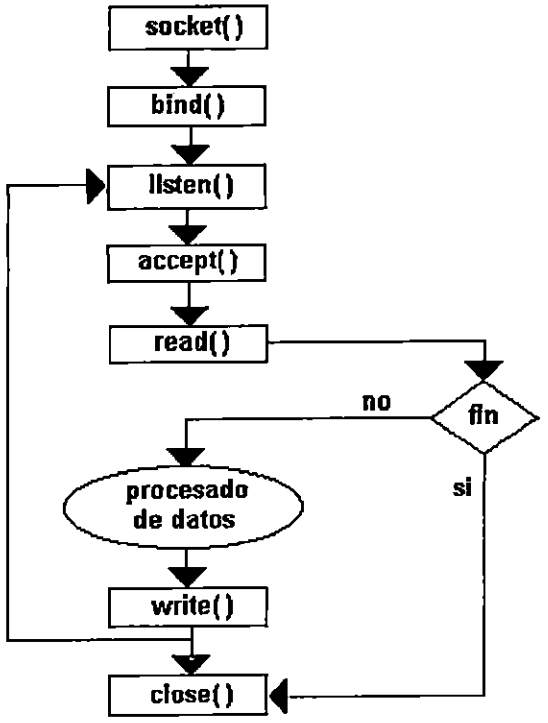


Figura 6: Servidor iterativo orientado a la conexión.

1.10.3. Servidor Concurrente Orientado a Conexión

Los servidores concurrentes se emplean cuando se dan algunas de las siguientes circunstancias:

- Es preciso dar respuesta rápida a las solicitudes de múltiples clientes.
- Cuando, para atender a un cliente, es preciso resolver ciertas funciones de entrada-salida de forma rápida y eficiente, atendiendo las distintas solicitudes de manera concurrente de forma que se pueda simultanear la atención por la CPU, con las transferencias de entrada-salida.
- Cuando el tiempo para atender una petición puede variar mucho. En estos casos, y con máquinas monoprocesadoras, las peticiones que requieran poco tiempo serán atendidas rápidamente, y las que requieran mucho tiempo sufrirán un breve retraso. Hay que tener en cuenta que el sistema operativo LINUX es multiproceso, de forma que todos los procesos reciben atencion.

En el servidor concurrente orientado a conexión tenemos un socket en un puerto conocido para atender a las peticiones de conexión de los clientes. Cuando llega una solicitud de conexión, el proceso padre, que escucha, crea un proceso hijo para atender dicha solicitud. El proceso hijo es quien realmente se conecta con el cliente. Mientras los clientes interactúan con los procesos servidores hijos, el proceso servidor padre escucha nuevas peticiones de servicio. Cada cliente puede intercambiar varias peticiones y respuestas con su proceso servidor. Los procesos servidores hijos terminan cuando lo hacen los clientes.

Un servidor concurrente orientado a la conexión se emplea en aplicaciones típicas donde el cliente establece una conexión con el servidor que dura un tiempo relativamente largo. Ejemplo: telnet, FTP, etc.

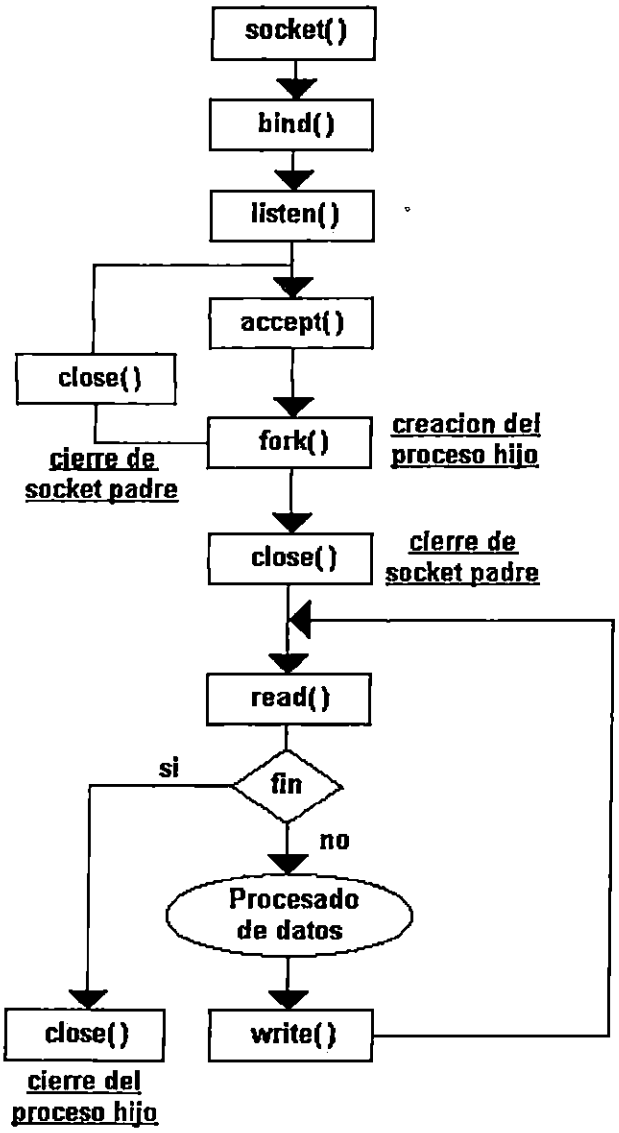


Figura 7: Servidor concurrente orientado a la conexión.

1.10.4. Servidor Concurrente No Orientado a Conexión

Este tipo de servidor no se utiliza mucho, ya que el algoritmo asociado a su implementación es ligermante más complicado. Su esquema de implementación se describe a continuación:

- Como siempre, se parte de un proceso servidor padre escuchando peticiones en un puerto conocido. En este caso no escucha peticiones de conexión, sino de servicio mediante `recvfrom()`.
- El cliente manda la primera petición en un datagrama UDP que contiene también su dirección IP y puerto, por lo que el proceso hijo servidor que se creará sabrá donde se encuentra el cliente.
- Tras esta primera petición, el proceso servidor padre crea un proceso servidor hijo, que toma un puerto local asignado por el sistema. Este proceso también debe cerrar la copia del socket del padre.
- El proceso servidor hijo elabora una respuesta a la primera petición del cliente, y le devuelve un datagrama UDP como respuesta, en donde figura la nueva dirección a donde el cliente debe dirigir las siguientes peticiones.
- El cliente y el proceso servidor hijo intercambiarán peticiones y respuestas. Mientras tanto, el proceso servidor padre, estará escuchando nuevas peticiones. Cuando el cliente termine, debe indicárselo al proceso servidor hijo para que este también lo haga.

A la hora de utilizar este modelo de servidor es bueno tener en cuenta que la creación de procesos es un procedimiento costoso en terminos computacionales. Hay que evaluar, en protocolos no orientados a conexión, si el costo de la concurrencia será mayor que lo que se gana en velocidad.

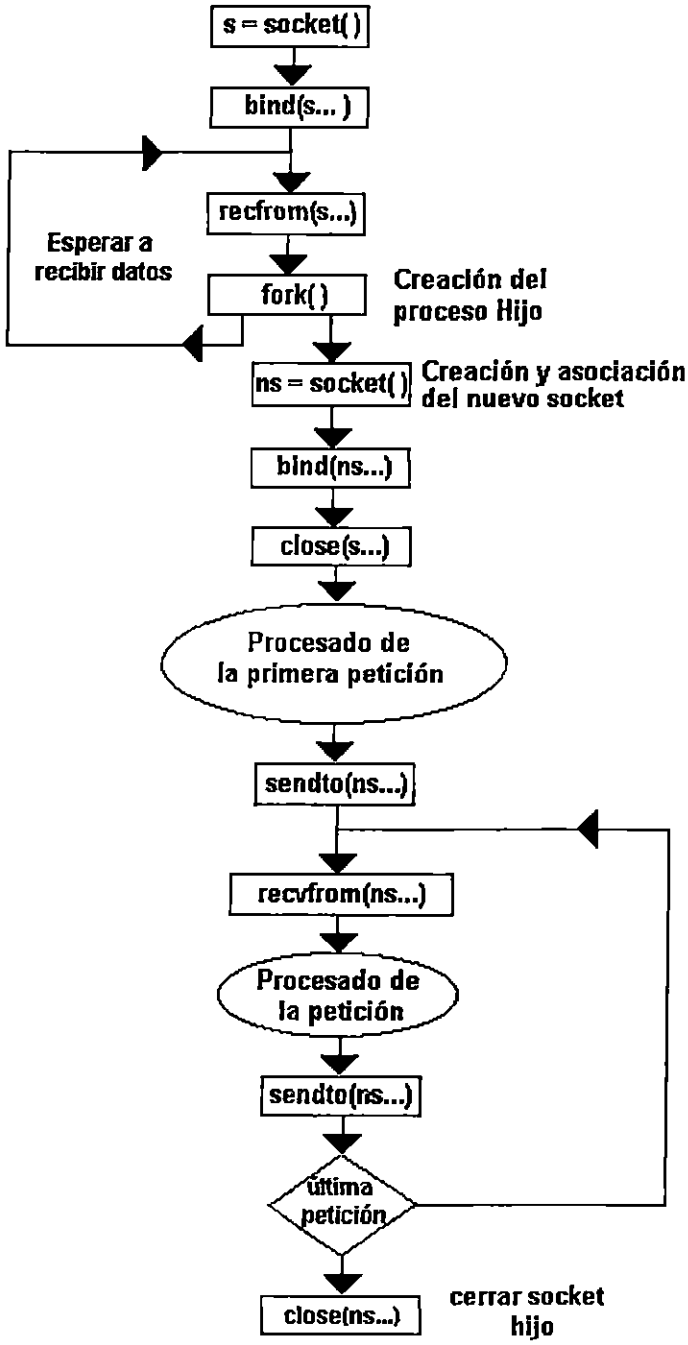


Figura 8: Servidor concurrente orientado a la NO conexión.

1.11. La Función fork

Para lograr que un proceso haga una copia de sí mismo utilizamos la función `fork`. El proceso que realiza la llamada es llamado *padre* y el nuevo proceso creado es llamado *hijo*. Esta función se describe en el archivo de cabecera `<unistd.h>` y su formato es:

```
pid_t fork (void)
```

Valor Devuelto: Cero en el proceso hijo, el número de identificador del proceso hijo (process ID) si se trata del proceso padre y -1 en caso de error.

Típicamente, hay 2 usos principales para `fork`:

1. Un proceso hace una copia de sí mismo para que dicha copia se dedique a una tarea específica. Este es el caso de los servidores de red.
2. Un proceso desea ejecutar otro programa. En este caso, el proceso llama a `fork` para hacer una copia de sí mismo y la copia llama a una de las funciones `exec`. Esto es común en programas como los *shells* (intérpretes de comandos).

1.12. Las Funciones exec

La única forma en que un archivo ejecutable sea echado a andar por UNIX o Linux es llamando a una de las funciones `exec`. Estas funciones reemplazan la imagen del proceso en curso con un nuevo programa. El ID de proceso no cambia. Dichas funciones se definen en el archivo de cabecera `<unistd.h>` y son:

```
int execl   (const char *camino, const char *arg,...);
int execlp  (const char *fichero, const char *arg,...);
int execle  (const char *camino, const char *arg ,..., char * const envp[]);
int execv   (const char *camino, char *const argv[]);
int execvp  (const char *fichero, char *const argv[]);
int execve  (const char *fichero, const char *argv [], const char *envp[])
```

Valor Devuelto: -1 en caso de error. Estas funciones no devuelven en caso de éxito.

camino: Especifica la ruta completa de ubicación del programa que se desea ejecutar, incluyendo el nombre de dicho programa.

fichero: Define únicamente el nombre del archivo a ejecutar. La ruta de búsqueda de dicho archivo será la definida por la variable de ambiente *PATH*. Si este argumento contiene el carácter de barra inclinada ("/") la variable de ambiente *PATH* es ignorada y el programa se ejecuta según la ruta especificada.

arg: Se refiere a una lista de argumentos separados por comas: arg0, arg1,... argn. Esta lista de argumentos no tiene una longitud definida, por lo que como último argumento DEBE terminar con un puntero nulo `((char*) 0` ó NULL).

argv: Indica un puntero a una cadena de argumentos.

envp: Define el entorno de proceso a ejecutar. Este es un puntero a cadenas de caracteres acabadas en cero y debe terminar con un puntero NULL.

Las primeras 5 funciones llaman a `execve`, por lo que se les considera solamente como interfazs de dicha función.

1.13. Funciones getsockname y getpeername

Estas funciones devuelven la dirección local de protocolo asociada con un socket (`getsockname`) o la dirección foránea de protocolo asociada con un socket (`getpeername`). Estas funciones se describen en el archivo de cabecera `<sys/socket.h>` y su formato es:

```
int getsockname      (int sfd, struct sockaddr *dp, int *long)
int getpeername      (int sfd, struct sockaddr *dp, int *long)
```

Valor Devuelto: 0 si la función tiene éxito. -1 si falla.

sfd: Descriptor del socket. Este es el número que devuelve la llamada a `socket`.

dp: En esta variable se devuelve la dirección de protocolo (que usualmente es llamada incorrectamente "nombre"), ya sea local o foránea asociada con el socket.

long: En este parámetro debe especificarse la cantidad de espacio en bytes reservada para `dp` (longitud)

Estas funciones suelen usarse siempre que se desconocen ya sea la dirección IP o el número de puerto asignado a un socket.

1.14. La Señal SIGCLD o SIGCHLD

Una señal (o interrupción se software) notifica a un proceso que ha ocurrido un evento. Las señales son impredecibles y asíncronas.

Un proceso UNIX, una vez finalizado pasa a ser un proceso *zombi*. Cuando un proceso hijo finaliza, pasa a estado zombi (dormido) mientras su padre no consulte su código de retorno. Si no se tiene cuidado en hacer que el padre consulte la finalización de su proceso hijo, la tabla de procesos termina por llenarse y los procesos zombi tarde o temprano llegan a saturar el kernel. Por esto se hace necesario climinar los procesos hijo cuando terminan.

Con la ayuda del comando `ps` (junto con la opción `j`)se pueden observar los procesos que se están ejecutando en un determinado momento. El campo `STAT` muestra el estado del proceso, `PID` muestra el identificador de proceso y `COMMAND` indica el proceso en cuestión.

```
> ps j
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
  560   568   568   568 pts/1      647 S        0   0:00 bash
  568   613   613   568 pts/1      647 TW       0   0:00 [less]
  568   618   618   568 pts/1      647 TW       0   0:00 [less]
  568   624   624   568 pts/1      647 T        0   0:00 less atalk.h
  568   647   647   568 pts/1      647 R        0   0:00 ps j
```

Las banderas posibles para `STAT` así como el respectivo significado se muestran a continuación:

R Ejecutándose
S Durmiendo
D Letargo Ininterruptible
T Detenido
Z Proceso Zombi
W El proceso no tiene página residente

En los sistemas basados en BSD, la solución consiste en capturar la señal SIGCLD que se envía al proceso padre cuando el proceso hijo termina. Esto se logra con la llamada a la función `signal`, (la cual se usa para instalar manejadores de señal) definida en el archivo de cabecera <`signal.h`> y cuyo formato es:

```
void (*signal(int signum, void (*manejador)(int)))(int)
```

Valor Devuelto: El número de manejador anterior de la señal o SIG_ERR en caso de error.

signum: Número que identifica a la señal cuyo manejador se instalará.

manejador: Puede ser una función definida por el usuario o alguna de las siguientes macros:

SIG_IGN:	Ignorar la señal.
SIG_DFL:	Restablecer el comportamiento por defecto de la señal. (Esta es la opción que se usa para finalizar los procesos zombie).

Alternativamente, suelen usarse funciones como `wait` y `waitpid` dentro de alguna otra función que haga las veces de manejador (en lugar de las macros SIG_IGN o SIG_DFL). Estas funciones se definen en el archivo de cabecera <`sys/wait.h`> y <`sys/types.h`> .Su formato es:

```
pid_t wait (int *estado)
pid_t waitpid (pid_t ID_proceso, int *estado, int opciones)
```

Valor Devuelto: Identificador del proceso el hijo que es terminado en caso de éxito, cero (si se usa `waitpid`, se define la función como no bloqueante o WNOHANG y al momento de ser llamada no hay hijos disponibles), o -1 si hay error (la variable `errno` se pone a un valor apropiado).

ID_proceso: Entero que identifica al proceso hijo cuya señal de terminación se desea capturar. Un valor de -1 especifica que se debe esperar por el primer proceso que termine (el mismo comportamiento que `wait`).

estado: Variable que después de llamada la función contiene el estado de terminación del proceso hijo.

opciones: Permite especificar varias opciones. Si no se desean especificar opciones, se le da el valor de cero (0). Dichas opciones son:

WNOHANG:	Devolver inmediatamente si ningún hijo ha terminado. Esta opción hace a la función no bloqueante.
WUNTRACED:	Devolver también si hay hijos detenidos y sobre los cuales no ha recibido notificación.

Estas funciones suspenden la ejecución de un proceso actual hasta que un proceso hijo ha terminado o hasta que se genera una señal cuya acción es terminar el proceso actual o llamar al manejador de la señal. Si al momento de llamar a estas funciones hay algún hijo en estado zombie, la función devuelve inmediatamente liberando todos los recursos reservados por el hijo.

Además del identificador de proceso, ambas funciones devuelven también el estado de terminación del proceso hijo por medio del puntero a `estado`.

Si no hay procesos hijos finalizados cuando se ejecuta el llamado a estas funciones pero hay procesos hijos en ejecución, tanto `wait` como `waitpid` (por defecto) se bloquean hasta que alguno de los procesos hijos en ejecución termine. `waitpid`. proporciona mayor control sobre cual proceso esperar y si la función debe o no ser bloqueante.

Un ejemplo de una función manejadora que eliminara a los procesos hijos convertidos en zombies sería:

```
#include <sys/wait.h>

pid_t mata_zombie ()
{     pid_t ID_proceso;
      int   estado;

      ID_proceso=wait(&estado);

      // La función devuelve el ID del proceso zombie que ha terminado.

      return ID_proceso;
}
```

Esta función termina un proceso zombie cuando es llamada, devolviendo el identificador (ID) de dicho proceso. Nótese que la variable `estado` solamente es creada para usarla como parámetro dentro de la llamada a `wait`. El valor que es devuelto al retornar la función `wait` nunca es usado y dicha variable es eliminada al salir de `mata_zombie`.

Sin embargo, esta función presenta la desventaja de ser bloqueante en el caso de que no haya procesos zombies en el momento de llamar a la función. Además, si hay varios zombies al momento de ejecutarse esta función manejadora, no podemos asegurar que todos los zombies serán terminados. Para corregir esto, hacemos uso de la función `waitpid`.

```
#include <sys/wait.h>

pid_t mata_zombies ()
{     int   estado;
      pid_t ID_proceso;

      do
      { ID_proceso=waitpid(-1, &estado, WNOHANG);}
      while (ID_proceso >0);
      return;
}
```

Como ya se mencionó, `waitpid` devuelve cero si se especifica la opción `WNOHANG` (la función no es bloqueante) y al momento de llamar la función no hay hijos disponibles, por lo que la función solamente retornará en caso de error o cuando haya acabado con todos los procesos hijos, no importa su número. El valor de -1 como primer argumento indica esperar por cualquier proceso hijo.

1.15. Multiplexado de Entrada-Salida

El multiplexado de E/S suele usarse en procesos que controlan varias fuentes de entrada o salida de datos. Dependiendo de las necesidades de la aplicación, suele usarse uno de los siguientes modelos para el multiplexado de E/S:

1.15.1. Modelo de Entrada-Salida Bloqueante

En este modelo, el proceso suele llamar a una función de E/S que permanece bloqueada hasta que la información que espera recibir le es entregada, es decir, se complete la operación de E/S, o hasta que se produzca un error. Por defecto, los sockets son bloqueantes. Un ejemplo de esto es la función `recvfrom`.

1.15.2. Modelo de Entrada-Salida No Bloqueante

Se refiere a funciones que no esperan a que se complete la operación de E/S para devolver un valor (si lo hacen). Cuando un socket es no bloqueante, se le dice que cuando ejecute una operación de E/S y esta no pueda ser completada inmediatamente, el proceso no debe ponerse a dormir, sino que en su lugar debe devolver un valor, el cual generalmente indica un error. Por lo general, este modelo requiere de una consulta constante a un proceso de E/S para verificar cuándo se completa satisfactoriamente una operación de este tipo, por lo que este modelo solamente se usa en sistemas dedicados a una función.

1.15.3. Modelo de Manejo de Señales de Entrada-Salida

Pueden usarse señales para indicar cuando hay datos válidos en algún punto de comunicación. Por supuesto, debemos instalar un manejador para la señal o señales que deseamos detectar, siendo estos manejadores los que se ocupan de la transferencia de información o bien le comunican a un proceso principal sobre el acontecimiento de algún evento y permiten que éste se haga cargo del envío o recepción.

1.15.4. Modelos de Entrada-Salida Asíncrono

En este modelo, le ordenamos al kernel iniciar una operación y notificar cuando ésta haya finalizado. La diferencia con el modelo anterior es que en el modelo de manejo de señal el kernel informa cuando un proceso inicia, mientras que con este modelo, indica cuando termina.

Es muy común ver que se usen los modelos bloqueante y de multiplexado en las aplicaciones de red que involucran E/S.

1.15.5. Modelo de Multiplexado de Entrada-Salida

Este modelo es similar al de E/S bloqueante en el sentido de que espera (se bloquea) hasta que haya datos válidos en un canal de E/S, pero a diferencia de este otro modelo, se desbloquea si recibe información valedera por cualquiera de sus puntos de comunicación (por cualquiera de sus sockets, en este caso).

La función `select` es la más usada cuando se usa el modelo de multiplexado de E/S, ya que permite decirle al kernel que espere por cualquiera de múltiples eventos, despertando al proceso solamente cuando uno de ellos ocurre o cuando haya transcurrido una cierta cantidad de tiempo. Esta función está descrita en los archivos de cabecera `<sys/select.h>` y `<sys/time.h>`. Su formato es:

```
int   select (int maxdesc, fd_set *desc_lectura, fd_set *desc_escritura,
              fd_set *desc_excepción, const struct timeval *t_limite)
```

Valor Devuelto: Número de descriptores listos, 0 si se sobrepasa el tiempo de espera y -1 en caso de error.

maxdesc: Especifica el número máximo de descriptores a ser evaluados. Su valor es el máximo descriptor a ser probado mas uno. Esto se debe a que la numeración de los descriptores inicia en cero (0). Usualmente, este valor no supera a 10.

desc_lectura, desc_escritura, desc_excepción: Especifican los descriptores que deseamos el kernel pruebe para verificar sus condiciones de lectura, escritura o excepción. Esta función solamente soporta 2 tipos de excepciones: La llegada de información fuera de banda al socket y la presencia de información de control de estado. Nótese que el tipo de variables de estos argumentos es `fd_set` (descriptor set) que se define como una cadena de enteros, en la cual cada bit en cada entero corresponde a un descriptor. Por ejemplo, si se usan enteros de 16 bits, el primer elemento de la cadena correspondería a los descriptores 0 al 15. El borrado o puesta de algún valor en este tipo de datos se hace por medio de las siguientes macros:

`void FD_ZERO (fd_set *desc_set);`	Pone todos los bits en `desc_set` a cero (0).

`void FD_SET (int desc, fd_set *desc_set):`	Activa el bit `desc` de la cadena `desc_set`.
`void FD_CLR (int desc, fd_set *desc_set):`	Desactiva el bit `desc` de la cadena `desc_set`.
`int FD_ISSET (int desc, fd_set *desc_set):`	Prueba si el bit `desc` de la cadena `desc_set` se encuentra activado.

Si no interesa cualquiera de estos 3 argumentos (`desc_lectura`, `desc_escritura` o `desc_excepción`), puede dárseles el valor de punteros nulos (NULL).

t_limite: Este argumento le dice al kernel cuanto tiempo esperar como máximo a que uno se los descriptores especificados se encuentre listo. Una estructura de tipo *timeval* define dicho tiempo en segundos y microsegundos.

```
struct timeval
{
long  tv_sec;     //segundos.
long  tv_usec;    //microsegundos.
};
```

Se pueden dar 3 casos posibles:

1- **Esperar por siempre.** En este caso el valor de este argumento es un puntero nulo (NULL).
2- **Esperar una cantidad fija de tiempo.** Esto dependerá de los valores dados a la estructura.
3- **No esperar.** En este caso se le da a ambos elementos de la estructura un valor de cero. Cuando se hace esto la función devuelve inmediatamente después de chequear los descriptores.

Para el caso de los sockets, se necesita saber cuándo uno de ellos se encuentra listo para una operación de E/S. Un socket se encuentra listo para una operación de lectura si se da alguno de los siguientes casos:

1. El número de bytes de datos en el buffer del socket receptor es mayor o igual que el tamaño mínimo definido para dicho socket.
2. La mitad de la conexión correspondiente a la lectura se encuentra cerrada. En este caso, una operación de lectura en el socket no se bloqueará y devolverá cero (0).
3. El número de conexiones completadas es diferente de cero. Esto para un socket de escucha (listening socket).
4. Un error de socket se encuentra pendiente. Una operación de lectura en el socket no se bloqueará y devolverá -1, dándole a la variable `errno` un valor apropiado.

Un socket está listo para escritura en los siguientes casos:

1. El número de bytes de espacio disponible en el buffer de transmisión del socket es mayor o igual al tamaño mínimo definido para ese socket (usualmente un valor de 2048 bytes para sockets TCP y UDP) y además:
   a) El socket está conectado
   b) El socket no requiere conexión.

2. La mitad de la conexión correspondiente a la escritura se encuentra cerrada.

3. Un error se encuentra pendiente en el socket. Una operación de escritura en el socket no se bloqueará y devolverá -1, dándole a la variable `errno` un valor apropiado.

Cuando un error ocurre en el socket, éste es marcado tanto listo para lectura como para escritura por la función `select`.

Como una alternativa a la función `select`, tenemos la función `poll`, la cual puede trabajar con descriptores de archivos o sockets. Esta función está definida en el archivo de cabecera `<sys/poll.h>` y cuyo formato es:

```
int poll(struct pollfd *descriptores, unsigned int num_desc, int t_lim)
```

Valor Devuelto: Número de descriptores para los que ha ocurrido un evento, 0 si ha transcurrido el tiempo límite de espera sin que ocurra ningún evento para los descriptores seleccionados y -1 en caso de error.

descriptores: Puntero a un vector (array) de estructuras tipo *pollfd*. Cada una representa un descriptor (un socket para este caso). Dichas estructuras se componen de la siguiente forma:

```
struct pollfd
{
      int fd;           /* Descriptor de fichero */
      short events;     /* Eventos solicitados */
      short revents;    /* Eventos ocurridos */
};
```

Las condiciones a ser probadas se especifican a través del miembro `events` y los resultados de la prueba son devueltos a través de `revents`. Estos 2 miembros especifican una condición específica. Dichas condiciones suelen ser especificadas a través de las siguientes constantes:

Constante	*Relacionada con:*	*Usada en:*	*Descripción*
POLLIN	Entrada	`events` y `revents`	Pueden ser leídos datos normales o con prioridad de banda
POLLRDNORM	Entrada	`events` y `revents`	Pueden ser leídos datos normales
POLLRDBAND	Entrada	`events` y `revents`	Pueden ser leídos datos con prioridad de banda
POLLPRI	Entrada	`events` y `revents`	Pueden ser leídos datos de alta prioridad
POLLOUT	Salida	`events` y `revents`	Pueden ser escritos datos con prioridad normal
POLLWRNORM	Salida	`events` y `revents`	Pueden ser escritos datos con prioridad normal
POLLWRBAND	Salida	`events` y `revents`	Pueden ser escritos datos con prioridad de banda
POLLERR	Errores	solamente `revents`	Ha ocurrido un error
POLLHUP	Errores	solamente `revents`	Se ha colgado la conexión
POLLNVAL	Errores	solamente `revents`	El descriptor (socket) no está abierto

Tabla 6: Constantes utilizadas para establecer o averiguar el valor de los argumentos `events` y `revents` respectivamente asociados con la función `poll`.

Hay 3 clases de datos identificados por la función `poll`: ***normales, prioridad de banda*** y ***de alta prioridad.***

num_desc: Especifica el número de descriptores que componen el vector `descriptores`.

t_limite: Especifica el tiempo máximo que la función esperará antes de retornar. Un valor negativo o la constante INFTIM hará que la función espere por siempre hasta que ocurra un evento o error. Un valor de cero (0) indica que la función debe retornar inmediatamente, sin bloquear y un valor positivo especificará el número de milisegundos que la función esperará como máximo antes de retornar.

Si ya no se está interesado en evaluar un descriptor en particular, se puede dar el valor de -1 al miembro `fd` de la estructura `pollfd`. Con esto el miembro `events` es ignorado y el miembro `revents` es puesto a cero.

1.16. Archivos de Configuración

Los archivos de configuración se encuentran en el directorio `/etc` y su estructura es casi la misma para todos. Cada línea es una entrada donde se relacionan un nombre y un número además de algún alias si lo hay. Dichos archivos son los siguientes:

- **/etc/hosts:** Contiene una tabla de correspondencias entre la dirección de internet (IP) y un nombre simbólico. Cada línea contiene la siguiente información:

  - Dirección IP
  - Nombre Oficial
  - Lista de alias.
  - Comentario.

  Un ejemplo de este archivo de configuración es el siguiente:

```
127.0.0.1       localhost.localdomain   localhost    # Dirección Loopback
131.107.15.1    nathan                  loghost      # Sitio nathan.net
```

- **/etc/networks:** Forma la base de datos de las redes conocidas. Cada línea representa una red con la siguiente información:

  - Nombre Oficial
  - Dirección IP.
  - Comentario.

- **/etc/protocols:** Contiene la lista de protocolos conocidos. Cada línea contiene la información siguiente:

  - Nombre Oficial
  - Número del Protocolo.
  - Lista de Alias.
  - Comentario.

  Un ejemplo del contenido de este archivo es el siguiente:

```
icmp        1     ICMP         # internet control message protocol
igmp        2     IGMP         # Internet Group Management
ggp         3     GGP          # gateway-gateway protocol
ipencap     4     IP-ENCAP     # IP encapsulated in IP (officially ``IP'')
st          5     ST           # ST datagram mode
tcp         6     TCP          # transmission control protocol
```

- **/etc/services:** Contiene una lista de los servicios Internet disponibles. Cada línea de este archivo contiene la información siguiente:

  - Nombre
  - Número de Puerto.
  - Protocolo.
  - Alias.
  - Comentarios.

  Un ejemplo del contenido de este archivo es el siguiente:

```
shell       514/tcp       cmd           # no passwords used
pop-2       109/tcp       postoffice    # POP version 2
sunrpc      111/tcp       portmapper    # RPC 4.0 portmapper TCP
omirr       808/udp       omirrd        # online mirror
```

1.16.1. Estructuras Relacionadas con Los Archivos de Configuración

Por lo general, los archivos de configuración descritos anteriormente son relacionados con estructuras hechas especialmente para accesar a dichos archivos. Estas estructuras se encuentran en el archivo de cabecera <netdb.h> y son las siguientes:

- La estructura `hostent` que consulta el archivo `/etc/hosts`

```
struct hostent
{
      char *h_name;          /* nombre oficial o "canónico" del anfitrión */
      char *h_aliases[];     /* lista de aliases terminada en NULL */
      int h_addrtype;        /* tipo dirección anfitrión */
      int h_length;          /* longitud de la dirección */
      char *h_addr_list[];   /* lista de direcciones IP terminada en NULL */
}

#define h_addr h_addr_list[0] /* por compatibilidad con versiones anteriores */
```

Los arrays de punteros siempre terminan con un puntero a NULL. La lista `h_addr_list` se compone de punteros a direcciones IP que pueden ser introducidas en estructuras tipo `in_addr`. En el caso de un ordenador con varios interfaces a distintas redes, puede usarse la lista de punteros a direcciones IP para conseguir la dirección de cada interfaz.

- La estructura `netent`, que consulta el archivo `/etc/networks`

```
struct netent
      {
         char *n_name;              /* nombre oficial de red */
         char **n_aliases;          /* lista de sinónimos */
         int n_addrtype;            /* tipo de dirección de red */
         unsigned long int n_net;   /* número de red */
      }
```

- La estructura `protoent` que consulta el archivo `/etc/protocols`.

```
struct protoent
      {
         char *p_name;        /* nombre oficial de protocolo */
         char **p_aliases;    /* lista de sinónimos */
         int p_proto;         /* número de protocolo */
      }
```

- La estructura `servent` que consulta el archivo `/etc/services`.

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

1.17. Conversiones Entre Nombres y Direcciones

Cuando Internet daba sus primeros pasos, no era raro que al comunicar una computadora con otra el usuario introdujera la dirección IP de aquel ordenador con quien deseaba entablar una comunicación; a veces incluso el número de puerto en que quería conectarse. Hoy día, sin embargo, un usuario generalmente define un nombre en lugar de una dirección IP y un número de puerto ya que los nombres son más fáciles de recordar y además se tiene la ventaja de que si en dado caso la dirección IP cambia, el nombre puede seguir siendo el mismo.

EL DNS o *Sistema de Nombres de Dominio* se usa para mapear entre nombres de anfitriones y direcciones IP. Un nombre de anfitrión (hostname) puede ser *simple* (como localhost, solaris, berkeley) o un *Nombre de Dominio Completamente Calificado* (*FQDN: Fully Qualified Domain Name,* conocido también como *nombre absoluto*),como www.icec.org.

Las funciones generalmente usadas para la conversión entre nombres y direcciones son `gethostbyname` y `gethostbyadress` para la conversión entre nombre y dirección IP, así como `getservbyname` y `getservbyport` para la conversión entre nombres de servicio y su respectivo número de puerto. Estas funciones hacen uso de estructuras tipo `hostent` (las 2 primeras) y `servent` (las 2 últimas). Tanto estas estructuras como las funciones se definen en el archivo de cabecera <`netdb.h`>. Dichas funciones se detallan a continuación:

- **gethostbyname:**

La función `gethostbyname` se encarga de realizar una consulta usando el nombre del anfitrión para devolver tantas direcciones IP se relacionen con dicho nombre (si las hay). Su formato es:

```
struct hostent *gethostbyname(const char *nombre)
```

Valor Devuelto: En caso de éxito, una estructura `hostent` que contiene todas las direcciones IP relacionadas con el nombre o un puntero nulo en caso de fracaso, junto con la variable `h_errno` puesta a un valor apropiado.

nombre: Se refiere al nombre del anfitrión para el cual se busca su dirección IP.

Esta función intenta consultar su archivo local `/etc/hosts`. Si no encuentra una dirección IP para el nombre dado, consulta al servidor de nombres (DNS) con el que el sistema tiene acceso (este depende del proveedor de Internet). Solamente si ambas búsquedas fallan devuelve error.

- **gethostbyaddr:**

Realiza el proceso opuesto a `gethostbyname`. Toma una dirección IP en formato numérico e intenta encontrar el nombre de anfitrión correspondiente a dicha dirección. Su formato es:

```
struct hostent *gethostbyaddr(const char *dirección, int longitud, int familia)
```

Valor Devuelto: En caso de éxito, una estructura `hostent` que contiene todas los nombres relacionados con la dirección IP o un puntero nulo en caso de fracaso, junto con la variable `h_errno` puesta a un valor apropiado.

dirección: Aunque se define como `char*`, es en realidad un puntero a una estructura tipo `in_addr` (para el caso de IPV4) conteniendo las direcciones IP.

longitud: Tamaño de la estructura definida en `dirección`.

familia: Especifica la *familia* o *dominio* del socket. Para IPV4 su valor es AF_INET.

El criterio de búsqueda para esta función es el mismo que para `gethostbyname`.

- **getservbyname:**

Los servicios, al igual que las direcciones IP son a veces conocidos por sus nombres. Esta función busca un número de servicio dado su nombre.

```
struct servent *getservbyname(const char *nombre, const char *protocolo)
```

Valor Devuelto: Estructura de tipo `servent` en caso de éxito, o un puntero NULL si ha ocurrido un error o se ha alcanzado el final del archivo de consulta `/etc/services`.

nombre: Se refiere al nombre del servicio.

protocolo: Nombre del protocolo. Este valor suele no ser especificado.

- **getservbyport:**

Realiza la función inversa a `getservbyname`. Busca el nombre de un servicio dado su número de puerto y, opcionalmente, su protocolo.

```
struct servent *getservbyport(int puerto, const char *protocolo)
```

Valor Devuelto: Estructura de tipo `servent` en caso de éxito, o un puntero NULL si ha ocurrido un error o se ha alcanzado el final del archivo de consulta `/etc/services`.

puerto: Se refiere al número de puerto para el cual se busca un nombre de servicio.

protocolo: Nombre del protocolo. Este valor suele no ser especificado.

Por otro lado, existen otras funciones, menos usadas, relacionadas con la conversión entre nombres y direcciones, las cuales también se definen en el archivo de cabecera `<netdb.h>` y son:

- **getnetbyname:**

Obtiene la dirección de una red dado su nombre. Su formato es:

```
struct netent *getnetbyname(const char *nombre)
```

Valor Devuelto: Estructura de tipo `netent` en caso de éxito, o un puntero NULL si ha ocurrido un error o se ha alcanzado el final del archivo de consulta `/etc/networks`.

nombre: Se refiere al nombre de la red.

- **getnetbyaddr:**

Obtiene los nombres de una red dada su dirección y tipo o familia.

```
struct netent *getnetbyaddr(long dirección, int familia)
```

Valor Devuelto: Estructura de tipo `netent` en caso de éxito, o un puntero NULL si ha ocurrido un error o se ha alcanzado el final del archivo de consulta `/etc/networks`.

dirección: la dirección IP de la red cuya información se desea obtener.

familia: Especifica la *familia* o *dominio* del socket. Para IPV4 su valor es AF_INET.

- **getprotobyname:**

Esta función se emplea para obtener un número de puerto basándose en el nombre del servicio relacionado con dicho puerto.

```
struct protoent *getprotobyname(const char *nombre)
```

Valor Devuelto: Estructura de tipo `protoent` en caso de éxito, o un puntero NULL si ha ocurrido un error o se ha alcanzado el final del archivo de consulta `/etc/protocols`.

nombre: Se refiere al nombre del servicio.

- **getprotobynumber:**

Esta función se emplea para obtener el nombre del servicio de un puerto basándose en el número de puerto respectivo.

```
struct protoent *getprotobynumber(int puerto)
```

Valor Devuelto: estructura de tipo `protoent` en caso de éxito, o un puntero NULL si ha ocurrido un error o se ha alcanzado el final del archivo de consulta `/etc/protocols`.

puerto: Se refiere al número de puerto para el que se busca el nombre del servicio relacionado.

Existen 2 funciones ampliamente usadas para obtener el nombre y dirección de la máquina local. Dichas funciones son:

- **gethostname:**

obtiene el nombre de la máquina anfitrión (host) o máquina local. Esta función se define en el archivo de cabecera `<unistd.h>`. Su formato es:

```
int gethostname(char *nombre, int longitud)
```

Valor Devuelto: 0 si la llamada a la función tiene éxito y 1- en caso de error, dándole a la variable `errno` el valor apropiado

nombre: Puntero al nombre del anfitrión.

longitud: Tamaño en bytes del nombre del anfitrión, es decir, longitud de `nombre`.

- **uname:**

Se usa para obtener información del sistema anfitrión. Suele usarse junto con `gethostbyname` para obtener la dirección IP del anfitrión. Esta función hace uso de una estructura llamada `utsname`. Tanto dicha estructura como la función se encuentran en el archivo de cabecera `<sys/utsname.h>`. La estructura `utsname` se compone de los elementos siguientes:

```
struct utsname
{
        char sysname[SYS_NMLN];      // Nombre del sistema operativo.
        char nodename[SYS_NMLN];     // Nombre del nodo
        char release[SYS_NMLN];      // Número de publicación (release)
        char version[SYS_NMLN];      // Versión del sistema operativo.
        char machine[SYS_NMLN];      // Tipo de hardware.
        char domainname[SYS_NMLN];   // Nombre del dominio.
};
```

La función uname tiene el formato siguiente:

```
int uname(struct utsname *nombre)
```

Valor Devuelto: 0 si la llamada a la función tiene éxito y 1- en caso de error, dándole a la variable errno el valor apropiado

nombre: Puntero a estructura utsname en la que se devolverá la información correspondiente al anfitrión..

1.18. Demonización

En los sistemas Linux y UNIX, un *demonio* es un proceso que se ejecuta en segundo plano, siendo independiente de cualquier terminal, lo que le permite, en caso de haber sido creado desde una terminal, seguir existiendo aún cuando la terminal que los creó sea terminada o usada para otra tarea.

La forma más fácil de enviar un proceso a segundo plano es agregando el carácter de *ampersand* (&) al final del comando a ejecutar. Sin embargo, para un servidor, lo ideal es hacer que los programas encargados de atender clientes se pongan a sí mismos en segundo plano y se vuelvan independientes de cualquier terminal.

Hay muchas formas de iniciar un demonio:

1. Durante el arranque del sistema, muchos demonios son iniciados por los scripts de inicialización, a menudo en el directorio /etc. Estos demonios inician con privilegios de superusuario (root). Este es el caso del demonio inetd y cron.
2. Por medio del demonio inetd y su archivo de configuración correspondiente.
3. Por medio del demonio cron.
4. Por medio del comando at.
5. A través de terminales de usuario.

Ya que los demonios no tienen terminal de control, necesitan de una forma para enviar mensajes cuando ocurra algo importante. Para ello se usa la función syslog, la cual manda dichos mensajes al demonio syslogd.

El demonio syslogd tiene asociado un archivo de configuración: /etc/syslog.conf, el cual define qué hacer con cada tipo de mensaje que le llega al demonio. Generalmente el tratamiento que se les da es el ser agregados a un archivo, ser escritos a un usuario en particular o enviados a otro demonio syslogd en otro anfitrión.

Al inicializar este demonio, suele asociarse un socket UNIX al archivo /var/run/log. Dicho socket, de tipo UDP, es asociado al puerto (reservado para el servicio syslog). Finalmente se abre /dev/klog. Cualquier mensaje de error en el kernel aparecerá como entrada a este dispositivo.

La función syslog se define en el archivo de cabecera <syslog.h> y su formato es:

```
void syslog( int prioridad, char *mensaje,...)
```

prioridad: Es una combinación de *nivel* y *facilidad*, generalmente unidos por un operando OR lógico ("|"). Dichos valores se describen a continuación:

Nivel	*Valor*	*Descripción*
LOG_EMERG	0	Sistema inservible (prioridad máxima)
LOG_ALERT	1	Debe tomarse acción inmediatamente
LOG_CRIT	2	Condición crítica
LOG_ERR	3	Condición de error
LOG_WARNING	4	Advertencia
LOG_NOTICE	5	Condición normal pero significativa (valor por defecto)
LOG_INFO	6	Información
LOG_DEBUG	7	Mensajes de depuración (prioridad mínima)

***Tabla 7:** Posibles valores que puede tomar la variable `nivel`.*

La `facilidad` identifica el tipo de proceso que envía el mensaje. Sus posibles valores son:

Facilidad	*Descripción*
LOG_AUTH	Mensajes de seguridad/autorización
LOG_AUTHPRIV	Mensajes de seguridad/autorización (privados)
LOG_CRON	Demonio `cron`
LOG_DAEMON	Demonios del sistema
LOG_FTP	Demonio FTP
LOG_KERN	Mensajes del kernel
LOG_LOCAL0 a LOG_LOCAL7	Uso local
LOG_LPR	Sistema de impresión
LOG_MAIL	Sistema de correo
LOG_NEWS	Sistema de noticias en red
LOG_SYSLOG	Mensajes generados internamente por `syslog`
LOG_USER	Mensajes de nivel de usuario aleatorios (valor por defecto)
LOG_UUCP	Sistema UUCP

***Tabla 8:** Posibles valores de la variable `facilidad`.*

El propósito de *nivel* y *facilidad* es permitir que los mensajes de cierto nivel de prioridad sean manejados de igual forma entre sí.

mensaje: Es el mensaje que se desea enviar al demonio `syslogd` con formato similar al de `printf`. Generalmente incluye la especificación `%m`, la cual es reemplazada por el mensaje de error correspondiente al valor de la variable `errno`.

Cuando una aplicación llama a `syslog` la primera vez, crea un socket datagrama en el dominio UNIX y llama a `connect` asociando el socket con la ruta del socket asociada por medio del demonio `syslogd`. Este socket permanece abierto hasta que el proceso termina. Un proceso puede ser abierto o cerrado por medio de las funciones `openlog` y `closelog`, definidas en el archivo de cabecera `<syslog.h>` y cuyo formato es:

```
void openlog     (const char *ident, int opciones, int facilidad)
void closelog    (void)
```

ident: Usualmente el nombre del programa.

opciones: Se compone de la combinación (a través del operador lógico OR) de una o más de las siguientes constantes:

LOG_CONS:	Enviar mensajes a consola si no puede enviarlas al demonio `syslogd`.
LOG_NDELAY:	No retardar apertura, crear socket ahora.
LOG_PERROR:	Conectarse a la terminal *error estándar* (`stderr`) y también enviar mensajes al demonio `syslogd`.
LOG_PID:	Incluir el ID de proceso con cada mensaje.

Una forma sencilla de *"demonizar"* un proceso es por medio del demonio `inetd`. Puede ser usado por servidores TCP y UDP, pero no puede manejar otros protocolos (como sockets UNIX). El proceso `inetd` se establece a sí mismo como demonio, para luego leer y procesar su archivo de configuración (generalmente `/etc/inetd.conf`), el cual especifica los servicios a manejar y qué hacer cuando una solicitud de servicio llegue. Este archivo de configuración se compone de los siguientes campos:

Nombre del servicio:	Debe aparecer en `/etc/services`.
Tipo de socket:	TCP (stream) o UDP (dgram).
Protocolo:	Debe aparecer en `/etc/protocols`: puede tomar los valores `udp` o `tcp`.
Bandera de espera:	Normalmente vale `nowait` para TCP y `wait` para UDP.
Login o nombre de usuario:	definido en el archivo `/etc/passwd`. Típicamente `root`.
Programa servidor:	Ruta completa del programa a ser ejecutado.
Argumentos del programa servidor:	Argumentos a ser usados por el programa servidor.

Un ejemplo de las líneas que contiene este archivo de configuración es el siguiente:

```
auth      stream   tcp    wait      root     /usr/sbin/in.identd   in.identd -e \o
bootp     dgram    udp    wait      root     /usr/sbin/tcpd        bootpd
ftp       stream   tcp    nowait    root     /usr/sbin/tcpd        in.ftpd -l -a
```

En vista de lo anterior, basta con incluir una línea similar a las mostradas en el archivo `/etc/inetd.conf` para hacer que un programa sea demonizado, por supuesto, modificando nuestro programa con las adaptaciones pertinentes.

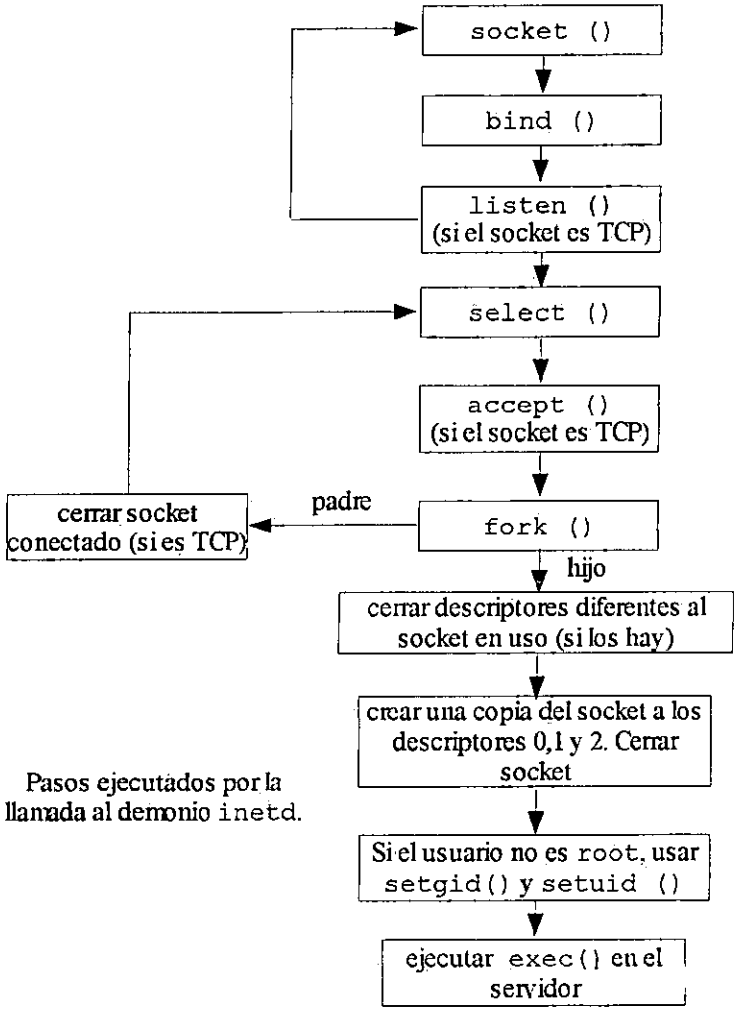


Figura 9: Pasos que sigue un proceso para su demonización a través del demonio `inetd`.

Los pasos seguidos por la llamada al demonio `inetd` son los siguientes:

1. Durante la inicialización `inetd` lee el archivo `/etc/inetd.conf` y crea un socket del tipo apropiado (`stream` o `dgram`) para el servicio especificado en el archivo.

2. `bind` es llamado para el socket, especificando el puerto para el servidor y la dirección IP comodín (INADDR_ANY). Este número de puerto TCP o UDP es obtenido llamando a la función `getservbyname` con los campos *nombre de servicio* y *protocolo* del archivo de configuración como argumentos.

3. Para sockets TCP, `listen` es llamado para que las peticiones de conexiones que lleguen sean aceptadas. Este paso no es realizado para sockets UDP.

4. Después de que todos los sockets son creados, `select` es llamado para esperar por cualquiera de los sockets que puedan ser leídos.

5. Cuando `select` indica que un socket está disponible para ser leído, si el socket es TCP, `accept` es llamado para aceptar una nueva conexión.

6. `inetd` crea un proceso hijo por medio de `fork` para manejar la solicitud de servicio, similarmente a un servidor concurrente estándar.

El hijo cierra todos los descriptores menos el descriptor del socket que esta utilizando: El nuevo socket conectado retornado por `accept` para un servidor TCP o el socket servidor UDP.

El proceso hijo llama a la función `dup2` 3 veces, duplicando el socket en los descriptores 0 (entrada estándar), 1 (salida estándar) y 2 (error estándar). El descriptor de socket original es cerrado.

Posterior a esto, el hijo llama a `getpwnam` para saber si se está operando como superusuario (root). Si no es así, llama a las funciones `setgid` y `setuid` para convertirse en superusuario.

7. Si el socket es tipo *stream* (flujo de bits), el proceso padre debe cerrar el socket conectado (como en el servidor concurrente). El padre llama nuevamente a `select`, aguardando a que un nuevo socket pueda ser leído.

El demonio `inetd` se encarga de la creación del respectivo socket para la comunicación, por lo que no es necesario que el código respectivo sea incluido en el programa del usuario. Además, tampoco es necesario la creación de lazos infinitos para estar atendiendo las peticiones de conexión, ya que el proceso se invoca una vez por conexión de cliente.

1.19. Opciones de Socket

Las opciones de socket se usan para modificar ciertas características propias de los mismos, las cuales pueden estar a diferentes niveles de protocolo. Las principales maneras de obtener y establecer las opciones relacionadas con sockets son:

- Las funciones `getsockopt` (para obtener opciones de socket) y `setsockopt` (para establecer las opciones de socket).
- La función `fcntl`
- La función `ioctl` (entrando en desuso)

1.19.1. Las Funciones getsockopt y setsockopt

Estas funciones se encuentran definidas en los archivos de cabecera `<sys/socket.h>` y `<sys/types.h>`. Su formato es:

```
int getsockopt(int s, int nivel, int nomopc, void *valopc, int *lonopc)
int setsockopt(int s, int nivel, int nomopc, const void *valopc, int lonopc)
```

Valor Devuelto: 0 Si la llamada a la función se realiza con éxito y -1 en caso de error, tomando la variable `errno` el valor apropiado.

s: Se refiere al descriptor asociado con un socket abierto. Este valor suele ser el devuelto por la llamada a la función `socket`.

nivel: Especifica el código con el cual interpretar la opción. Los valores que suele tomar este parámetro son los siguientes (estas y otras constantes están definidas en los archivos `<sys/socket.h>` e `<in.h>`):

`SOL_SOCKET`:	Para modificar opciones a nivel de capa 5 o nivel de sesión (sockets).
`IPPROTO_TCP`:	Para modificar opciones a nivel de capa 4 o nivel de transporte (TCP).
`IPPROTO_UDP`:	Para modificar opciones a nivel de capa 4 o nivel de transporte (UDP).
`IPPROTO_IP`:	Para modificar opciones a nivel de capa 3 o nivel de red (IP).
`IPPROTO_ICMP`:	Para modificar opciones a nivel de capa 3 o nivel de red (ICMP).

nomopc: Nombre de la opción.

valopc: Puntero que indica el valor de la opción. En `getsockopt` este es el resultado que la función devuelve al retornar (argumento valor-resultado). En `setsockopt` es un parámetro que debe ser introducido.

lonopc: Longitud del campo `valopc`.

Existen 2 tipos básicos de opciones:

BANDERAS: Son aquellas que habilitan o deshabilitan una determinada característica (sólo poseen 2 estados).
VALORES: Son aquellas que pueden buscar y devolver valores específicos que pueden ser establecidos o examinados.

La siguiente tabla resume algunas de las opciones de socket más usadas y que pueden ser usadas por `getsockopt` o establecidas por `setsockopt`.

a) Nivel de socket (SOL_SOCKET):

Nombre Opción	*get*	*set*	*Descripción*	*Bandera*	*Tipo de Dato*
SO_BROADCAST	X	X	Permite el enviado o multidifusón de datagramas (Broadcast)	X	int
SO_DEBUG	X	X	Habilita el trazado para depuración	X	int
SO_DONTROUTE	X	X	Ignora la tabla de búsqueda de ruteo.	X	int
SO_ERROR	X		Obtiene error pendiente y limpia		int
SO_KEEPALIVE	X	X	Prueba periódicamente si la línea sigue "viva".	X	int
SO_LINGER	X	X	Demora el cerrado de la conexión si hay datos que enviar.		linger
SO_OOBINLINE	X	X	Deja los datos fuera de banda en línea	X	int
SO_RCVBUF	X	X	Tamaño de buffer de recepción		int
SO_SNDBUF	X	X	Tamaño de buffer de transmisión		int
SO_RCVLOWAT	X	X	Marca de agua para el buffer de recepción (tamaño mínimo)		int
SO_SNDLOWAT	X	X	Marca de agua para el buffer de transmisión (tamaño mínimo)		int
SO_RCVTIMEO	X	X	Tiempo de espera máximo para recepción		timeval
SO_SNDTIMEO	X	X	Tiempo de espera máximo para transmisión		timeval
SO_REUSEADDR	X	X	Permitir el reuso de la dirección local	X	int
SO_REUSEPORT	X	X	Permitir el reuso del puerto local	X	int
SO_TYPE	X		Obtener el tipo de socket		int

***Tabla 9:** Opciones de socket más comunes a nivel de socket.*

b) Nivel TCP (IPPROTO_TCP):

Nombre Opción	*get*	*set*	*Descripción*	*Bandera*	*Tipo de Dato*
TCP_KEEPALIVE	X	X	Tiempo de ocio en segundos antes de probar si la conexión sigue viva.		int
TCP_MAXRT	X	X	Tiempo máximo de retransmisión para TCP.		int
TCP_MAXSEG	X	X	Tamaño máximo de segmento para TCP.		int
TCP_NODELAY	X	X	Deshabilitar el algoritmo de Nagle.	X	int

***Tabla 10:** Opciones de socket más comunes a nivel de TCP.*

c) Nivel IP (IPPROTO_IP):

Nombre Opción	*get*	*set*	*Descripción*	*Bandera*	*Tipo de Dato*
IP_HDRINCL	X	X	Incluir encabezado IP con los datos	X	int
IP_OPTIONS	X	X	Opciones de cabecera IP		
IP_RECVDSTADDR	X	X	Devolver dirección IP destino	X	int
IP_TOS	X	X	Tipo de servicio y precedencia		int
IP_TTL	X	X	Tiempo de vida		int

***Tabla 11:** Opciones de socket más comunes a nivel de IP.*

SO_BROADCAST: Esta opción habilita o deshabilita la capacidad del proceso para enviar mensajes de "broadcast". Este tipo de mensajes sólo está soportado por sockets de tipo datagramas (IP) y redes que soportan este servicio.

SO_DEBUG: Esta opción es soportada por TCP solamente. Cuando se habilita, el kernel guarda un registro con información detallada de los paquetes enviados o recibidos por TCP a través del socket.

SO_DONTROUTE: Especifica que los paquetes enviados ignorarán los mecanismos de enrutado normales del protocolo de capa inferior. Esta opción es usada generalmente por demonios de enrutamiento para forzar un paquete a que pase por una determinada interfaz.

SO_ERROR: Esta opción solamente puede ser consultada. Cuando ocurre un error en un socket dentro de un sistema basado en el núcleo de Berkeley da cierto valor a la variable `so_error`, según los valores exxx de UNIX estándar. Por medio de a consulta a esta opción podemos conocer el valor de `so_error`.

SO_KEEPALIVE: Se usa con sockets TCP. Cuando esta opción está activada y no ha habido intercambio de datos a través del socket en ninguna dirección por 2 horas, TCP envía automáticamente una prueba a la otra terminal. Esta prueba es un segmento TCP que debe ser respondido. Si no hay respuesta de ningún tipo, se mandarán 8 mensajes de prueba más separados por 75 segundos hasta recibir respuesta o que transcurran 11 minutos y 15 segundos después de enviado el primer mensaje de prueba. Si para ese entonces no hay respuesta, la conexión se cierra. Si la otra terminal responde con una señal RST, la conexión también es cerrada.

SO_LINGER: Especifica el modo en que opera la función `close` para protocolos orientados al la conexión (como TCP). El comportamiento por defecto para la llamada a esta función es el cerrado inmediato del socket mientras trata de enviar los datos que han quedado en el buffer de transmisión.
Para modificar este comportamiento, necesitamos de una estructura como la siguiente, definida en el archivo de cabecera `<sys/socket.h>`:

```
struct      linger
{     int   l_onoff;    // 0= desactivado, otro valor= activado.
      int   l_linger;   // tiempo de demora, generalmente en segundos.
};
```

Según los valores de esta estructura, pueden darse los siguientes casos:
1. Si `l_onof=0`, la opción se desactiva, el valor de `l_linger` es ignorado y se aplica el comportamiento por defecto, descrito anteriormente.
2. si `l_onoff` no es cero y `l_linger=0`, se aborta la conexión cuando se cierra, es decir, se descarta cualquier dato en el buffer de transmisión del socket y se envía una señal RST a la otra terminal en lugar de la secuencia normal de terminación.

3. Si `l_onoff` no es cero ni tampoco `l_linger`, el kernel revisará si hay datos en el buffer de recepción al momento de cerrar el socket. Si los hay, el proceso es puesto a dormir hasta que todos los datos en dicho buffer son enviados y aceptados por el receptor o hasta que el tiempo de espera (ocio) termine. Si el socket se ha definido como no bloqueante, no esperará a que se complete la llamada a `close`, sin importar el valor de `l_linger`.

SO_OOBINLINE: Esta opción especifica si los datos *"fuera de banda"* serán colocados en la cola de entrada normal. Cuando esto ocurre, la bandera MSG_OOB de las funciones de recepción no puede ser usada para leer los datos *"fuera de banda"*.

SO_RCVBUF y SO_SNDBUF: Permiten consultar o definir el tamaño de los búferes de transmisión o recepción. Estos valores varían según las implementaciones aunque su valor típico es 8192 bytes. Es necesario que si estos parámetros son modificados, esto se haga antes de establecer la conexión. Se recomiendan tamaños de buffer grandes para conexiones rápidas.

SO_RCVLOWAT y SO_SNDLOWAT: Estas opciones permiten cambiar los valores de las *"marcas de agua inferiores"* de transmisión y recepción. Estos valores son usados por la función `select`. Dichos valores determinan, en el caso de la recepción, la cantidad mínima de datos que debe haber el buffer del socket de recepción para que la función `select` marque dicho socket como "apto para lectura". Su valor por defecto es 1 para sockets TCP y UDP. En el caso del buffer de recepción, indica indica la cantidad mínima de espacio disponible que debe existir en el buffer del socket de transmisión para que la función `select` lo marque como "apto para escritura". Su valor por defecto es de 2048 bytes para sockets TCP.

SO_RCVTIMEO y SO_SNDTIMEO: Estas opciones permiten establecer un límite de tiempo para las funciones de transmisión y recepción de un socket. El argumento usado por estas opciones es un estructura tipo `timeval`, por lo que los límites de tiempo pueden especificarse en segundos y/o milisegundos. Estas opciones afectan las funciones `read`, `readv`, `recv`, `recvfrom` y `recvmsg`, `write`, `writev`, `send`, `sendto` y `sendmsg`[8]. Cuando todos los valores de la estructura `timeval` que comprende esta opción están a cero, dicha opción es desactivada.

SO_REUSEADDR: La opción SO_REUSEADDR (reusar dirección) tiene 4 propósitos básicos:

1. Permite a un servidor en escucha de peticiones inicializar y enlazar (bind) uno de sus puertos aunque ya tenga una conexión establecida previamente usando dicho puerto.
2. Permite que múltiples copias del mismo proceso servidor sean iniciadas en el mismo puerto, cada una enlazándose a una dirección IP diferente. Esto es particularmente útil en servidores que poseen uno o más alias de su dirección IP.
3. Permite a un solo proceso enlazar un mismo puerto a múltiples sockets, siempre y cuando cada uno especifique una dirección IP diferente.

Cuando se usa esta opción, esta debe establecerse antes de las llamadas a `bind` en sockets TCP.

SO_REUSEPORT: Esta opción se introdujo junto con el soporte para multidifusión para las distribuciones 4.4BSD. Por esto, no todos los sistemas la soportan actualmente. Esta opción es útil cuando se desea realizar enlaces completamente duplicados (mismo puerto y dirección IP). Este es el caso de la multidifusión. Para lograr lo anterior, todos los sockets involucrados deberán activar esta opción.

SO_TYPE: Esta opción solamente puede ser consultada. Devuelve el tipo de socket. El valor devuelto generalmente corresponde a las constantes SOCK_STREAM (TCP) o SOCK_DGRAM (UDP).

8 Para mayor información sobre estas funciones, consultar las respectivas páginas man.

b) Nivel TCP (IPPROTO_TCP):

TCP_KEEPALIVE: Esta opción aparece por primera vez con Posix 1.g, por lo que aún no es ampliamente implementada. Especifica el tiempo de espera en segundos para una conexión TCP antes de que empiecen a enviarse pruebas para averiguar si la conexión "sigue viva". Su valor por defecto es 7,200 seg (2 horas). Esta opción es efectiva únicamente cuando se habilita la opción SO_KEEPALIVE.

TCP_MAXRT: Esta opción implementada por primera vez con Posix 1.g y especifica la cantidad de tiempo en segundos antes de que una conexión se rompa una vez TCP inicia retransmisión de datos. Un valor de cero indica usar los valores por defecto del sistema y de -1 el retransmitir por siempre. Si se especifica un entero positivo, este se redondea al siguiente tiempo de retransmisión de la implementación.

TCP_MAXSEG: Permite consultar o definir el *Tamaño Máximo de Segmento* (*MSS: Maximum Segment Size*) para una conexión TCP. Cuando se consulta, devuelve la máxima cantidad de datos que podrán ser enviados al otro extremo.

TCP_NODELAY: Si se activa esta opción, se desactiva el *algoritmo de Nagle* para TCP. Por defecto, dicho algoritmo es usado. El propósito de dicho algoritmo es el de reducir el número de paquetes pequeños (inferiores al tamaño máximo de segmento) en una WAN. Básicamente, el algoritmo especifica que si una conexión tiene datos en tránsito (han sido enviados, pero aún no se recibe su aceptación por el receptor), entonces no se enviarán paquetes pequeños hasta recibir la respectiva señal de reconocimiento por el receptor.

c) Nivel IP (IPPROTO_IP):

IP_HDRINCL: Esta opción suele usarse con sockets "crudos" (raw sockets). Cuando está activada, el programador debe tomar la responsabilidad de construir la respectiva cabecera IP para todos los datagramas enviados. Cuando esta opción no se activa es el kernel quien realiza esta función. Esta opción no cubre los siguientes casos:

-) IP siempre calcula y guarda la suma de verificación de cabecera (checksum).
-) Si se establece el campo de identificación con valor de cero, el núcleo reasigna el valor del campo.
-) Si la dirección de IP de la fuente es INADDR_ANY, se le da el valor de la dirección IP primaria de la interfaz de salida.
-) El cómo establecer las opciones IP depende de la implementación.

IP_OPTIONS: Esta opción permite establecer las opciones para la cabecera IPV4. Esto requiere un sólido conocimiento del formato de las opciones IP en la respectiva cabecera.

IP_RECVDSTADDR: Esta opción permite que la dirección IP de destino para un datagrama UDP sea devuelto como datos suplementarios por `recvmsg`.

IP_TOS: Permite definir el campo referente al *tipo de servicio* de la cabecera IP. Los posibles valores para esta opción aparecen en el archivo de cabecera `<netinet/ip.h>` y son:

Constante	*Descripción*
IPTOS_LOWDELAY	Minimizar retardo
IPTOS_THROUGHPUT	Maximizar desempeño
IPTOS_RELIABILITY	maximizar confiabilidad
IPTOS_LOWCOST	Minimizar el costo.

***Tabla 12:** Posibles valores que acompañan a la opción IP_TOS.*

IP_TTL: Con esta opción, se puede establecer o consultar el tiempo de vida del datagrama (TTL).

1.19.2. La Función fcntl

Esta función , que significa "control de archivos" (file control) realiza varias operaciones para control de descriptores. Es una de las formas preferidas en el estándar Posix 1.g que se usan para el control de descriptores. Esta función provee, entre otras, las siguientes operaciones relacionadas con un entorno de red:

- E/S no bloqueante.
- Manejo de señales de E/S.
- Establecer u obtener el dueño del socket.

Dicha función se define en los archivos de cabecera `<unistd.h>` y `<fcntl.h>`. Su formato es:

```
int    fcntl(int fd, int cmd)
int    fcntl(int fd, int cmd, long arg)
```

Valor devuelto: Depende del parámetro `cmd` y -1 en caso de error.

fd: Descriptor de archivo (o socket para este caso).

cmd: Comando a procesar. Este parámetro puede tomar uno de los siguientes valores:

F_GETFL: Leer las banderas del descriptor.

F_SETFL: Asigna las banderas del descriptor según el valor asignado por `arg`. Se recomienda establecer una bandera específica por medio del operador OR lógico (||) para no alterar las banderas ya establecidas. Dentro de la programación de redes, las banderas más comúnmente usadas junto con este comando son:

O_NONBLOCK: Modo de E/S NO bloqueante.
O_ASYNC: Notificación de manejo de señales de E/S.

F_GETOWN: Obtiene el ID de proceso o el grupo de procesos que recibe las señales SIGIO y SIGURG para los eventos sobre el descriptor de fichero `fd`. Un socket creado con la función `socket` no tiene dueño por defecto, por lo que se usa esta función para asignarle uno.

F_SETOWN: Establece el ID de proceso o el grupo de procesos que recibirá las señales SIGIO y SIGURG para los eventos sobre el descriptor de fichero `fd`.

Los grupos de procesos se especifican mediante valores negativos.

Debe aclararse que estos valores no constituyen los únicos que puede tomar este parámetro; sin embargo son los más usados dentro de la programación de redes.

1.20. Sockets "Crudos" (Raw Sockets)

Los sockets "crudos" proporcionan 3 características que no se encuentran en los sockets TCP o UDP.

1. Permiten leer y escribir paquetes ICMPv4, IGMPv4 e ICMPv6. Esto posibilita la construcción de aplicaciones que manejen mensajes ICMP e IGMP manejadas completamente por el usuario.

2. Es posible para un proceso leer y escribir datagramas IPV4 con un campo de protocolo que el kernel no necesariamente soporta. la mayoría de versiones de kernel solamente soportan los códigos 1 (ICMP), 2 (IGMP), 6 (TCP) y 17 (UDP). Estos valores son los más comunes, pero no los únicos.
3. Es posible construir una cabecera IPV4 según el deseo del programador usando la opción de socket IP_HDRINCL.

Para la creación de un socket "crudo" se debe definir como segundo argumento de la función `socket` (el argumento `tipo`) la constante SOCK_RAW. a diferencia de los otros tipos de socket descritos anteriormente, el tercer argumento (conocido como `protocolo`) suele tener un valor diferente de cero. Los valores que toma están definidos en el archivo de cabecera `<netinet/in.h>`. Debe hacerse énfasis de que el simple hecho de que un protocolo se encuentre definido en dicho archivo de cabecera no significa que el kernel lo soporte.

Únicamente alguien con los privilegios del superusuario puede crear sockets "crudos". Esto es para evitar que cualquier usuario mande sus propios datagramas a la red.

A pesar de que es posible hacer llamadas a las funciones `bind` y `connect` con sockets "crudos", este es un caso muy poco común, ya que no existe el concepto de *número de puerto* con un socket "crudo".

1.20.1. Reglas para la Salida de Sockets "Crudos"

La salida normal se realiza generalmente con las funciones `sendto` o `sendmsg` especificando la dirección IP de destino.

Si no se activa la opción IP_HDRINCL, la dirección inicial de los datos para escritura es el primer byte que sigue a la cabecera IP, ya que es el kernel el que creará dicha cabecera. Por tanto, es el kernel quien define el campo de protocolo en la cabecera IPV4, ignorando el tercer campo de la función `socket`.

Si se activa la opción IP_HDRINCL, la dirección inicial de los datos para escritura es el primer byte de la cabecera IP. La cantidad de datos a escribir debe incluir el tamaño de dicha cabecera IP. El proceso debe responsabilizarse por construir la cabecera IP con excepción del campo de identificación (identificador), al que deberá dársele el valor de cero (con esto se le dice al kernel que le asigne un valor) y del campo de la suma de verificación de la cabecera (este campo es siempre llenado por el kernel).

Con IPV4, es responsabilidad del proceso creado por el programador el cálculo y escritura de cualquier suma de verificación que sea requerida posterior a la cabecera IPV4.

1.20.2. Reglas para la Entrada de Sockets "Crudos"

En primer lugar, se debe definir cuáles datagramas IP son los que el kernel pasa a sockets "crudos":

1. Los paquetes TCP y UDP recibidos NUNCA son pasados a sockets "crudos". Si un proceso desea leer datagramas IP que contengan paquetes TCP o UDP deben ser leídos en la capa de enlace de datos.
2. La mayoría de paquetes ICMP se pasan a sockets "crudos" después de que el kernel termine de procesar el mensaje ICMP. Lo mismo sucede con los mensajes IGMP.
3. Todos los datagramas IP con un campo de protocolo que el kernel no comprenda son pasados a sockets "crudos". Lo único que el kernel hace en estos casos es verificar algunos campos de cabecera como la versión, suma de verificación, longitud de cabecera y dirección IP destino.
4. Si el datagrama llega en fragmentos, nada es pasado a socket "crudo" hasta que todos los fragmentos hayan sido re-ensamblados.

Cuando el kernel tiene una datagrama IP para pasar a socket "crudos", todos los sockets "crudos" son examinados por todos los procesos, buscando coincidencias. Se envía una copia del datagrama IP a cada socket coincidente, realizando antes de ello 3 pruebas que el socket debe cumplir para poder recibir el datagrama:

a) Si se especifica un protocolo diferente de cero al crear el socket "crudo", el campo de protocolo del datagrama recibido debe coincidir con este valor o dicho datagrama es rechazado.
b) Si la dirección IP local está unida al socket "crudo" por medio de `bind`, la dirección IP destino del datagrama recibido debe coincidir con esta dirección de enlace o el datagrama es rechazado.
c) Si una dirección IP foránea se especificó en el socket "crudo" por medio de `connect`, la dirección IP fuente del datagrama recibido debe coincidir con esta dirección o el datagrama es rechazado.

CONCLUSIONES DEL CAPITULO I

- La programación usando sockets es uno de los modelos universales para la creación de aplicaciones distribuidas en los sistemas operativos orientados a la comunicación de datos debido a su simplicidad y practicidad, ya que ha sido diseñado para utilizarse en aplicaciones de propósito general y así obtener el máximo provecho del modelo de programación cliente-servidor.

- Las aplicaciones cliente-servidor creadas utilizando los conceptos vertidos en este capítulo se basan en el protocolo IPV4. Debido a que algunas estructuras, constantes y funciones se encuentran definidas para trabajar únicamente con dicho protocolo. En caso que se requiera utilizar estas aplicaciones bajo IPV6 será necesario hacer las modificaciones respectivas.

- Al momento de realizar una aplicación cliente-servidor, es necesario dar un tratamiento a los procesos "zombies", ya que si éstos no son finalizados pueden llegar a saturar la tabla de procesos del sistema operativo. Es por ello que las aplicaciones servidoras a crear deben contener el código necesario para manejar esta situación.

- La programación con sockets proporciona funciones que nos permiten modificar el comportamiento por defecto de un canal de comunicación, lo cual hace posible al programador adaptar de mejor manera una aplicación a necesidades específicas.

- Los dos tipos de servidores más ampliamente utilizados son el *servidor iterativo orientado a la NO conexión* y el *servidor concurrente orientado a la conexión*. El primero se usa cuando el servicio solicitado es atendido en un tiempo corto, por lo que no interesa crear varias copias del proceso servidor ya que un solo proceso es capaz de atender todas las peticiones. Por otro lado, un servidor concurrente orientado a la conexión se emplea en aplicaciones en las que el cliente establece una conexión con el servidor que dura un tiempo considerable, por lo que se hace necesario que exista un proceso servidor por cada cliente atendido.

- Los sockets "crudos" proveen ventajas que no pueden ser adquiridas con sockets normales (TCP y UDP), ya que permiten leer y escribir paquetes ICMP e IGMP (entre otros) y hasta construir el encabezado IP de un datagrama. Debido a estas ventajas se hace posible la creación de programas orientados al análisis de ciertos parámetros de red como generación de tráfico.

REFERENCIAS BIBLIOGRAFICAS

[1] Brian W. Kernighan.
"El Entorno de Programación UNIX"."
Editorial Prentice Hall, 1992.
Primera Edición.

[2] W. Richard Stevens.
"Unix Network Programming, Networking API's Sockets and XTI".
Editorial Prentice Hall, 1998.
Segunda Edición.

[3] Uyless Black
"Redes de Computadoras. Protocolos, Normas e Interfaces"
Editorial RA-MA, 1995.
Segunda Edición.

[4] Douglas E. Comer
"Redes Globales de Información con Internet y TCP/IP. Principios Básicos, Protocolos y Arquitectura".
Editorial Prentice-Hall Hispanoamericana, 1996.
Tercera Edición.

[5] Jean-Marie Rifflet
"Comunicaciones en UNIX".
Editorial McGraw-Hill, 1992.
Primera Edición.

[6] Uday O. Pabrai
"UNIX: Interconexión de Redes"
Editorial RA-MA, 1997.
Segunda Edición.

[7] Páginas de Manual (Páginas man) de todas las funciones mencionadas.

[8] An Advanced 4.3BSD Interprocess Comunicación Tutorial
http://www.sparc.spb.su/jjb/Docs/internet/ipc-tutorial/index.html

[9] "An Advanced Socket Comunicación Tutorial"
http://viks.mvrop.org/networking/sock_advanced_tut.html

[10] Jim Frost
"UNIX Signals and Process Groups"
Agosto 17 de 1994.
http://world.std.com/~jimf/papers/signals/signals.html

[11] "The Story of the PING Program"
ftp://ftp.arl.mil/pub/ping.shar

CAPITULO
II

EL ESTANDAR XDR

Introducción

Siempre que en una red interactúan ordenadores con diferente arquitectura, lo cual hoy en día ya no es un caso aislado, pueden darse problemas debido a las diferentes representaciones internas de datos entre arquitecturas. Para solucionar este inconveniente, la técnica más aceptada es traducir los datos de la representación interna de cada máquina a una representación estándar o de red, en el emisor, haciendo posteriormente la operación inversa en el receptor.

Lo anterior puede conseguirse por medio de uno de los estándares de conversión más usados actualmente, conocido como *Representación Externa de Datos* (XDR: *External data Representation*). En este capítulo se trata dicho estándar, los tipos de datos que comprende, sintaxis y principales funciones de las que se vale para la codificación y decodificación de datos. Todo esto con el objeto de que sea posible construir programas cliente-servidor capaces de comunicarse sin importar la arquitectura de ordenador sobre el que se ejecuten.

XDR es uno de los componentes de otro protocolo más amplio conocido como RPC (*Remote Procedure Call*), el cual será tratado en el siguiente capítulo.

2.0. Generalidades Sobre XDR

XDR (Representación Externa de Datos) es el estándar más usado en ambientes de red para la codificación y transferencia de datos entre ordenadores con diferente arquitectura, es decir, se encarga de la conversión de los datos a una forma de representación estándar para que dichos datos puedan ser procesados por cualquier tipo de ordenador, sin importar su arquitectura.

El único uso que se le da al lenguaje XDR es el de la conversión de formatos de datos, *no es un lenguaje de programación*. El lenguaje XDR es similar al lenguaje C. XDR asume que los datos se componen de unidades de 8 bits (bytes).

XDR representa todos los datos en múltiplos de 4 bytes (32 bits), esto se hace con el objeto se solucionar problemas de alineamiento para la mayoría de las diferentes arquitecturas de ordenadores[9]. Lo anterior trae una desventaja: los datos tienden a crecer hasta el múltiplo de 4 bytes más cercano, haciendo que la cantidad de información crezca, usualmente rellenando el espacio correspondiente con ceros.

9 Este estándar no es totalmente compatible con los ordenadores basados en la arquitectura Cray, cuyo alineamiento se basa en palabras de 8 bytes (64 bits).

Sin embargo, no existe un lenguaje único para la conversión de datos entre diferentes arquitecturas. Los principales son el ASN.1 (*Abstract Syntax Notation 1* o *Notación de Sintaxis Abstracta 1*) definida pro ISO para el modelo OSI y el estándar XDR propuesto por SUN Microsystems[10] para Internet (descrito arriba). Este último es el más aceptado actualmente.

La norma XDR es reciente, esto implica que las aplicaciones más antiguas de Internet no la utilizan. XDR se encuentra principalmente en las aplicaciones basadas en RPC[11].

2.1. Tipos de Datos en XDR

XDR es la sintaxis de representación de datos definida en la familia de protocolos de TCP/IP. La sintaxis XDR (*eXternal Data Representation*)[12] e incluye los siguientes tipos de datos:

2.1.1. Enteros

Un entero XDR es una palabra de 32 bits que codifica un número entero dentro del rango [-2147483648, +2146483647]. El entero se representa en notación complemento a 2, con su primera dirección de memoria como su byte más significativo (big endian). Su declaración es la siguiente:

```
int identificador
```

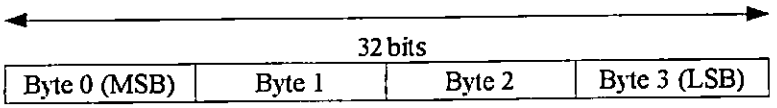


Figura 10: Representación de un entero en formato XDR.

Ejemplo:

```
int temperatura;
```

2.1.2. Enteros sin Signo

Un entero sin signo XDR es una palabra de 32 bits en el rango [0, 4294967295]. Su representación es idéntica a la de los enteros y se declara de la siguiente forma:

```
unsigned int identificador
```

Ejemplo:

```
unsigned int edad;
```

2.1.3. Enumeraciones

Los datos enumerados son un subconjunto de los enteros, para el cual cada entero tiene un nombre asignado, siendo su representación igual que los enteros. Se declaran de la siguiente forma:

```
enum {nombre_del_identificador = constante,...} identificador
```

Un ejemplo de este tipo de representación es la siguiente:

10 El estándar XDR fué propuesto en 1987.
11 Remote Procedure Call (LLamadas a Procedimientos Remotos).
12 XDR está definida en el estándar RFC-1014.

```
enum {Lun=1, Mar=2, Mie=3, Jue=4, Vie=5, Sab=6, Dom=7} día_semana;
```

Es un error el asignar a un tipo de dato enum un entero que no ha sido asignado en la declaración respectiva.

2.1.4. Booleanos

Este tipo de datos solamente pueden tener 2 valores: 0 (falso) o 1 (verdadero). Su declaración es:

```
bool identificador
```

Esto es el equivalente a:

```
enum { FALSE = 0, TRUE = 1 } identificador.
```

Ejemplo:

```
bool respuesta;
```

2.1.5. Hiper Entero e Hiper Entero sin Signo

Estos 2 tipos de datos son la versión de 8 bytes (64 bits) de los enteros y los enteros sin signo respectivamente. Su declaración tiene el formato siguiente:

```
hyper identificador;           /* hiper entero */
unsigned hyper identificador;  /* hiper entero sin signo */
```

2.1.6. Punto Flotante

Son tipos de datos múltiplos de 32 bits (4 bytes) que usan el estándar IEEE para la normalización de números de punto flotante . Su formato de declaración es:

```
float identificador;              /* 32 bits, precisión simple */
double identificador;             /* 64 bits, precisión doble */
quadruple identificador;          /* 128 bits, precisión cuádruple */
```

Ejemplo:

```
float promedio;
double distancia;
quadruple saldo;
```

2.1.7. Datos Opacos de Longitud Fija

Se les llama datos opacos a aquellos datos que no son interpretados por el sistema de una forma en especial. Su formato de declaración es:

```
opaque identificador [n];
```

Donde n es el número de bytes necesarios para contener los datos opacos. Si n no es múltiplo de 4, se rellena con ceros para cumplir con esta condición.

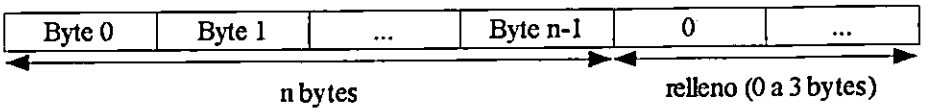


Figura 11: Representación de datos opacos de longitud fija en formato XDR.

Ejemplo: .

```
opaque cadena [5];
```

2.1.8. Datos Opacos de Longitud Variable

Este tipo de datos se define como una secuencia de n bytes arbitrarios, donde n se codifica como un entero sin signo que encabeza a dicha secuencia. Ya que n es un entero sin signo, se codifica como una palabra de 32 bits, siendo los primeros 4 bytes de la secuencia los que especifican la longitud de la misma. Su declaración sigue el formato:

```
opaque identificador <m>;
opaque identificador < >;
```

Al tratarse de datos opacos de longitud variable, m refleja la cantidad máxima de bytes que pueden usarse para contener los datos opacos. Si no se especifica dicho límite, se asume el valor máximo (2^{32}-1). En caso de que la longitud no sea múltiplo de 4 bytes, se rellena con ceros hasta alcanzar el múltiplo más cercano.

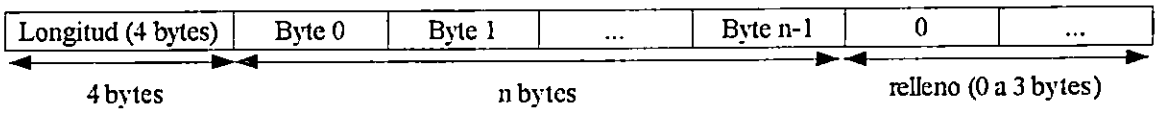


Figura 12: Representación de datos de longitud variable en formato XDR.

Ejemplo:

```
opaque cadena <10>;      /* Longitud máxima: 10 caracteres */
opaque cadena < >;       /* Longitud máxima: 2^32-1 caracteres */
```

2.1.9. Cadenas

Una cadena se define como un conjunto de n caracteres ASCII, codificado n como un entero sin signo seguido de dichos caracteres. Si la longitud de la cadena no es múltiplo de 4, se rellena con ceros. Su declaración obedece al formato siguiente:

```
string objeto <n>;
string objeto < >;
```

Donde n define la cantidad máxima de bytes que contendrá la cadena. Si no se especifica, se asume longitud máxima (2^{32} -1). Su formato de presentación es idéntico al de los datos opacos de longitud variable.

2.1.10. Arreglos de Longitud Fija

La forma en que se declara un arreglo (array) de longitud fija, cuyos elementos son todos del mismo tipo, es la siguiente:

```
tipo identificador [n];
```

El tamaño de cada elemento es un múltiplo de 4 bytes. Todos los elementos poseen la misma longitud y su numeración va de cero (0) a n-1

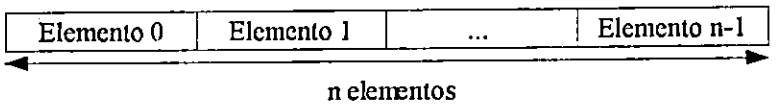


Figura 13: Representación de arreglos de longitud fija en formato XDR.

Ejemplo:

```
int elemento [6];
```

2.1.11. Arreglos de Longitud Variable

La declaración de los arreglos de longitud variable sigue el formato siguiente:

```
tipo identificador <m>;
tipo identificador < >;
```

La constante m define la máxima cantidad de elementos que el arreglo puede aceptar. Si m no se especifica, se asume el valor máximo (2^{32}-1). Todos los elementos tienen la misma longitud y son del mismo tipo.

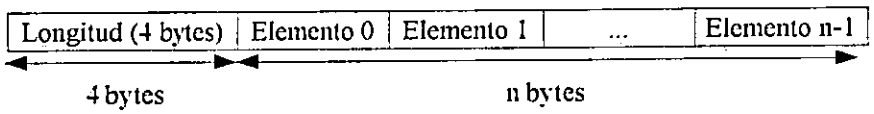


Figura 14: Representación de arreglos de longitud variable en formato XDR.

2.1.12. Estructuras

Las estructuras se declaran de la forma siguiente:

```
struct identificador
{ declaración_de_componente_A;
  declaración_de_componente_B;
 ...
};
```

Los componentes de la estructura son codificados en el orden en que son declarados. El tamaño de cada componente es un múltiplo de 4 bytes y no son necesariamente del mismo tipo, pudiendo tener tamaños diferentes.

Ejemplo:

```
struct fecha
{     int año;
      string mes<10>;
      int día;
};
```

Su representación sigue la forma:

Componente A	Componente	Componente C	...

Figura 15: Representación de una estructura en formato XDR.

2.1.13. Unión con Discriminante

La unión con discriminante comprende un conjunto de posibles valores de los cuales sólo se codifica uno, esto viene determinado por el valor que tenga el discriminante. El discriminante suele ser un tipo de dato entero (con o sin signo) o un tipo de dato enumerado. Los componentes u opciones de la unión son llamados "brazos" y se declaran precedidos por el valor del discriminante que lo selecciona. Se puede tener una declaración *"por defecto"* para el cual el discriminante tome un valor no contemplado por ninguno de los brazos, aunque esto no es obligatorio.

La unión con discriminante sigue la siguiente sintaxis:

```
union switch (declaración_discriminante)
{ case valor_A:
            brazo-declaración_A;
 case valor_B:
            brazo-declaración_B;
...
 default:
            declaración_por_defecto;
} identificador;
```

El tamaño de cada declaración "brazo" es múltiplo de 4 bytes.

Ejemplo:

```
union switch (tipo_entrada entrada)
{
      case VEHÍCULO:     string matrícula <8>;
      case PERSONA:      int carnet;
      case ANIMAL:       string nombre <10>;
} control_entrada;
```

Donde:

```
enum
{
          PERSONA=0,
          ANIMAL=1,
          VEHÍCULO=2
} tipo_entrada;
```

Su representación sigue la forma:

Discriminante (4 bytes)	Brazo 1	Brazo 2	...

Figura 16: Representación de una unión con discriminante en formato XDR.

2.1.14. Void

Un dato tipo void en XDR es una cantidad de 0 bytes. Este tipo de datos son útiles para describir operaciones que no toman datos como entrada o que no devuelven datos como salida. Se usa también en uniones donde al menos uno de sus brazos no genera ningún dato. Su declaración tiene el formato siguiente:

```
void;
```

2.1.15. Constantes

La declaración de constantes en XDR se da de la siguiente forma:

```
const identificador = valor;
```

Un ejemplo de lo anterior es:

```
const PI = 3.14159;
```

2.1.16. Definición de Tipos (Typedef)

`typedef` es útil para definir nuevos identificadores al momento de declarar datos partiendo de tipos de datos preexistentes. Su sintaxis es:

```
typedef declaración;
```

Un ejemplo de lo anterior es:

```
typedef int mes[31];
```

El siguiente es un ejemplo del uso de las variables declaradas con `typedef`:

```
mes Octubre;
```

Lo cual es equivalente a:

```
int Octubre[31];
```

El nuevo tipo de nombre es en realidad el nombre de la variable en la parte declaratoria de `typedef`. Cuando se define un tipo basado en una estructura, dato enumerado o unión, la sintaxis puede tener cualquiera de las siguientes formas:

```
typedef tipo {·····} identificador;
tipo identificador{·····};
```

Como ejemplo, se presentan 2 formas equivalentes para definir los tipos de datos booleanos:

```
typedef enum { FALSO=0, VERDADERO=1 } bool;
enum bool { FALSO=0, VERDADERO=1 };
```

2.1.17. Datos Opcionales

Se usan para definir y codificar estructuras dinámicas, que en lenguaje C se expresan mediante punteros. Su sintaxis es:

```
tipo *identificador;
```

Se definen de la misma forma que los punteros del lenguaje C, aunque su significado no es el mismo. este tipo de datos equivalen a la declaración siguiente:

```
union switch (bool opcional)
{
        case TRUE:    tipo elemento;
        case FALSE:   void;
} identificador;
```

O bien a:

```
tipo identificador <1>;
```

Son útiles para definir estructuras de datos recursivas como arboles o listas enlazadas. Ejemplo:

```
struct *descriptor
{
        string elemento<>;
        descriptor siguiente; /* apunta al siguiente elemento */
}
```

Esta declaración es equivalente a:

```
union descriptor switch (bool opción)
{
        case TRUE:
                struct
                {
                    string elemento<>;
                    descriptor siguiente;
                } elemento;
        case FALSE: void;
};
```

O también a:

```
struct descriptor<1>
{
        string elemento<>;
        descriptor siguiente;
}
```

2.2. Filtros XDR

Para la conversión de datos del formato específico de cada ordenador al formato de red y viceversa, se debe generar una función bidireccional encargada de codificar y decodificar los datos de cada estructura específica. Esta función es conocida como *filtro*.

Todo filtro necesita de una zona de almacenamiento en memoria para realizar las operaciones de conversión de formatos. Dicha zona de almacenamiento se conoce como *flujo XDR (XDR stream)*.

El uso de los filtros XDR es siempre el mismo, siguiendo los pasos que se detallan a continuación:

1. Crear un flujo para operar con los datos.
2. Llamar al filtro respectivo para hacer las conversiones
3. Destruir el flujo.

Para gestionar el flujo, se necesita una estructura especial, tipo XDR, que contenga información sobre qué operación se realizará sobre dicho flujo. Dicha estructura se encuentra definida en el archivo de cabecera <rpc/xdr.h>, la cual se muestra a continuación:

```
typedef struct
{       enum xdr_op   x_op;       /* Operación a realizar en el flujo. */

        struct  xdr_ops
        { bool_t      (*x_getlong)();   /* Toma un entero largo del flujo.*/
          bool_t      (*x_putlong)();   /* Pone un entero largo en el flujo.*/
          bool_t      (*x_getbytes)();  /* Toma algunos bytes del flujo.*/
          bool_t      (*x_putbytes)();  /* Pone algunos bytes en el flujo.*/
          u_int       (*x_getpostn)();  /* Devuelve la posición actual. */
                                        /* del puntero de lectura/escritura.*/
          bool_t      (*x_setpostn)();  /* Posiciona el flujo XDR.*/
          int32_t     *(*x_inline)();   /* Devuelve un puntero a un*/
                                        /* buffer del flujo.*/
          void        (*x_destroy)();   /* Libera el flujo XDR.*/
        } *x_ops;

        caddr_t x_public;   /* Datos del usuario.*/
        caddr_t x_private;  /* Puntero a datos privados.*/
        caddr_t x_base;
        int           x_handy;
}       XDR;
```

NOTA: No se muestran los parámetros de las funciones dentro de la estructura `xdr_ops` para efectos de simplicidad. Sin embargo, estos aparecen en el archivo de cabecera mencionado anteriormente.

Algunas distribuciones pueden contener elementos adicionales, pero los descritos arriba aparecen en todas ellas. Generalmente se hará referencia a esta estructura por medio de un puntero.

2.3. Operaciones Sobre un Flujo XDR

Las operaciones sobre un flujo XDR especifican la acción a realizar en el filtro con relación a los datos. Las operaciones se definen dentro del tipo de datos enumerado `xdr_op`, definido en el archivo de cabecera <rpc/xdr.h>:

```
enum xdr_op {
             XDR_ENCODE = 0,  /* Codificar.*/
             XDR_DECODE = 1,  /* Decodificar.*/
             XDR_FREE = 2     /* Liberar espacio de memoria reservado*/
                              /* automáticamente en operaciones de codificación */
                              /* anteriores.*/
           };
```

2.3.1. Creación de un Flujo

El proceso para crear un flujo en memoria empieza con declarar un puntero a una estructura XDR y después inicializar dicha estructura antes de cualquier operación de codificación o decodificación. Esto se logra por medio de la función `xdrmem_create`, definida en el archivo de cabecera <rpc/xdr.h>. Su formato es:

```
void xdrmem_create (XDR *xdrptr, char *buffer, int l, enum xdr_op operación);
```

Valor Devuelto: Ninguno.

xdrptr: Puntero a estructura tipo XDR.
buffer: Dirección de buffer (datos a codificar o decodificar).
l: longitud máxima de los datos a tratar.
operación: Operación a realizar. Nótese que este parámetro es un tipo de dato enumerado `xdr_op`, por lo que puede tomar uno de 3 valores: `XDR_ENCODE`, `XDR_DECODE` ó `XDR_FREE`.

Una forma alternativa para crear un flujo XDR es por medio de los mecanismos de entrada/salida con archivos. En lugar de usar un buffer en memoria, se emplea un descriptor a un archivo. Lo anterior se consigue por medio de la función `xdrstdio_create`, definida en el archivo de cabecera `<rpc/xdr.h>`. Su formato es:

```
void xdrstdio_create (XDR *xdrptr, FILE *archivo, enum xdr_op operación);
```

Valor Devuelto: Ninguno.

xdrptr: Puntero a estructura tipo XDR.
buffer: Puntero al archivo de datos a codificar o decodificar.
operación: Operación a realizar.

2.3.2. Destrucción de un Flujo

Una vez terminadas las operaciones de codificación y decodificación de datos, es conveniente liberar los recursos y datos de gestión empleados por el flujo, a través de la función `xdr_destroy`, especificada en el archivo de cabecera `<rpc/xdr.h>`.

```
void xdr_destroy(XDR *xdrp);
```

Valor Devuelto: Ninguno.

xdrptr: Puntero a estructura tipo XDR.

2.3.3. Longitud de Datos Convertidos

Se obtiene usando la función `xdr_getpos`, definida en el archivo de cabecera `<rpc/xdr.h>`. Se debe emplear para saber cuántos bytes se van a enviar. Su formato es:

```
int xdr_getpos (XDR *xdrp);
```

Valor Devuelto: Longitud de datos convertidos.

xdrptr: Puntero a estructura tipo XDR.

2.4. Operaciones de Codificación y Decodificación

Estas operaciones e realizan por medio de un filtro y se usan para agregar o extraer información de un flujo XDR. Generalmente los filtros son creados por el compilador especial `rpcgen`. Los archivos generados por dicho compilador generalmente tienen el formato `xdr_aplicación`.

Todas las funciones filtro son muy parecidas entre sí, variando solamente en el nombre de la aplicación. Su forma genérica es la siguiente:

```
bool_t xdr_aplicación (XDR *xdrp, <tipo> *datos);
```

Siempre se pasan las direcciones del descriptor XDR y la de los datos a convertir. La ejecución de las funciones filtro devuelven un valor tipo `bool_t`, que puede ser 1 (VERDADERO) o cero (FALSO), dependiendo si la operación se realizó o no con éxito.

2.5. Principales Filtros XDR de Conversión

Son aquellas funciones que vienen ya definidas en las librerías XDR, capaces de realizar las conversiones básicas. Los filtros generados por `rpcgen` se basan en ellos. Dichos filtros de conversión pueden clasificarse en 2 tipos:

Simples: Convierten directamente entre datos C y XDR.
Complejos: Se usan para convertir estructuras de mayor complejidad.

Los nombres de los filtros suelen llevar el prefijo `xdr_`, seguido por el nombre del respectivo tipo de datos, según se muestra a continuación:

Tipo XDR	Filtro
bool	`xdr_bool`
int	`xdr_int`
char	`xdr_char`
unsigned int	`xdr_u_short`
short	`xdr_short`
long	`xdr_long`
unsigned long	`xdr_u_long`
float	`xdr_float`
double	`xdr_double`
void	`xdr_void`
enum	`xdr_enum`

Para algunos tipos más complejos, existen los siguientes filtros:

Tipo XDR	Filtro
Arreglo de longitud variable, con elementos de longitud variable	`xdr_array`
Arreglo de bytes de longitud variable	`xdr_bytes`
Datos opacos de longitud fija	`xdr_opaque`
Referencias a punteros	`xdr_pointer`
Referencias a objetos	`xdr_reference`
Cadenas de caracteres, terminados con NULL	`xdr_string`
Uniones Discriminantes	`xdr_union`
Arreglos de longitud fija con elementos de longitud variable	`xdr_vector`
Cadenas de caracteres de longitud variable, terminadas con NULL	`xdr_wrapstring`

Todos los filtros de conversión devuelven un valor tipo `bool_t`, el cual solamente toma 2 valores:
TRUE: Si la conversión se realizó con éxito.
FALSE: Si la conversión no se realizó satisfactoriamente.

2.5.1. xdr_array

Este filtro realiza la conversión entre arreglos de longitud variable y su respectiva representación externa.

```
bool_t xdr_array(XDR *xdrs, char **arrp, u_int *pnum, u_int máximo,
                 u_int t_elem, xdrproc_t proc);
```

xdrs: Puntero a estructura XDR.
arrp: Puntero al puntero del arreglo.
pnum: Puntero al número de elementos del arreglo. Su cantidad no debe superar a máximo.
máximo: El tamaño máximo que puede tener pnum.
t_elem: El tamaño de los elementos que componen el arreglo.
proc: Rutina XDR encargada de llamar a los manejadores de los elementos del arreglo.

2.5.2. xdr_bool

Este filtro realiza la conversión entre booleanos (enteros) y su respectiva representación externa.

```
bool_t xdr_bool (XDR *xdrs, bool_t *bp);
```

xdrs: Puntero a estructura XDR.
bp: Puntero a la variable booleana.

2.5.3. xdr_bytes

Este filtro realiza la conversión entre un arreglo de bytes de longitud variable y su respectiva representación externa. Dicho arreglo de bytes no puede ser mayor a máximo.

```
bool_t xdr_bytes (XDR *xdrs, char **bpp, u_int *longitud, u_int máximo);
```

xdrs: Puntero a estructura XDR.
bpp: Puntero al puntero de la cadena de bytes. Si este parámetro vale NULL, xdr_bytes asume la longitud de la cadena como máximo.
longitud: Puntero al tamaño de la cadena de bytes.
máximo: Longitud máxima de la cadena de bytes.

2.5.4. xdr_char

Este filtro realiza la conversión representaciones internas de caracteres y su respectiva representación externa.

```
bool_t xdr_char(XDR *xdrs, char *cp);
```

xdrs: Puntero a estructura XDR.
cp: Puntero al carácter.

2.5.5. xdr_double

Este filtro realiza la conversión entre números de precisión doble y su respectiva representación externa.

```
bool_t xdr_double(XDR *xdrs, double *dp);
```

xdrs: Puntero a estructura XDR.
dp: Puntero al número de punto flotante con doble precisión.

2.5.6. xdr_enum

Este filtro realiza la conversión entre tipos de datos enumerados (enteros) y su respectiva representación externa.

```
bool_t xdr_enum(XDR *xdrs, enum_t *ep);
```

xdrs: Puntero a estructura XDR.
ep: Puntero a la estructura de datos enumerados.

2.5.7. xdr_float

Este filtro realiza la conversión entre números de precisión simple y su respectiva representación externa.

```
bool_t xdr_float(XDR *xdrs, float *fp);
```

xdrs: Puntero a estructura XDR.
fp: Puntero al número de punto flotante con precisión simple.

2.5.8. xdr_hyper

Este filtro realiza la conversión entre hiper enteros y su respectiva representación externa.

```
bool_t xdr_hyper(XDR *xdrs, quad *hp);
```

xdrs: Puntero a estructura XDR.
hp: Puntero a un hiper entero.

2.5.9. xdr_int

Este filtro realiza la conversión entre números enteros y su respectiva representación externa.

```
bool_t xdr_int(XDR *xdrs, int *ip);
```

xdrs: Puntero a estructura XDR.
ip: Puntero a un entero.

2.5.10. xdr_long

Este filtro realiza la conversión entre enteros largos y su respectiva representación externa.

```
bool_t xdr_long(XDR *xdrs, long *lp);
```

xdrs: Puntero a estructura XDR.
lp: Puntero a un entero largo.

2.5.11. xdr_opaque

Este filtro realiza la conversión entre datos opacos de longitud fija y su respectiva representación externa. Este filtro trata a los datos como un arreglo de bytes de longitud fija y no trata de convertir dichos bytes.

```
bool_t xdr_opaque(XDR *xdrs, char *op, u_int tamaño);
```

xdrs: Puntero a estructura XDR.
op: Puntero a los datos opacos.
tamaño: Longitud de los datos opacos en bytes.

2.5.12. xdr_pointer

Este filtro realiza la conversión entre estructuras que contienen punteros a otras estructuras (como una lista de enlaces) y su respectiva representación externa. Este filtro es muy similar a `xdr_reference` con la diferencia de que `xdr_pointer` maneja punteros con valor nulo y también puede convertir los valores de punteros a booleanos. Si el valor del booleano es TRUE, los datos siguen al booleano.

```
bool_t xdr_pointer(XDR *xdrs, char **objpp, u_int tamaño, xdrproc_t proc);
```

xdrs: Puntero a estructura XDR.
objp: Puntero al puntero de los datos que serán convertidos.
tamaño: Longitud de la estructura de datos en bytes.
proc: Procedimiento XDR que realiza la conversión de la estructura entre su forma original y su representación externa.

2.5.13. xdr_reference

Este filtro realiza la conversión entre punteros indirectos, así como los datos a los que apunta, y su respectiva representación externa.

```
bool_t xdr_reference(XDR *xdrs, char **objpp, u_int tamaño, xdrproc_t proc);
```

xdrs: Puntero a estructura XDR.
objp: Puntero al puntero de la estructura que contiene los datos que serán convertidos. Si este argumento es nulo, `xdr_reference` reserva la cantidad de memoria suficiente para la decodificación. Este argumento NO debe ser cero durante la codificación.
tamaño: Longitud de la estructura en bytes.
proc: Procedimiento XDR que realiza la conversión de la estructura entre su forma original y su representación externa.

2.5.14. xdr_short

Este filtro realiza la conversión entre enteros cortos y su respectiva representación externa.

```
bool_t xdr_short(XDR *xdrs, short *sp);
```

xdrs: Puntero a estructura XDR.
sp: Puntero a un entero corto.

2.5.15. xdr_string

Este filtro realiza la conversión entre cadenas y su respectiva representación externa. La cadena no puede ser más larga que max. Esta rutina es igual a xdr_wrapstring, con la diferencia de que con xdr_string es posible especificar la longitud máxima.

```
bool_t xdr_string(XDR *xdrs, char **spp, u_int max);
```

xdrs: Puntero a estructura XDR.
spp: Puntero al puntero de la cadena de caracteres. Si este argumento es nulo, xdr_string reserva la memoria necesaria para la decodificación.
max: El tamaño máximo de la cadena de caracteres.

2.5.16. xdr_u_char

Este filtro realiza la conversión entre caracteres sin signo y su respectiva representación externa.

```
bool_t xdr_u_char(XDR *xdrs, char *ucp);
```

xdrs: Puntero a estructura XDR.
spp: Puntero a un caracter (sin signo).

2.5.17. xdr_u_hyper

Este filtro realiza la conversión entre hiper enteros sin signo y su respectiva representación externa.

```
bool_t xdr_u_hyper(XDR *xdrs, unsigned quad *uhp);
```

xdrs: Puntero a estructura XDR.
uhp: Puntero a un hiper entero sin signo.

2.5.18. xdr_u_int

Este filtro realiza la conversión entre enteros sin signo y su respectiva representación externa.

```
bool_t xdr_u_int(XDR *xdrs, unsigned int *uip);
```

xdrs: Puntero a estructura XDR.
uip: Puntero a un entero sin signo.

2.5.19. xdr_u_long

Este filtro realiza la conversión entre enteros largos sin signo y su respectiva representación externa.

```
bool_t xdr_u_long(XDR *xdrs, unsigned long *ulp);
```

xdrs: Puntero a estructura XDR.
ulp: Puntero a un entero largo sin signo.

2.5.20. xdr_u_short

Este filtro realiza la conversión entre enteros cortos sin signo y su respectiva representación externa.

```
bool_t xdr_u_short(XDR *xdrs, unsigned short *usp);
```

xdrs: Puntero a estructura XDR.
usp: Puntero a un entero corto sin signo.

2.5.21. xdr_union

Este filtro realiza la conversión entre una unión con discriminante y su respectiva representación externa. Este filtro convierte primero el discriminante de la unión en `dscmp`. Este discriminante es siempre del tipo `enum_t`. A continuación convierte los datos de la unión ubicada en `unp`, para lo cual busca la estructura apuntada por el argumento `choices`. Si el filtro encuentra una coincidencia, se llama al respectivo procedimiento del par valor/procedimiento para realizar la conversión. Caso contrario se llama a la rutina `default`.

```
bool_t xdr_union(XDR *xdrs, enum *dscmp, char *unp,
                 struct xdr_discrim *choices, xdrproc_t default);
```

xdrs: Puntero a estructura XDR.
dscmp: Puntero a una unión discriminante.
unp: Puntero a los datos de la unión.
choices: Puntero a un arreglo de estructuras `xdr` discriminantes. Cada estructura contiene un par valor/procedimiento. La última estructura del arreglo es siempre un puntero a un valor nulo (NULL).
default: Dirección de la rutina XDR por defecto que será aplicada en caso de que el valor del argumento `dscmp` no sea encontrado en el arreglo `choices`.

2.5.22. xdr_vector

Este filtro realiza la conversión entre un arreglo de longitud fija y su respectiva representación externa.

```
bool_t xdr_vector(XDR *xdrs, char **vecpp, u_int num, u_int t_elem,
                  xdrproc_t proc);
```

xdrs: Puntero a estructura XDR.
vecpp: Puntero al puntero del arreglo.
num: Número de elementos del arreglo.
t_elem: Tamaño en bytes de los elementos del arreglo.
proc: Rutina XDR encargada de manejar los elementos.

2.5.23. xdr_void

Se usa este filtro en un programa que no pasa datos en una llamada a un procedimiento remoto. La mayoría de rutinas cliente y servidor esperan sean llamadas rutinas XDR, aún cuando no hay datos que pasar.

```
bool_t xdr_void();
```

2.5.24. xdr_wrapstring

Este filtro realiza llama a `xdr_string` con la constante MAXUNSIGNED como tercer argumento. Esta constante es el valor máximo que puede tomar un entero sin signo.

```
bool_t xdr_wrapstring(XDR *xdrs, char **spp);
```

xdrs: Puntero a estructura XDR.
spp: Puntero a un puntero a una cadena (string). Si este parámetro es nulo, `xdr_wrapstring` reserva la memoria suficiente para la decodificación.

2.6. Archivos Involucrados en una Aplicación XDR Típica

Los archivos a generar en una aplicación cliente-servidor utilizando el estándar XDR y el compilador `rpcgen` son los siguientes (aquellos escritos en negrita son los generados por el compilador `rpcgen`):

Archivo	*Contenido*
`aplicacion.x`	Especificaciones de estructuras XDR.
`aplicacion.h`	Archivos de cabecera.
`aplicación_xdr.c`	Filtros XDR
`aplicación_cliente.c`	Código de la aplicación cliente.
`aplicación_servidor.c`	Código de la aplicación servidor.

Tabla 13: Archivos que típicamente aparecen en una aplicación cliente-servidor usando el estándar XDR.

Las extensiones asignadas a estos archivos son las más frecuentemente usadas, lo que no impide que el programador pueda usar nomenclatura alterna según su conveniencia (por ejemplo, colocar extensión `.cpp` o `.cxx` a los archivos `.c`)

2.7. Ejemplo del Uso de XDR

Un ejemplo del uso de XDR es el siguiente:

El primer paso es definir una estructura básica, llamada `transferencia`, la cual ubicaremos en un archivo .x (para este caso, lo llamaremos `banco.x`):

```
struct transferencia
{
        string          beneficiario<20>;
        unsigned int    num_cuenta;
        float           cantidad;
        opaque          comprobación<30>;
};
```

Esta información necesita es pasada pasada por un compilador especial (el `rpcgen`), para traducirla al lenguaje C. Ejecutaremos el comando:

```
rpcgen -h banco.x
```

Con ello se obtendrá la estructura equivalente en C, el archivo `banco.h`. Debe recalcarse que ni la creación de este archivo ni su contenido son generados por el usuario, sino por el compilador mencionado.

```
#include <rpc/rpc.h>

struct transferencia
{
      char      *beneficiario;
      u_int     num_cuenta;
      float     cantidad;
      struct
      {
            u_int     comprobación_len;
            char      *comprobación_val;
      } comprobación;
};

typedef struct transferencia transferencia;
extern bool_t xdr_transferencia(XDR *, transferencia*);
bool_t xdr_transferencia();
```

La traducción de los tipos XDR a C se puede intuir. El archivo `banco.h` suministra una declaración de la estructura transferencia, y la definición de la función filtro, `xdr_transferencia`, que devuelve un valor booleano que indica el resuelto de la ejecución.

Este archivo de cabecera debe incluirse en la aplicación, insertando al principio de los archivos fuente del cliente y servidor, `banco.c` y `bancos.c` (asumiendo esos serán los nombres para las aplicaciones del cliente y del servidor respectivamente), la linea:

```
#include "banco.h"
```

Ademas, el compilador `rpcgen` nos suministra la función filtro para codificar y decodificar los datos. Este archivo es creado simultáneamente con `banco.h`. El archivo generado es `banco_xdr.c`:

```
#include <rpc/types.h>
#include <rpc/xdr.h>

#include "banco.h"

bool_t xdr_transferencia(XDR *xdrs, transferencia *objp)
{

      register long *buf;

      if (!xdr_string(xdrs, &objp->beneficiario, 20)) {
            return (FALSE);
      }
      if (!xdr_u_int(xdrs, &objp->num_cuenta)) {
            return (FALSE);
      }
      if (!xdr_float(xdrs, &objp->cantidad)) {
            return (FALSE);
      }
      if (!xdr_bytes(xdrs, (char **)&objp->comprobacion.comprobacion_val,
     (u_int *)&objp->comprobacion.comprobacion_len, 30)) {
            return (FALSE);
      }
     return (TRUE);
}
```

Esta función será invocada desde nuestra aplicación cuando queramos enviar o recibir datos, pasándoles la dirección de los datos a tratar y la operación a efectuar. La generación del filtro es trivial, ya que cada tipo elemental tiene su filtro, Así, el filtro de la estructura se hace en base a los filtros de los tipos elementales que la componen.

En el archivo fuente del cliente utilizaremos la estructura y el filtro como sigue:

```
#include "banco.h"
...
/*introducir valores de la estructura transferencia*/
...
/*codificarlos para enviarlos*/

if(!xdr_transferencia(&operacion, &transferencia))
{
      perror("fallo en la conversion);
      exit(1);
}

sendto(...);

/*espera respuesta*/
....
```

En el archivo fuente del servidor se hace el proceso inverso:

```
#include"banco.h"
...
recvfrom(...);
/*decodificación para procesarlos*/

if(!xdr_transferencia(&operación, &transferencia))
{
      perror("fallo en la conversión);
      exit(1);
}

/*envío de respuesta*/
...
```

Los tres archivos: `banco.h`, `banco_xdr.c` y `banco_cliente.c` serán compilados para generar el archivo ejecutable del cliente, mientras que los archivos `banco.h`, `banco_xdr.c` y `banco_servidor.c` se compilan para generar el del servidor.

En resumen, cuando se necesita convertir datos a formato de red, se realizan los siguientes pasos:

1. Crear un descriptor con la llamada `xdr<tipo>_create ()`.
2. Llamar al respectivo filtro para hacer la conversión
3. Calcular la longitud de los bytes convertidos con `xdr_getpos ()`. Esto dato servirá para ser usada en la función que se encargue de enviar los datos (`write, writen, sendto`, etc.)
4. Destruir el descriptor con `xdr_destroy ()`.

De igual forma, para convertir los datos captados a formato interno del receptor:

1. Crear un descriptor con la llamada `xdr<tipo>_create ()`.
2. Llamar al respectivo filtro para hacer la conversión.
3. Destruir el descriptor con `xdr_destroy ()`.

Este formato de llamadas es el mismo independientemente si las funciones respectivas son escritas por el programador o generadas por `rpcgen`.

CONCLUSIONES DEL CAPITULO II

- XDR es el estándar más utilizado para la conversión de representaciones de datos entre ordenadores con arquitectura diferente. Dado que la comunicación entre ordenadores con diferentes representaciones de datos es algo cotidiano en la actualidad, el estándar XDR acentúa su importancia como herramienta en la transferencia de información.

- Debido a su similitud con el lenguaje C, el aprendizaje de la programación con XDR se vuelve un proceso muy sencillo.

- XDR facilita la comunicación entre ordenadores sin importar su arquitectura. Sin embargo, incrementa el tráfico que circula por una red ya que los datos a ser transmitidos son expandidos a su múltiplo de 32 bits más cercano, además de incrementar ligeramente la complejidad en la programación tanto del lado del cliente como del servidor debido a la inclusión de rutinas de conversión de datos que hacen uso constante de punteros.

- En caso de que el programador necesite transmitir tipos de datos híbridos, cuya definición no se incluye dentro del estándar XDR, es posible la definición específica para dichos tipos de datos ya sea manualmente o a través de un compilador especial (`rpcgen`), el cual facilita la programación de la aplicación.

- En vista de que prácticamente todas arquitecturas de ordenadores manejan un mismo concepto de tipo de dato caracter, la información entre dos ordenadores puede ser transmitida sin utilizar XDR siempre que dicha información (cualquier tipo de dato) se envíe como un grupo de caracteres (convirtiéndose a tipo de dato carácter). Esta técnica es recomendada solamente si la información a transmitir no posee un grado de complejidad significativo.

REFERENCIAS BIBLIOGRAFICAS

[1] RFC 1832 (Request for Comments). Especificación del estandar XDR.

[2] Jean-Marie Rifflet
"Comunicaciones en UNIX".
Editorial McGraw-Hill, 1992.
Primera Edición.

[3] Páginas del Manual `rpcgen` y `xdr`.

[4] "Cisco IOS for S/390 RPC/XDR Programmer's Reference"
http://www.cisco.com/univercd/cc/td/doc/product/software/ioss390/ios390rp

[5] "Communications Programming Concepts": Primera Edición.
http://anguilla.u.arizona.edu/doc_link/en_US/a_doc_lib/aixprggd/progcomc/

[6] "DIGITAL TCP/IP Services for OpenVMS ONC RPC Programming"
http://cctr.umkc.edu/vms/72final/6528/

CAPITULO
III

LLAMADAS A PROCEDIMIENTOS REMOTOS

Introducción

En muchas ocasiones, el esquema bajo el que se lleva a cabo la programación con sockets llega a un punto en el que se puede alcanzar niveles de complejidad significativos, aún en aplicaciones relativamente sencillas. El lenguaje de *Llamadas a Procedimientos Remotos* (RPC: *Remote Procedure Call*) facilita de manera considerable al programador el realizar aplicaciones obviando muchos pasos que podrían resultar engorrosos y que generalmente no son el centro de la aplicación, tales como la codificación y decodificación de los datos a transmitir, la creación del canal de comunicación, etc.

Este capítulo se enfoca en el lenguaje de programación RPC, sus diferentes niveles de programación, principales funciones así como los pasos a seguir para construir aplicaciones utilizando este lenguaje. Se estudia también de manera breve el compilador `rpcgen`, el cual es ampliamente usado para generar el código básico de una determinada aplicación a través del lenguaje RPC.

3.0. Generalidades

Hasta el momento, las aplicaciones cliente-servidor descritas han seguido un esquema de programación repetitivo:

En el cliente	*En el servidor*
Llamada para contactar proceso servidor	Llamadas para contactar proceso cliente
Codificación de datos	Esperar por la recepción de datos
Envío de datos	Decodificación de datos
	Procesamiento de datos
	Codificación de datos
Recepción de datos	Envío de datos
Decodificación de datos	
Tratamiento de datos y obtención de resultados	

***Tabla 14:** Esquema de programación de una típica aplicación cliente-servidor.*

Este esquema de programación puede ser simplificado considerablemente si la mayoría de las llamadas descritas anteriormente se integran en una sola llamada de alto nivel, la cual invocará el cliente cuando se requiera ejecutar el procedimiento de la máquina remota.

Lo anterior puede lograrse con el lenguaje de *Llamadas a Procedimientos Remotos* o RPC (*Remote Call Procedure*), una extensión del lenguaje XDR que, a diferencia de este último, es usada para efectos de programación como tal.

Al igual que XDR, el lenguaje RPC es muy similar al lenguaje C.

Por medio del lenguaje RPC pueden ser ocultadas al programador todas las llamadas XDR necesarias, siendo vistas las llamadas a procedimientos remotos con la misma facilidad que un proceso local. Los procedimientos por medio de los cuales las llamadas XDR son ocultadas se conocen como *cabos* o *stubs*.

Existen muchas implementaciones de RPC, siendo una de las más empleadas la establecida por Sun Microsystems[13], conocida como *Open Network Computing* (ONC). Esta implementación incluye tanto RPC como XDR, utilizando típicamente los protocolos UDP y TCP como protocolos de transporte.

Esta implementación utiliza el método de *tipado implícito*, lo que significa que se transmiten por la red únicamente el valor de las variables y no su tipo.

La RPC de Sun soporta 2 tipos de llamadas a procedimientos remotos. Una versión apoya los sockets de BSD como por ejemplo API (*Aplication Program Interface*) que funciona con protocolos TCP y UDP. La otra versión se conoce como TIRPC (*Transport Independent RPC*) que se basa en el API TLI (*API Transport Layer Interface*) de AT&T y trabaja con cualquier protocolo de transporte.

Bajo el protocolo RPC, un programa puede ser identificado por medio de un entero de 32 bits y cada procedimiento de dicho programa con otro entero de 32 bits. Además, cada programa posee un número de versión para permitir la evolución de los mismos (incorporación de nuevas versiones sin dejar inhabilitadas las anteriores con el objeto de no causar molestias a los clientes que no se hayan actualizado).

En vista de lo anterior, para solicitar una llamada a un procedimiento remoto se necesita especificar:
- Número de Programa.
- Número de Versión.
- Número de Procedimiento.

En todo programa servidor, el procedimiento número cero (0) se encuentra reservado y se invoca con el único propósito de verificar si el servidor está disponible.

Los números de programa son enteros largos comprendidos entre los valores hexadecimales 00000000 y FFFFFFFF. Sin embargo, no todos estos valores pueden ser usados. Tomando como referencia la implementación de Sun Microsystems, los segmentos se distribuyen de la siguiente forma:

Rango en Hexadecimal	*Uso*
00000000 a 1FFFFFFF	Definidos por Sun
20000000 a 3FFFFFFF	A definir por los usuarios
40000000 a FFFFFFFF	Reservado

Tabla 15: Distribución de enteros de 32 bits para la asignación de números de programa según el estándar de Sun Microsystems.

13 Existen también otras implementaciones como por ejemplo Xerox Courier, que usa el protocolo SPP (*Sequence Packet Protocol*) de *Xerox Network System* (XNS).

3.1. El Proceso portmap

Dado que se hace necesario saber en cual puerto un servidor RPC se encontrará escuchando, se hace uso del proceso `portmap`, el cual asocia un número de puerto a un servicio RPC. `portmap` es en sí un proceso RPC cuyo número de programa es el **100000** y cuyo puerto es el **111**. Es necesario que este servicio se encuentre activo para que el mecanismo general RPC sea accesible, es decir, `portmap` debe estar funcionando antes de invocar cualquier servidor RPC. El funcionamiento de este proceso se describe a continuación:

1. Un proceso servidor solicita la dirección de escucha (número de puerto) a `portmap` y éste le asigna una.
2. El cliente, antes de solicitar un servicio, consulta `portmap` en la máquina que hace las veces de servidor para saber el número de puerto de servicio.
3. El cliente y el servidor establecen la comunicación usando los números de puerto provistos por `portmap`.

Un servicio RPC activo está disponible para cualquier cliente, previa indicación de disponibilidad al proceso `portmap` por parte del servidor. Este mecanismo se conoce como *grabación* del servicio.

Durante la secuencia de arranque, el proceso `portmap` entrará en funcionamiento antes que el proceso `inetd`, ya que hay servicios RPC que usan este proceso. El proceso `portmap` viene actualmente incluido en la gran mayoría, sino todas, las distribuciones de UNIX/LINUX.

3.2. El Comando rpcinfo

Este comando permite obtener información sobre los diferentes servidores RPC disponibles en una máquina específica. Las diferentes formas de uso de este comando se muestran a continuación (las expresiones entre corchetes son opcionales):

```
rpcinfo -p [máquina]
```

De esta forma es posible obtener la lista de todos los procesos servidores registrados sobre la máquina especificada, donde dicha máquina puede ser representada por su nombre o su dirección IP . Otra forma de usar este comando es:

```
rpcinfo -u máquina número_programa [versión]
```

De este modo se realiza una llamada al procedimiento número cero (0) del servicio u (UDP) cuyo número de programa es `número_programa` sobre la máquina especificada. También se tiene la forma:

```
rpcinfo -t máquina número_programa [versión]
```

Con este formato, se causa un efecto similar al caso anterior, con la diferencia de que se usa un servicio TCP.

3.3. El Archivo /etc/rpc

En este archivo se encuentra la lista de servicios RPC junto con su número de programa y se usa para averiguar dicho número de programa a partir del nombre del servicio o un alias del mismo. Este archivo contiene un solo servicio por línea, cada una de las cuales conteniendo la información siguiente:

- Nombre oficial.
- Número de programa.
- Lista de alias.

Un ejemplo del contenido de este archivo es el siguiente:

```
nfs          100003      nfsprog
ypserv       100004      ypprog
mountd       100005      mount showmount
ypbind       100007
walld        100008      rwall shutdown
yppasswdd    100009      yppasswd
etherstatd   100010      etherstat
rquotad      100011      rquotaprog quota rquota
```

Para acceder a la información de este archivo, existe la estructura `rpcent` y las funciones `getrpcbyname` y `getrpcbynumber` definidos en el archivo de cabecera `<rpc/netdb.h>`. Sus respectivos formatos son:

```
struct rpcent
{
   char *r_name;        /* Nombre del servidor para el programa RPC. */
   char **r_aliases;    /* lista de alias */
   int r_number;        /* Número de programa RPC */
};

struct rpcent *getrpcbyname (char *servicio)

struct rpcent *getrpcbynumber (int número)
```

3.4. Niveles de uso RPC

En RPC, pueden usarse 3 niveles diferentes a la hora de implementar programas cliente-servidor. cada uno de ellos ofrece diferentes grados de transparencia al programador en función de sus conocimientos y necesidades. Estos niveles son:

Nivel Alto: Es el que oculta más detalles al programador. Generalmente sólo se requiere una llamada a la función de biblioteca indicando el nombre de la máquina destino, así como otros parámetros requeridos por la respectiva función. Este nivel presenta la limitante de que el programador no puede desarrollar sus propios servicios, quedando restringido a los que se tienen disponibles.

Nivel Medio: Evita al programador el trabajar con sockets y/o XDR. Sin embargo, tiene la desventaja de que solamente se puede utilizar el protocolo UDP. Tampoco puede cambiarse el valor por defecto de algunos parámetros. Este nivel exige conocimientos mínimos de los protocolos RPC y XDR. Es en este nivel donde se desarrollan la mayoría de aplicaciones RPC.

Nivel Bajo: En este nivel es donde la programación se vuelve más compleja, pero a su vez permite el mayor grado de libertad haciendo posible el desarrollo de programas tanto en TCP como UDP, el cambio de los valores por defecto de todos los parámetros, etc.

3.4.1. Nivel Medio

Permite escribir servicios de forma simple apoyándose solamente en el protocolo UDP, por lo que se limita el tamaño de los mensajes intercambiados (a 8 Kb). Las principales operaciones que se deben realizar en una aplicación cliente-servidor usando este nivel son:

En el servidor:

1. Escribir los procedimientos del servicio.
2. Asignar un número de programa y versión.
3. Registrar los procedimientos del programa servidor en el demonio `portmap` usando la función `registerrpc`.
4. Esperar peticiones de los clientes por medio de `svc_run`.

En el cliente:

1. Realizar llamadas usando la función `callrpc`.

En una llamada a una función local, los argumentos se pasan de forma individual. Para una llamada remota, los argumentos deben estar integrados en una sola estructura, cuyo puntero se pasa como único argumento. La razón de esta limitante se encuentra en XDR donde se usan punteros a estructuras de datos en las funciones usadas para cambiar formatos. Debido al mismo motivo, el resultado devuelto será también un puntero donde se almacenan los resultados.

Cuando se hacen llamadas en el servidor, suele necesitarse al menos 2 llamadas a cada función para hacerlas accesibles al cliente: una llamada para registrarlos en `portmap` y otra para bloquear a la espera de peticiones.

3.4.1.1. Registro de Procedimientos

Mediante la función `registerrpc` se le indica al demonio `portmap` la existencia de un procedimiento. Cada función debe agregarse a la lista de `portmap` de forma individual. Dicha función se encuentra definida en el archivo de cabecera `<rpc/rpc.h>` y su formato es:

```
int registerrpc   (unsigned long numprog, unsigned long numvers,
                   unsigned long numproc, char *(*nombreproc),
                   xdrproc_t inproc, xdrproc_t outproc);
```

Valor Devuelto: 0 en caso de que el registro de la función tenga éxito y -1 en caso de falla.

numprog: Número de programa correspondiente al procedimiento a registrar.

numvers: Número de versión.

numproc: Número de procedimiento.

nombreproc: Nombre del procedimiento.

inproc: Función filtro a utilizar para decodificar los datos captados por la función.

outproc: Función filtro a utilizar para codificar los datos obtenidos como resultados de la función.

Los procedimientos remotos registrados por medio de esta función son accesados por medio del protocolo UDP

Una vez registrado, cualquier cliente puede llamar a este procedimiento consultando a `portmap` a través del número de procedimiento, de versión y de programa. La respuesta de `portmap` se dará en el puerto por donde se está escuchando.

3.4.1.2. Espera de Solicitudes

Una vez que el servidor registra sus procedimientos por medio de `registerrpc`, se usa la función `svc_run` la cual queda a la escucha en un socket en espera de alguna llamada. Al llegar una petición de ejecución, la función `svc_run` despierta para atender la petición, volviendo luego a su estado de espera (dormido). Esta función no devuelve nada, es decir, el programa se bloquea a menos que ocurra un error en la llamada, devolviendo el control. Dicha función se encuentra definida en el archivo de cabecera `<rpc/rpc.h>` y su formato es:

```
void svc_run ()
```

Lo que esta función hace en realidad es llamar a `select` para leer el descriptor del socket donde se recibirán todas las peticiones para ejecutar algún procedimiento. Al recibir recibir una petición, analiza el número de procedimiento, ejecuta el procedimiento correspondiente, devuelve los resultados y ejecuta nuevamente `select`.

3.4.1.3. Llamada al Procedimiento Remoto

La llamada para solicitar la ejecución de un procedimiento se lleva a cabo del lado del cliente. Para ello se usa la función `callrpc`. Dicha función se encuentra definida en el archivo de cabecera `<rpc/rpc.h>` y su formato es:

```
int callrpc (char *máquina, unsigned long numprog, unsigned long numvers,
             unsigned long numproc, bool_t (*inproc)(), char *in,
             bool_t (*outproc)(), char *out)
```

Valor Devuelto: 0 en caso de éxito, -1 en caso de fallo

máquina: Ordenador en el que se encuentra el servidor.

numprog: Número de programa.

numvers: Número de versión.

numproc: Número de procedimiento.

inproc: Puntero que indica el filtro que se usará para la conversión (decodificación) de los datos de entrada.

in: Puntero a los datos de entrada a la función.

outproc: Puntero que indica el filtro que se usará para la codificación de los datos de salida.

out: Puntero a los datos de salida de la función.

La llamada a esta función es bloqueante. Utiliza protocolo UDP además de los tiempos de espera y validaciones por defecto. El programador no es capaz de cambiar ninguno de estos parámetros a menos que use el nivel bajo de programación.

3.4.2. Nivel Bajo

En este nivel no existen restricciones de programación, esto a costa de una mayor complejidad para el programador. Usualmente este nivel de programación no suele usarse a menos que se necesite de alguna opción no disponible en el nivel intermedio.

Existen 2 posibilidades para la programación en Este nivel: hacer directamente la llamada a las funciones que implementan los cabos o usar el compilador `rpcgen` para generar dichas funciones, lo que reduce el número de llamadas a efectuar, pero complica la programación del cliente.

3.4.2.1. Elección del Protocolo de Transporte

Para elegir el protocolo de capa 4 (nivel de transporte) debe considerarse lo siguiente:

Cuando se usa UDP no se está seguro si se ejecutó o no el procedimiento remoto. En estos casos donde no se recibe respuesta suelen hacerse reenvíos periódicos durante cierto tiempo. Si no se recibe respuesta, no sabemos si el procedimiento se ejecutó o no. Si se recibe una respuesta, se puede asegurar que el procedimiento se ejecutó al menos una vez.

En vista de lo anterior, si usamos UDP debemos estar seguros que el proceso servidor pueda procesar varias veces la petición sin problemas. Si no es así, deben evitarse los reenvíos.

Al usar TCP, si se recibe una respuesta, el procedimiento se ejecutó exactamente una vez. Caso contrario no podemos asegurar si el procedimiento se ejecutó o no.

Usando UDP, la longitud máxima por mensaje no puede superar los 8 Kb.

UDP es más rápido que TCP ya que UDP no realiza pasos para el establecimiento de una conexión .

Si las aplicaciones se ejecutarán en un ambiente seguro, como una red LAN, lo mejor es usar UDP. Si el medio no es muy confiable, como en una red WAN, se recomienda TCP.

3.4.2.2. Llamadas en el Servidor

La principal diferencia en el código desde el punto de vista del servidor es es la necesidad de re-escribir todos los procedimientos bajo una única función servidora, la cual se invocará al llegar cualquier petición. Esta función analizará de qué petición se trata y ejecutará el procedimiento apropiado.

Los pasos a seguir en la función principal del servidor son:

1. Crear un descriptor de transporte.
2. Registrar solamente la función servidora que agrupa todos los procedimientos en el demonio `portmap`.
3. Esperar peticiones con `svc_run`.

Asimismo, dentro de la función servidora se deben seguir los siguientes pasos:

1. Analizar la petición con `switch`.
2. leer los datos con `svc_getargs`.
3. Ejecutar el procedimiento.
4. Devolver los resultados con `svc_sendreply`.
5. Regresar a la espera de peticiones con `return`.

3.4.2.3. Creación de un Descriptor de Transporte

Consiste en crear un socket, enlazarlo y crear una estructura especial donde se anotará las características de transporte deseadas. Esta estructura es del tipo SVCXPRT, definida en <rpc/rpc.h> y <rpc/svc.h>[14]. Esta estructura tiene el siguiente formato:

```
typedef struct

{

int xp_sock;                          /* Descriptor del socket de servicio*/
u_short xp_port;                      /* Número de puerto asociado */


const struct xp_ops
     {

     bool_t (*xp_recv) (SVCXPRT *xprt, struct rpc_msg *msg);
     /* Recibe solicitudes entrantes */

     enum xprt_stat (*xp_stat) (SVCXPRT *xprt);
     /* Obtener estado del puerto*/

     bool_t (*xp_getargs) (SVCXPRT *xprt, xdrproc_t xdr_args, ddr_t args_ptr);
     /* Obtener argumentos */

     bool_t (*xp_reply) (SVCXPRT *xprt, struct rpc_msg *msg);
     /* Enviar respuesta */

     bool_t (*xp_freeargs) (SVCXPRT *xprt, xdrproc_t xdr_args, caddr_t args_ptr);
     /* Liberar memoria reservada por argumentos */

     void (*xp_destroy) (SVCXPRT *xprt);
     /* Destruir estructura SVCXPRT */

     } *xp_ops;

int   xp_addrlen;                     /* Longitud de dirección remota */
struct sockaddr_in xp_raddr;          /* Dirección remota */
struct opaque_auth xp_verf;           /* Verificador de respuesta */
caddr_t       xp_p1;                  /* Privado */
caddr_t        xp_p2;                 /* Privado */

} SVCXPRT;
```

Para registrar la unción servidora, se necesita de un objeto SVCXPRT además de un socket enlazado. Existen 2 funciones que permiten crear objetos tipo SVCXPRT, encargándose también de crear y enlazar el socket correspondiente. Estas funciones se encuentran definidas en los archivos de cabecera <rpc/rpc.h> y <rpc/svc.h> son las siguientes:

```
SVCXPRT *svcudp_create (int sockfd);
SVCXPRT *svctcp_create (int sockfd, u_int tamaño_env, u_int tamaño_rec);
```

sockfd: Descriptor de socket. Si el socket no ha sido creado al momento de llamar a esta función (lo cual es usualmente el caso) suele usarse la constante RPC_ANYSOCK en este parámetro.

14 Al momento de programar, lo más recomendable es incluir el archivo de cabecera <rpc/rpc.h> ya que éste incluye a <rpc/svc.h> así como a muchos otros archivos de cabecera

tamaño_env: Tamaño del buffer de transmisión. Un valor de cero especifica el tamaño por defecto.

tamaño_rec: Tamaño del Buffer de recepción. Un valor de cero especifica el tamaño por defecto.

Estas funciones se usan para crear los objetos SVCPRT y enlazarlos a sockets UDP o TCP respectivamente.

3.4.2.4. Registro de Servicios y Espera de Peticiones

Para relacionar un número de programa y versión con un puerto determinado, comunicando a su vez dicha asociación al demonio `portmap`, usamos la función `svc_register`, definida en los archivos de cabecera `<rpc/svc.h>` y `<rpc/rpc.h>`. Su formato es:

```
bool_t svc_register (SVCXPRT *xprt, u_long programa, u_long version,
                     dispatch_fn_t f_servidora, u_long protocolo));
```

Valor Devuelto: VERDADERO en caso de éxito, FALSO si falla.

xptr: Puntero a estructura SVCXPRT que representa el manejador de servicio de transporte RPC.

programa: Número de programa.

version: Número de versión.

f_servidora: Función servidora, la cual debe tener la forma siguiente:

```
f_servidora(struct svc_req *solicitud, SVCXPRT *xprt)
```

Una explicación más amplia de esta función se encuentra en el siguiente apartado.

protocolo: Especifica el protocolo de transporte con el cual queda registrado en `portmap`. Sus valores más comunes son:

- IPPROTO_UDP (protocolo UDP).
- IPPROTO_TCP (protocolo TCP).
- 0 (No registrar en el demonio `portmap`)

3.4.2.5. La Función Servidora

Esta función, definida por el usuario, agrupa todos los procedimientos y será llamada cuando llegue cualquier petición. Como se indicó anteriormente, su formato es:

```
f_servidora(struct svc_req *solicitud, SVCXPRT *xprt)
```

Por supuesto, la función servidora puede tomar cualquier nombre además de `f_servidora`, lo mismo aplica para los nombres de los argumentos.

El primero de los argumentos, un puntero a una estructura `svc_req` se refiere a la información sobre los procedimientos a ejecutar. El segundo argumento es un puntero a un descriptor de transporte SVCXPRT.

La estructura `svc_req` (service request) está definida en los archivos de cabecera `<rpc/svc.h>` y `<rpc/rpc.h>`. Su formato es:

```
struct svc_req
{
        u_long              rq_prog;        /* Número de programa */
        u_long              rq_vers;        /* Número de versión */
        u_long              rq_proc;        /* Procedimientos deseado */
        struct opaque_auth  rq_cred;        /* Credenciales totales */
        caddr_t             rq_clntcred;    /* Sólo de lectura */
        SVCXPRT             *rq_xprt;       /* Transporte asociado */
};
```

El campo más interesante es `rq_proc`, que se refiere al procedimiento a ejecutar. Lo usual es analizar los posibles valores de este campo con `switch` y según su valor, ejecutar un procedimiento dado. Además de los valores para los cuales el usuario tiene definido un procedimiento, deben considerarse otros casos como:

- El procedimiento 0 (NULLPROC), que no devuelve ningún valor. De este modo el cliente sabe que así servidor está esperando peticiones sin ejecutar ninguna función en especial. Después suele volverse a la escucha con `return`.
- El procedimiento encargado de tratar los casos en los cuales se ingresa un número de procedimiento erróneo (el cual debería ser el caso por defecto). Suele usarse la función `scverr_noproc`, regresando después a la escucha con `return`.

Una vez analizado el procedimiento a ejecutar, se deben leer los datos enviados por el cliente con la macro `svc_getargs`, definida en <rpc/svc.h> y <rpc/rpc.h>. Su formato es:

```
bool_t svc_getargs(SVCXPRT *xprt, xdrproc_t xargs, char *args)
```

Valor Devuelto: VERDADERO en caso de éxito, FALSO si falla.

xptr: Puntero a estructura SVCXPRT que representa al manejador del servicio de transporte RPC.

xargs: Rutina XDR usada para decodificar los datos extraídos del flujo y enviados por el cliente.

args: Dirección en la que se ponen los datos decodificados por la función `xargs`.

A continuación se ejecuta el procedimiento con los datos del cliente para después enviar los resultados de vuelta al cliente por medio de `svc_sendreply`. Esta macro está definida en <rpc/svc.h> y <rpc/rpc.h>. Su formato es:

```
bool_t svc_reply(SVCXPRT *xprt, xdrproc_t outproc, char *out)
```

Valor Devuelto: VERDADERO en caso de éxito, FALSO si falla.

xptr: Puntero a estructura SVCXPRT que representa al manejador del servicio de transporte RPC. Los resultados codificados en formato de red se transmiten por el puerto de servicio designado en esta estructura.

outproc: Rutina XDR para codificar los resultados a formato de red.

out: Dirección en la que se encuentran los resultados a codificar por `outproc`.

El último paso a incluir en la función servidora es el regreso al estado de escucha por medio de `return`.

3.4.2.6. Llamadas en el Cliente

En el cliente se hacen como mínimo 2 llamadas en lugar de una: la primera para buscar la dirección del servidor y la segunda para ejecutar el procedimiento remoto. Con esto se tiene la ventaja de que en futuras ejecuciones sólo se necesita de la segunda llamada puesto que la dirección ya es conocida para ese entonces. Con esto se evita tráfico innecesario en la red.

Hay 2 tipos de llamadas para buscar la dirección del servidor según el transporte escogido. Ambas crean y asocian un socket, iniciando además la estructura `sockaddr_in` con el puerto y la dirección IP del servidor. Dichas llamadas devuelven un puntero a una estructura tipo CLIENT, definida en `<rpc/clnt.h>`. Su formato es:

```
typedef struct CLIENT
{
      AUTH *cl_auth;          /* Autentificador */
      struct clnt_ops
      {
      enum clnt_stat (*cl_call) ();       /* Llamada al procedimiento remoto */
      void (*cl_abort) ();                /* Abortar llamada */
      void (*cl_geterr) ();               /* Obtener código específico de error */
      bool_t (*cl_freeres) ();            /* Liberar resultados */
      void (*cl_destroy) ();              /* Destruir esta estructura */
      bool_t (*cl_control) ();            /* El equivalente a ioctl() para RPC */
      } *cl_ops;
      caddr_t cl_private;       /* privado */
};
```

NOTA: No se muestran los argumentos de las funciones de la estructura arriba descrita.

La función para buscar la dirección del servidor usando el protocolo UDP es:

```
CLIENT *clntudp_create (struct sockaddr_in *máquina, unsigned long programa,
                        unsigned long versión, struct timeval tiempo,
                        int *sockfd)
```

máquina: Dirección IP de la máquina en la que se encuentra el programa remoto (la función servidora). Si `máquina->sin_port = 0`, se establece como puerto aquel que la función remota está usando para escuchar peticiones (para ello se consulta a `portmap`).

programa: Número de programa.

versión: Número de versión.

tiempo: Especifica el intervalo de tiempo que la función debe esperar para el reenvío de peticiones hasta que se reciba una respuesta del servidor o hasta que el tiempo de espera se agote. Dicho tiempo de espera es definido por `clnt_call`.

sockfd: Descriptor de socket. Si el socket no ha sido creado al momento de llamar a esta función (lo cual es usualmente el caso) suele usarse la constante RPC_ANYSOCK en este parámetro.

Con esta función se crea un cliente para el programa `programa`, con la versión `versión` en la dirección `máquina`, usando el descriptor `sockfd`.

NOTA: Recuérdese que con UDP el tamaño máximo de mensajes que puede ser enviado es de 8 Kb.

La función para buscar la dirección del servidor usando el protocolo TCP es:

```
CLIENT *clnttcp_create (struct sockaddr_in *máquina, unsigned long programa,
                        unsigned long versión, int *sockfd,
                        unsigned int tamaño_env, unsigned int tamaño_rec)
```

máquina: Dirección IP de la máquina en la que se encuentra la el programa remoto (la función servidora). Si `máquina->sin_port = 0`, se establece como puerto aquel que la función remota está usando para escuchar peticiones (para ello se consulta a `portmap`).

programa: Número de programa.

versión: Número de versión.

sockfd: Descriptor de socket. Si el socket no ha sido creado al momento de llamar a esta función (lo cual es usualmente el caso) suele usarse la constante RPC_ANYSOCK en este parámetro.

tamaño_env: Tamaño del buffer de transmisión. Un valor nulo (NULL, 0) especifica el tamaño por defecto.

tamaño_rec: Tamaño del Buffer de recepción. Un valor nulo (NULL, 0) especifica el tamaño por defecto.

Una vez creado el cliente, se procede a la llamada al procedimiento remoto por medio de la función `clnt_call`, definida en el archivo de cabecera `<rpc/clnt.h>` y cuyo formato es:

```
enum clnt_stat clnt_call (CLIENT *cliente, unsigned long proc,
                          xdrproc_t xargs, caddr_t args, xdrproc_t xres,
                          caddr_t res, struct timeval espera)
```

Valor Devuelto: cero (RPC_SUCCESS) en caso de éxito. Un valor diferente de cero en caso de fallo.

cliente: Manejador del cliente.

proc: Número de procedimiento.

xargs: Rutina XDR usada para decodificar los datos extraídos del flujo y enviados por el cliente.

args: Dirección en la que se ponen los datos decodificados por la función `xargs`.

xres: Rutina XDR usada para codificar los datos devueltos por el procedimiento con número `proc`.

res: Dirección de la que se toman los datos a codificar por la función `xres`.

espera: Proporciona el tiempo de espera máximo por un resultado. Si durante el tiempo especificado en este parámetro no hay respuesta del servidor, se asume la existencia de un fallo.

El tipo de datos numerado `clnt_stat`[15] devuelto por la función `clnt_call` se encuentra definido en `<rpc/clnt.h>`.

Finalmente, cuando el cliente finaliza sus llamadas al servidor, debe eliminar la estructura CLIENT por medio de la función `clnt_destroy` definida en el archivo de cabecera `<rpc/clnt.h>` y cuyo formato es:

```
void clnt_destroy (CLIENT *cliente)
```

15 La definición del tipo de datos enumerado `clnt_stat` se encuentra en el archivo de cabecera `<rpc/clnt.h>`.

3.4.3. Programación en El Servidor

Del lado del servidor, solamente es necesario escribir el programa *programa*`_server.c` que contiene los procedimientos. No es necesario llamar a ninguna función RPC ya que todas las funciones necesarias están ocultas en el cabo servidor (*programa*`_svc.c`)

El compilador `rpcgen` genera por defecto los cabos para servidores con UDP y TCP a través, de las llamadas `svcudp_create` y `svctcp_create`.

Al momento de escribir los procedimientos debe declararse la cadena de caracteres (donde se pondrá el resultado) en una zona estática de memoria. Esto se hace para evitar la destrucción de los datos cuando el cabo termine su ejecución. Si el procedimiento necesita memoria, no se debe olvidar liberar la memoria utilizada posteriormente por medio de la llamada a la función `xdr_free`. El formato de dicha función es el siguiente:

```
void xdr_free (xdrproc_t proc, char* datos);
```

proc: Rutina XDR (filtro) correspondiente al procedimiento que será liberado.

datos: Puntero al array de caracteres de salida.

El nombre del procedimiento es el mismo que aparece en la definición ,seguido del símbolo "_" y del número de versión.

3.4.4. Programación en el Cliente

Del lado del cliente es necesario realizar 2 pasos:

1. Para cada servidor con el que se interactúe es necesario inicializar una estructura tipo CLIENT mediante la llamada a `clntudp_create` o `clnttcp_create`.

2. Hacer la llamada al procedimiento remoto como si fuese local, es decir, escribiendo directamente el nombre del procedimiento.

No se debe olvidar que RPC trabaja con punteros a estructuras, los parámetros se pasan con un puntero y los resultados son punteros a cadenas de caracteres.

3.5. Programación en Nivel Bajo Usando el Compilador rpcgen

Usando el compilador `rpcgen` se simplifica la programación en el nivel bajo. Lo que se hace es definir las estructuras a emplear en lenguaje RPCL (RPC Language) así como el programa, versión y procedimiento. Estas especificaciones son procesadas por `rpcgen` el cual genera los archivos de cabecera, filtros y cabos del cliente y servidor. En los cabos se ocultan todas las llamadas y pasos del nivel bajo, por lo que sólo se necesita escribir los procedimientos. Respecto al cliente sólo es necesario averiguar el número de puerto del servidor y llamar directamente a cada procedimiento las veces que se requiera.

El primer paso consiste en escribir la especificación del programa a desarrollar en RPCL, la cual debe contener los nombres de los programas, versiones y procedimientos, así como las estructuras de datos que intercambiarán cliente y servidor. Todo esto se incluye en un fichero de texto con el nombre de la aplicación y extensión `.x` (es la extensión más usada, aunque nada impide que use alguna otra. Sin embargo se recomienda esta extensión para efectos de orden). Luego, se compila este archivo con `rpcgen` de la siguiente forma:

```
rpcgen programa.x
```

Al ejecutar el comando anterior, son generados los siguientes archivos:

- ***programa*.h** (Archivos de cabecera)
- ***programa*_xdr.c** (Filtros de Codificación y decodificación).
- ***programa*_svc.c** (Cabo servidor)
- ***programa*_clnt.c** (Cabo cliente)

Es responsabilidad del programador la creación de los archivos fuente del cliente donde se hacen las llamadas a los procedimientos remotos, así como los archivos fuente del servidor donde se escribirán dichos procedimientos. Estos ficheros pueden tener cualquier nombre, pero para mantener la correlación con los archivos creados con `rpcgen`, se le suele dar nombres con el formato siguiente:

- ***programa*_client.c** (Archivo fuente del cliente)
- ***programa*_server.c** (Archivo con los procedimientos a ejecutar en el servidor)

Las versiones más recientes de `rpcgen` soportan la opción `-a`, con la que se generan automáticamente los esqueletos de estos dos últimos archivos.

```
rpcgen -a programa.x
```

Al momento de compilar y enlazar se ejecutarían instrucciones como las siguientes (usando el compilador `gcc` y asumiendo que los archivos ejecutables llevarán los nombres `cliente` y `servidor` respectivamente):

Para compilar:

```
gcc -c programa_client.c programa_clnt.c programa_xdr.c
gcc -c programa_server.c programa_svc.c programa_xdr.c
```

El ejecutar esas instrucciones solamente compila los archivos involucrados (esto se logra especificando la opción `-c`: compilar), generando archivos con extensión `.o`. Estos mismos archivos sirven como parámetros de entrada para en enlazado, que se realiza digitando:

```
gcc -o cliente programa_client.o programa_clnt.o programa_xdr.o
gcc -o servidor programa_server.o programa_svc.o programa_xdr.o
```

Con estas instrucciones se enlazan los archivos generados anteriormente. El nombre de los archivos ejecutables se especifica con la opción `-o` seguida del nombre que se desea tengan dichos archivos.

Debe hacerse notar que este proceso es válido para compilar y enlazar cualquier programa escrito en C y no solamente los que se generan con la ayuda de RPC.

3.5.1. El Lenguaje de Descripción de Protocolo RPCL (RPC Language)

El lenguaje usado para escribir el archivo `programa.x` (RPCL) es una extensión del lenguaje de definición XDR, el cual sirve para definir las estructuras de datos a intercambiar entre cliente y servidor. Este lenguaje consiste de una serie de definiciones de la forma:

```
lista_de_definiciones:
      definicion_1;
      definicion_2;
      ...
      definicion_n;
      nombre_de_lista
```

Las definiciones contempladas por este lenguaje son:

3.5.1.1. Constantes.

Su definición es la siguiente:

```
const identificador = entero
```

Un ejemplo de definición de constantes es la siguiente:

```
const MAXIMO = 12;
```

Estas definiciones pasarán a formar parte del archivo `programa.h` de la forma:

```
#define identificador entero
```

Para el caso del ejemplo:

```
#define MAXIMO 12
```

3.5.1.2. Enumeraciones

Su definición es la siguiente:

```
enum identificador
{ valores_enumerados }
```

La lista de valores enumerados deberán separarse por comas. La asignación de cada valor en la lista se hará por medio del signo igual (=). Ejemplo:

```
enum color
{ ROJO = 0, AMARILLO = 0, AZUL = 2 };
```

Esta enumeración, una vez compilada en `rpcgen`, formará parte de `programa.h` de la forma:

```
enum identificador
{ valores_enumerados };
typedef enum identificador identificador;
bool_t xdr_identificador ();
```

Para el caso del ejemplo:

```
enum color
{ ROJO = 0, AMARILLO = 0, AZUL = 2 };
typedef enum color color;
bool_t xdr_color ();
```

3.5.1.3. Estructuras

La definición para este tipo de datos es:

```
struct identificador
{ lista_de_declaraciones };
```

Cada elemento de la lista se separa por punto y coma (`;`). La estructura finaliza también con punto y coma. Ejemplo:

```
struct coordenada
{     int x ;
      int y ;
} ;
```

Al ser compilado en `rpcgen` e incorporada en `programa.h`, se vuelve de la forma:

```
struct identificador
{ lista_de_declaraciones };
typedef struct identificador identificador;
bool_t xdr_identificador ();
```

Lo que para el ejemplo sería:

```
struct coordenada
{     int x ;
      int y ;
} ;
typedef struct coordenada coordenada;
bool_t xdr_coordenada ();
```

3.5.1.4. Uniones

La definición para este tipo de datos es:

```
union identificador switch (variable_entera)
{     case valor_1: declaración_1;
      case valor_2: declaración_2;
      ...
      default: declaración_por_defecto;
};
```

Un ejemplo de lo anterior es:

```
union resultado switch (int valor)
{     case 0:
            opaque datos [128];
      default:
            void;
};
```

Dicha declaración pasa a formar parte de `programa.h` tomado la forma siguiente:

```
struct identificador
{     int variable_entera;
      union
      {     declaración_1;
            declaración_2;
            ...
            declaración_por_defecto;
      }
      identificador_u;
};
typedef struct identificador identificador;
bool_t xdr_identificador ();
```

Para el caso del ejemplo:

```
struct resultado
{     int valor;
      union
      {     char datos [128];
      }
      resultado_u;
};
typedef struct resultado resultado;
bool_t xdr_resultado ();
```

3.5.1.5. Definición de Tipos (typedef)

Su definición es la siguiente:

```
typedef declaración;
```

Un ejemplo de lo anterior es lo siguiente:

```
typedef punto coord[4];
```

Este tipo de definiciones aparecerán de forma idéntica en el archivo de cabecera `programa.h`, salvo por los tipos de datos que representan casos especiales y se definen posteriormente.

3.5.1.6. Programas

Los nombres de programas, versiones y procedimientos suelen escribirse en mayúsculas. Su definición es:

```
program programa
{     versión:
            identificador_de_versión_1
            {     tipo_identificador proc_1-A (tipo_identificador) = valor;
                  tipo_identificador proc_1-B (tipo_identificador) = valor;
                  ...
            } = valor;

            identificador de versión_2
            {     tipo_identificador proc_2-A (tipo_identificador) = valor;
                  tipo_identificador proc_2-B (tipo_identificador) = valor;
                  ...
            } = valor;
            ...
} = valor;
```

Ejemplo:

```
program ACCION
{     version PRELIMINAR
      {     void SUBIR (void) = 1;
            int BAJAR (void)= 2;
      } = 4;
} = 20000;
```

Esta definición pasaría a formar parte del archivo de cabecera `programa.h` tomando la forma siguiente:

```
#define programa                          valor
#define versión                           valor
#define identificador_de_versión_1        valor
#define proc_1-A                          valor
#define proc_1-B                          valor
...
#define identificador_de_versión_1        valor
#define proc_2-A                          valor
#define proc_2-B                          valor
...

extern *tipo_identificador proc_1-A (tipo_identificador);
extern *tipo_identificador proc_1-B (tipo_identificador);
...
extern *tipo_identificador proc_2-B (tipo_identificador);
extern *tipo_identificador proc_2-B (tipo_identificador);
...
```

Lo que para el ejemplo sería:

```
#define ACCION      20000
#define PRELIMINAR  1
#define SUBIR       1
#define BAJAR       2

extern void *subir_4();
extern void *bajar_4 ();
```

El procedimiento número cero (NULLPROC o procedimiento nulo) no devuelve ningún valor. Nótese que los procedimientos pasan a llamarse de la misma forma que la definición, agregándoseles al final el carácter "_" seguido del número de versión. Esto debe tomarse el cuenta al momento de hacer las llamadas en el cliente.

3.5.2. Declaraciones

Existen 4 tipos de declaraciones en RPCL:

3.5.2.1. Simple

Su formato es.

```
tipo_identificador variable;
```

Un ejemplo de este tipo de declaración es:

```
color c;
```

3.5.2.2. Arreglos de Longitud Fija

Su formato es.

```
tipo_identificador [valor];
```

Un ejemplo de este tipo de declaración es:

```
hoja libro [100];
```

3.5.2.3. Arreglos de Longitud Variable

Su formato es.

```
tipo_identificador variable <valor>;
tipo_identificador variable < >;
```

Un ejemplo de este tipo de declaración es:

```
int x <25>;              /* A lo sumo 25 elementos */
float y < >;             /* cualquier cantidad de elementos */
```

Estas definiciones, al compilarse en `rpcgen`, pasarían a tomar la forma descrita a continuación (esto se debe a que no existe en C una definición para los arreglos de longitud variable):

```
struct
{     u_int variable_len;
      tipo_identificador *variable_val;
};
```

Para el caso del primero de los ejemplos:

```
struct
{     u_int x_len;
      int *x_val;
};
```

3.5.2.4. Punteros

Son declarados de la misma forma que en lenguaje C. Siguen el formato siguiente:

```
tipo_identificador *variable;
```

un ejemplo es:

```
int *número;
```

3.5.3. Casos Especiales

Existen algunos tipos de datos que sufren ciertas transformaciones al pasar por el compilador `rpcgen`. estos son:

3.5.3.1. Booleanos

Son declarados de la manera siguiente:

```
bool variable;
```

Al ser procesados por rpcgen, toman la forma:

```
bool_t variable;
```

3.5.3.2. Cadenas

Su declaración es la siguiente:

```
string variable <longitud>;

string variable < >;
```

Al pasar por rpcgen, se convierten respectivamente en:

```
char variable [longitud];

char *variable;
```

3.5.3.3. Datos Opacos

Su declaración tiene las formas:

```
opaque variable [longitud];   /* Datos de longitud fija */

opaque variable <longitud>;   /* Datos de longitud variable */
```

Los cuales son compilados como:

```
char variable[longitud];      /* Datos de longitud fija */

struct                        /* Datos de longitud variable */
{     u_int variable_len;
      char *variable_val;
};
```

3.5.3.4. Void

En una declaración de este tipo, la variable no es nombrada. La declaración es simplemente void. Este tipo de declaraciones pueden ocurrir como el argumento o resultado de un procedimiento remoto solamente si se está dentro de una definición tipo union o program.

CONCLUSIONES DEL CAPITULO III

- La principal ventaja de trabajar usando RPC es que las llamadas a sockets y XDR quedan ocultas, obviando muchos pasos que podrían resultar engorrosos, permitiendo al programador concentrarse más en la aplicación en lugar de los detalles de programación. Gracias a ello se vuelve más fácil la programación cliente-servidor.

- RPC permite tres niveles de programación, lo cual permite escoger la combinación sencillez-flexibilidad que mejor se adapte a las necesidades de la aplicación.

- El nivel medio es el más ampliamente usado debido a que requiere un mínimo conocimiento de los protocolos XDR y RPC. Su principal desventaja radica en que se limita al protocolo UDP, impidiendo además el cambio de los valores por defecto de ciertos parámetros. Sólo es recomendable el empleo del nivel bajo si se requiere de alguna característica no contemplada por el nivel medio de programación, como el uso del protocolo TCP.

- El compilador `rpcgen` (usado en el nivel bajo) permite la creación automática de los procedimientos de conversión de datos (filtros), así como de los esqueletos de las aplicaciones cliente-servidor. Esto ahorra trabajo para el programador y reduce la posibilidad de errores en el código.

- A través del concepto de *número de versión* de RPC es posible mantener actualizada una aplicación, permitiendo a diferentes procesos cliente ser atendidos por los procesos servidor adecuados basándose en el respectivo número de versión para una aplicación dada.

REFERENCIAS BIBLIOGRAFICAS

[1] Jean-Marie Rifflet
"Comunicaciones en UNIX".
Editorial McGraw-Hill. 1992.
Primera Edición.

[2] "Communications Programming Concepts."
http://anguilla.u.arizona.edu/doc_link/en_US/a_doc_lib/aixprggd/progcomc

[3] Cisco Guides
http://www.cisco.com/univercd/cc/td/doc/product/software/ios390/ios390rp/

[4] Web Documents
http://webdocs.sequent.com/docs/rpcpxa01/

[5] RPC Programming Documents
http://webdocs.sequent.com/docs/rpcpxa01/book_toc.htm

CONCLUSIONES GENERALES Y RECOMENDACIONES

Para que una aplicación cliente-servidor sea completa,debe tener:
- ✔ Capacidad de comunicarse con cualquier ordenador sin importar su arquitectura.
- ✔ Capacidad de manejar la mayor cantidad posible de condiciones de error.
- ✔ Eficiencia en términos de longitud de código y cálculo.
- ✔ Claridad conceptual.
- ✔ Concordancia entre la finalidad de la aplicación y el modelo de servidor utilizado.

Debido a las limitaciones que actualmente posee el centro de cómputo de la EIE no fué posible cubrir temas como sockets de ruteo, broadcasting y multicastign. En el futuro, cuando se cuente con el equipo apropiado (switches, enrutadores, etc) será posible la puesta en marcha de aplicaciones que cubran dichos conceptos, tomando como base para una posterior investigación el presente documento. Por ello se hace imperativo, para el desarrollo posterior de los temas tratados un mejor equipamiento de la EIE.

En vista de que toda la red de la EIE está compuesta por ordenadores que utilizan la misma representación de datos, se hace poco evidente a nivel práctico el efecto de los programas creados usando el estándar XDR.

Para obtener un máximo aprovechamiento de los temas expuestos en este documento se vuelve imperante la necesidad de aumentar el nivel de conocimientos de programación en lenguaje C del estudiante ya que con esto no sólo se logrará una mejor asimilación de conceptos sino que, además, si se cuenta con herramientas complementarias como programación en ambiente visual (por ejemplo, Motiff, Qt, etc.) se pueden diseñar aplicaciones con utilidad no solo académica sino también comercial.

Para que una aplicación cliente-servidor sea portable, se vuelve necesario que sea capaz de transmitir información sin importar la representación de datos que utilicen los ordenadores sobre los que se ejecute. Para ello se debe hacer uso de las técnicas descritas en este documento.

En la creación de aplicaciones cliente-servidor, debe considerarse en primer lugar, el uso del protocolo RPC ya que presenta muchas facilidades al programador, usando el nivel medio como primera opción o el nivel bajo en los casos que lo requieran debido a las limitantes del nivel medio. Por otro lado, en los casos donde ni siquiera el nivel bajo de RPC puede adaptarse a las necesidades de la aplicación a desarrollar (como cuando se requieren capacidades que solamente ofrecen los sockets "crudos" u otra capacidad sólo ofrecida a nivel de socket) la mejor alternativa es la programación directa con sockets.

GLOSARIO

API: Programa de Interfaz de Aplicación (Aplication Program Interface).

ARP: Address Resolution Protocol. Protocolo de Resolución de Direcciones.

ARPA: Agencia de Proyectos de Investigación Avanzada (Advanced Research Proyects Agency). Conocida a partir de 1972 como DARPA (Defense Advanced Research Proyects Agency). Fundadora de ARPANET.

ARPANET: Red de la Agencia de Proyectos de Investigación Avanzada (Advanced Research Proyects Agency Network) creada en los años 70's por ARPA. Esta red conectaba varias entidades gubernamentales, laboratorios y universidades. Se volvió la columna vertebral de lo que ahora se conoce como Internet.

Asociación: Conjunto de coordenadas que identifican de forma única a un canal de comunicación entre dos procesos. Una asociación está compuesta por los siguientes componentes:

- Tipo de Protocolo.
- Dirección local.
- Proceso Local.
- Dirección remota.
- Proceso remoto.

BSD: Berkeley Software Distribution.

Cabo (Stub): Procedimiento a través de los cuales son ocultadas las llamadas a las funciones XDR.

CCITT: Consulttive Comitee for International Telephony and Telegraphy. Comité Consultivo para Telegrafía y Telefonía Internacional.

Datos Fuera de Banda: Se le llaman datos *fuera de banda* a aquellos que poseen un grado de prioridad superior al de los datos normales.

Demonio: Proceso que es ejecutado en segundo plano y es totalmente independiente del control de alguna terminal.

Dominio o Familia: Conjunto de protocolos de nivel 3 relacionadas con un socket. Ejemplo: La familia de protocolos AF_INET abarca a los protocolos IP, ICMP, IGMP, etc.

FQDN: Fully Qualified Domain Name. Nombre de Dominio Completamente Calificado.

Filtro: Función utilizada para codificar y decodificar los datos de una estructura, de formato interno a formato estándar de red y viceversa.

Flujo XDR: XDR Stream. Se le llama así a la zona de almacenamiento donde utilizada por un filtro para realizar las operaciones de conversión de formatos.

FTP: File Transfer Protocol. Protocolo de Transferencia de Archivos.

ICMP: Internet Control Message Protocol o Protocolo de Mensajes de Control de Internet.

IGMP: Internet Group Management Protocolo. Protocolo de Manejo de Grupos de Internet.

IPV4: Internet Protocol, versión 4. Es el formato mas usado actualmente para direcciones de internet. Dicho formato de disección tiene una longitud de 4 bytes y se representa de la forma xxx.xxx.xxx.xxx donde xxx es un número entero entre 0 y 255.

IPV6: Internet Protocol, versión 6. Es un nuevo formato para las direcciones de Internet, que eventualmente sustituirá al IPV4. Este formato de disección tiene una longitud de 16 bytes y se representa de la forma xxxx:xxxx:xxxx:xxxx:xxx:xxxx:xxxx:xxxx, donde xxxx es un número hexadecimal entre 0000 y FFFF.

ONC: Open Network Computing.

Proceso: Programa o parte de un programa.

RPC: Remote Procedure Call. Llamada a Procedimientos Remotos.

Socket: Punto de comunicación por medio del cual un proceso puede enviar o recibir información.

POSIX: Interfaz de Sistema Operativo Transportable (Portable Operating System Interface). Este estándar cubre específicamente las llamadas al sistema, bibliotecas, interfaz, verificación, prueba, utilidades de tiempo real y seguridad. El estándar POSIX cuenta con la aprobación del Instituto nacional de Estándares y Tecnologías (NIST: National Institute os Standards and Technology).

TCP: Protocolo de Control de Transmisión (Transmission Control Protocol).

XDR: External Data Representation. Representación Externa de Datos. Lenguaje desarrollado por Sun Microsystems utilizado actualmente para la conversión de representaciones de datos entre arquitecturas de ordenador diferentes.

ANEXOS

ANEXO A: GUIAS DE LABORATORIO

Guía de Laboratorio No. 1

Sockets Orientados a Conexión

Introducción

Esta guía introduce la descripción de la Interfaz de Programación de Aplicaciones (API) y los sockets, así como muchos conceptos básicos que son utilizados ampliamente dentro de la programación en entorno de Red.

La programación en Red involucra escribir programas que se comunican con otros a través de una red de computadoras, es decir, el desarrollo de aplicaciones distribuidas bajo el modelo de programación cliente-servidor.

Para escribir programas que corran en un entorno de Red se necesita una Interfaz de Programación de Aplicaciones o API (*Aplication Program Interface*). Una de las API de programación en red que se utilizan ampliamente son los ***sockets***, por tal motivo se hará énfasis en el estudio de éste.

NOTA: Se asume que el estudiante posee **SÓLIDOS CONOCIMIENTOS** del lenguaje C (estructuras, paso de argumentos, punteros y otros aspectos relacionados con la programación orientada a objetos) así como una regular experiencia trabajando en ambientes como UNIX/Linux. También son necesarios conocimientos elementales sobre comunicaciones de datos. Se recomienda que, para un mejor aprovecahmiento del tiempo asignado a la práctica, el estudiante digite los programas con anticipación. Esto ayudará también a plantear dudas que podrán ser aclaradas durante el desarrollo de la práctica.

¿Que son los Sockets?

Un *socket* es una interfaz de programación para el desarrollo de aplicaciones relativas a la computación distribuida. Específicamente, un *socket* es un descriptor de canal de comunicación que utiliza un proceso local (un programa ejecutándose en memoria) para la transmisión o recepción de datos con un proceso remoto a través de una red, ya sea esta una Red LAN o Red WAN.

La interfaz socket está pensada para trabajar de forma parecida a como se trabaja con un archivo. Por ejemplo, cuando se abre un socket, se devuelve un descriptor del mismo. Con este descriptor se pueden enviar o recibir datos de forma similar a como se leen o se escriben datos en un archivo.

Es importante saber que cuando un programa tipo UNIX/Linux desea realizar alguna operación de Entrada/Salida, lo realiza a través de una operación de lectura o escritura al respectivo descriptor.

Dominios y Estructuras de Dirección de Sockets

Muchas de las funciones para sockets requieren un puntero a una estructura de dirección de socket como argumento. Hay que mencionar que las estructuras que serán tratadas están orientadas a la versión 4 del protocolo IP (IPV4).

La siguiente estructura mantiene la información de referente a una estructura dirección de socket. Dicha definición se encuentra en el archivo `<sys/socket.h>` :

```
struct sockaddr
{
     unsigned short int    sa_family;
     unsigned char         sa_data[14];
};
```

El miembro `sa_family` puede tomar una variedad de valores, pero el valor que se le dará será "`AF_INET`" ya que se trabajará solamente con protocolos tipo Internet (IP). El miembro `sa_data` contiene una dirección destino y un número de puerto para el socket.

Esta estructutura es genérica. Las llamadas que utilizan el direccionamiento de un socket deben utilizar esta estructura. Cada tipo de socket utiliza una estructura de direccionamiento que no coincide necesariamente con esta estructura genérica, lo que obliga a hacer una conversión del puntero de la estructura especifica al tipo de estructura genérica.

Dominio Internet (AF_INET)

Este dominio corresponde a la versión 4 del protocolo IP. Los sockets bajo este dominio comunican dos procesos utilizando protocolos TCP/IP, con su correspondiente direccionamiento. Las estructuras utilizadas se encuentran definidas en el archivo `<netinet/in.h>`, las cuales son las siguientes:

```
struct in_addr
{
    unsigned long s_addr;              /*Dirección IP 32 bits, en formato de red */
};

struct sockaddr_in
{
    short int            sin_family    /*Familia de Dirección o Dominio */
    unsigned short int   sin_port;     /*Número de puerto */
    struct in_addr       sin_addr;     /*Dirección Internet  */
    unsigned char        sin_zero[8];  /*No usada, campo de 8 bytes de ceros*/
};
```

Esta estructura está pensada para hacer fácil el manejo de la información de la dirección socket. Se debe notar que `sin_zero` (el cual es incluído para concordar en tamaño con la estructura `sockaddr` genérica) debe ser puesta a cero.. También se debe notar que `sin_family` corresponde a `sa_family` en la estructura `sockaddr` y se le dará el valor de "`AF_INET`". Finalmente, `sin_port` y `sin_addr` deben estar representados en el mismo formato de ordenamiento con el que trabaja la red: *Network Byte Order* o representación de datos *Little Endian*.

Big Endian y Little Endian

Algunas computadoras almacenan los datos guardando primero el byte más significativo. Esto es lo que se llama el *estilo Big Endian* de representación de datos. Otras computadoras guardan primero el byte menos significativo, este es el *estilo Little Endian*.

De igual forma, existen normas de comunicaciones de datos Big Endian que transmiten los datos empezando con el byte más significativo.

Quienes escribieron las normas del protocolo de Internet eran Big Endian. Sin embargo, recuerde que existen otros grupos, como el IEEE, que transmiten los datos empezando con el byte menos significativo.

Para poder subsanar las diferencias que pudieran existir entre ordenadores con diferentes representación interna de datos, se han definido cuatro funciones estándar de conversión de formatos, que hacen conversiones del formato interno al de red, y viceversa:

```
unsigned long htonl  ( unsigned long x  )
unsigned short htons ( unsigned short x )
unsigned long ntohl  ( unsigned long x  )
unsigned short ntohs ( unsigned short x )
```

Estas funciones serán invocadas para convertir los valores de la cabecera de los paquetes (y que por tanto no son datos), como por ejemplo la dirección IP y el puerto, en el caso de dominio `AF_INET`.

Entradas al Sistema Orientadas a Conexión

Creación del Socket

Esta se lleva a cabo por medio de la función `socket`. El socket debe ser creado tanto en el cliente como en el servidor.

```
#include <sys/types.h>
#include <sys/socket.h>
int socket (int dominio , int tipo, int protocolo);
```

El argumento `dominio` debe ser "`AF_INET`", como en la estructura `sockaddr_in` . Después el argumento `tipo` le dice al kernel (el núcleo del sistema operativo) que tipo de socket se usará. Este puede ser SOCK_STREAM (TCP) o SOCK_DGRAM (UDP). Finalmente, el argumento `protocolo` es generalmente "0". Las tablas 1 y 2 muestran algunas de las constantes que se puede utilizar con los argumentos `dominio` y `tipo` respectivamente. En caso de error se devuelve -1, en caso de éxito un entero positivo.

Dominio	*Descripción*
AF_UNIX (Protocolos Internos UNIX)	Protocolos de dominio UNIX
AF_INET (Protocolos ARPA)	Protocolos IPV4
AF_ISO (Protocolos ISO)	Protocolos ISO

Tabla 1

Tipo	*Descripción*
SOCK_STREAM	Socket orientado a conexión (TCP)
SOCK_DGRAM	Socket no orientado a conexión (UDP)
SOCK_RAW	Permite el acceso a los protocolos de nivel inferior (IP, ICMP, etc)

Tabla 2

Enlace de un Socket

El siguiente paso después de crear un socket orientado a la conexión es enlazarlo por medio de la función `bind`. Este paso es obligatorio en el lado del servidor, pero opcional en el cliente, ya que si se decide no hacerse dicha llamada, el sistema operativo asocia un puerto a la respectiva dirección IP. Si est es el caso, la dirección escogida por el sistema operativo estará comprendida en el rango 1024-5000, que corresponde a los puertos asisgnados por el sistema.

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

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

`sockfd` es el descriptor de archivo del socket devuelto por `socket()`, `mi_dir` es un apuntador a `struct sockaddr` que contiene información acerca la dirección, puerto y la dirección IP, addrlen debe ser `sizeof(struct sockaddr)`. En caso de error se devuelve -1, en caso de éxito un entero positivo.

Establecimiento de la Conexión en el Cliente

Esta función es llamada únicamente en el cliente. Su trabajo es solicitarle al servidor una conexión. Un cliente orientado a la conexión no pedrá ser atendido mientras la conexión que solicite no sea aceptada por el servidor.

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

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

sockfd es el descriptor de archivo obtenido con la función socket(), serv_addr es una estructura a sockaddr conteniendo el puerto destino y la dirección IP destino, y addrlen puede ser puesto a sizeof(struct sockaddr). En caso de error se devuelve -1, en caso de éxito un entero positivo.

Espera de Conexión en el Servidor

Esta función es llamada únicamente en el servidor. Se usa para crear un cola de escucha, es decir, la máxima cantidad de clientes que el servidor puede tener en "espera" o que puede atender a la vez.

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

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

sockfd es el descriptor de archivo obtenido con la función socket(), es decir, el valor que dicha función devuelve. backlog es el número de conexiones permitidos en la cola de espera. En caso de error se devuelve -1, en caso de éxito un entero positivo.

Aceptación de la Conexión en el Servidor

Se llava a cabo por medio de la función accept. Esta función, que se ejecuta únicamente en el lado del servidor, es la encargada de aceptar las solicitudes de conexión efectuadas por los clientes a través de connect.

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

int accept (int sockfd, struct sockaddr *p_addr, int addlen);
```

accept extrae la primera conexión pendiente de la cola creada en el descriptor indicado sockfd, accept() crea un nuevo socket cuyo valor de descriptor es el entero devuelto por la llamada. Este socket esta conectado con un cliente cuya dirección será colocada por el sistema operativo en la estructura apuntada por p_addr. El tamaño de la dirección conectado es addlen. En caso de error se devuelve -1, en caso de éxito un entero positivo.

Supresión de un Socket

La llamada al sistema close() cierra un socket.

```
close(int sockfd );
```

En protocolos orientados a conexión, el sistema devuelve el control inmediatamente, pero el kernel continua enviando datos que estén todavía en cola.

Envío y Recepción de Datos Orientados a la Conexión

Una vez se tiene realizada la conexión entre el cliente y el servidor los dos procesos podrán intercambiar información.

Para recepción podemos utilizar la llamada read():

```
int read(int sockfd, char *mesg, int len);
```

Donde:

```
int sockfd  /*descriptor del socket local*/
char *mesg  /*dirección del buffer de recepciones */
int len     /*longitud del buffer de recepciones */
```

En caso de error se devuelve -1. En caso de éxito, el número de bytes recibidos.

Para enviar datos se tiene la función `write()`:

```
int write(int sockfd, char *mesg, int len);
```

Donde:

```
int sockfd  /*descriptor del socket local*/
char mesg   /*dirección del mensaje a enviar*/
int len     /*longitud del mensaje*/
```

En caso de error se devuelve -1. Si tiene éxito. el número de bytes realmente escritos.

Asignación de Puertos

Para llevar a cabo una conexión entre un cliente y un servidor es necesario, además de proporcionar la dirección de red (IP) también el número de puerto al que el cliente se conectará con servidor. Es por eso que se hace necesario conocer la distribución de los puertos, la cual está dada de la siguiente forma:

Uso Previsto	*Puerto*
Puertos Reservados	1 -1023
Puertos Asignados por el Sistema	1024 - 5000
Puertos Libres para los Usuarios	5001 - 65535

Desarrollo

Un Cliente Simple Orientado a la Conexión

Digitar el siguiente programa el cual es un cliente que obtiene una cadena del programa servidor para ser impresa en pantalla.
A este programa cliente se le debe proporcionar la dirección IP del host servidor en notación de puntos como parámetro. Esta dirección puede ser la "127.0.0.1" (en caso de que no este trabajando en una red real) la cual corresponde a la interfaz de "loopback " la cual se considera una interfaz virtual. Dicha interfaz se utiliza para comunicar dos procesos dentro de la misma máquina Linux/Unix (es decir tanto el cliente como el servidor se ejecutan en el mismo ordenador). De este modo, aunque la máquina no este conectada a una red con protocolos TCP/IP, puede considerarse que esta conectada a la red 127.0.0.0 a través del citado interfaz.

Código del programa cliente (cliente.c)

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

#define PORT            3490
#define MAXDATASIZE     100
#define error(mesg) { perror(mesg); exit(1); }
```

```
int main(int argc, char *argv[])
{
   int sockfd, numbytes;
   char buf[MAXDATASIZE];
   struct sockaddr_in serv_addr;

   if (argc != 2)
   {
      fprintf(stderr,"Uso: cliente <Dirección IP>\n");
      exit(1);
   }

   if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) error("socket");

   serv_addr.sin_family = AF_INET;
   serv_addr.sin_port = htons(PORT);
   serv_addr.sin_addr.s_addr = inet_addr(argv[1]);
   bzero(&(serv_addr.sin_zero), 8);

   if (connect(sockfd, (struct sockaddr *)&serv_addr,
               sizeof(struct sockaddr)) == -1)  error("connect");

   if ((numbytes = read(sockfd, buf, MAXDATASIZE)) == -1) error("read");

   buf[numbytes] = '\0';
   printf("Cadena Recibida: %s\n",buf);

   close(sockfd);
   return 0;
}
```

Un Servidor Simple Orientado a Conexión

Digitar el siguiente código para un servidor orientado a conexión que proporciona una cadena a cualquier ordenador que ejecute el programa cliente respectivo.

Uno de los puntos importantes a notar es que este programa obtiene la dirección IP del anfitrión (host) utilizando la constante INADDR_ANY, no siendo proporciona directamente del código. Con esto se le dice al sistema operativo que acepte conexiones desde cualquier dirección IP. Gracias a ello el programa puede ser ejecutado en cualquier host sin tener que editar el código para modificar la dirección IP del ordenador con el que se establece la conexión.

Se debe notar que en la llamada a la entrada del sistema `socket()`, se especifica al parámetro dominio como AF_INET (IPV4), y al tipo de protocolo como SOCK_STREAM (TCP), esto con el fin de crear una aplicación orientada a conexión TCP/IP.

Código del programa servidor (servidor.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/wait.h>

#define MYPORT    3490
#define BACKLOG   10
#define error(mesg){ perror(mesg); exit(1); }
```

```
main()
{
  int sockfd, new_fd;
  struct sockaddr_in serv_addr;
  struct sockaddr_in cli_addr;
  int sin_size, len_cadena;
  char cadena[] = "Hola, Este es un ejemplo de socket stream!";

  if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) error("socket");

  serv_addr.sin_family = AF_INET;
  serv_addr.sin_port = htons(MYPORT);
  serv_addr.sin_addr.s_addr = INADDR_ANY;
  bzero(&(serv_addr.sin_zero), 8);

  if (bind(sockfd, (struct sockaddr *)&serv_addr,
           sizeof(struct sockaddr)) == -1) error("bind");

  if (listen(sockfd, BACKLOG) == -1) error("listen");

  while(1)
  {
    sin_size = sizeof(struct sockaddr_in);
    if ((new_fd = accept(sockfd, (struct sockaddr *)&cli_addr, &sin_size))
    == -1) error("accept");

    len_cadena = sizeof(cadena);

    if (write(new_fd, cadena, len_cadena) == -1) error("send");
    close(new_fd);
  }

  return 0;
}
```

Asignaciones

Para todas las aplicaciones cliente-servidor que se deban desarrollar, se debe asegurar la utilización de números de puertos apropiados.

1. El comando uname (mostrar información del sistema) con el parámetro -a muestra la siguiente información :

   ```
   > uname -a
   > Linux localhost.localdomain 2.0.36 #1 Tue Oct 13 22:17:11 EDT 1998 i586
   ```

   Crear un programa que muestre la misma información anterior y además que determine la representación de datos del host (anfitrión). Para determinar la representación de datos se recomienda utilizar un tipo de datos `union` y para obtener la información del host utilizar la estructura `utsname`.

   El formato de la información que dicha programa debe mostrar, es el siguiente:

   ```
   > sisinfo
   Representación de datos: Little-Endian
   Sistema Operativo: Linux
   Nodo: localhost.localdomain
   O.S. Distribución: 2.0.36
   O.S. versión: #1 Tue Oct 13 22:17:11 EDT 1998
   Tipo de hardware: i586
   ```

Para determinar la representación interna de datos del host se recomienda utilizar un tipo de datos `union` y para obtener la información del general del host utilizar la estructura `utsname` definida en el archivo `utsname.h`

2) Modificar el programa servidor de la practica para que muestre en pantalla la dirección IP de cliente que este atendiendo actualmente, se recomienda utilizar la función `inet_ntoa()`.

3) Crear un programa cliente y un servidor que utilicen el protocolo TCP que permitan realizar el eco (envío de un mensaje al servidor por parte del cliente y la espera por parte de este del vuelta del mensaje enviado) Ambos mensajes se deben imprimir en la pantalla del cliente, esto se repite por cada mensaje enviado hasta que se presiona **<ENTER>** o Fin de archivo **<Control+D>**. Al cliente se le debe proporcionar como parámetro la dirección IP del host servidor.

4) Desarrollar una aplicación cliente-servidor en la que el cliente transmita un comando a la vez al programa servidor para que este lo ejecute en host servidor.
La respuesta de la ejecución de dicho comando debe ser devuelta al programa cliente para que sea mostrada en la pantalla del host cliente. Se debe utilizar protocolo TCP para la implementación de la aplicación. Se recomienda utilizar la función `dup2()` para redireccionar tanto la entrada, la salida y el error estándar al socket de comunicación en el programa servidor para que la salida del comando ejecutado en el host servidor pueda ser mostrada apropiadamente en el host cliente.
El cliente debe tener la capacidad de aceptar un comando o el carácter `s` (Salir), el cual hace que el proceso cliente termine. Al programa cliente se le proporcionará en la llamada la dirección IP del host servidor.

   La ejecución del programa cliente debe tener el aspecto siguiente:

```
> remoto 169.168.0.5
169.168.0.5> date
169.168.0.5> Thu May 22 19:28:11 2000
169.168.0.5> s
> remoto terminado.
```

5) Desarrollar una aplicación cliente-servidor que tenga la capacidad de transferencia de archivos. Se debe utilizar el protocolo TCP para crear la implementación. El programa cliente transfiere un archivo al host servidor y este devuelve el archivo al host cliente en formato comprimido (utilizar el programa `gzip` o cualquier otra utilidad de compresión). La aplicación debe tener la capacidad de transferencia de archivos en cualquier formato (archivos de texto, binarios, graficos, etc.) y sin restricción de tamaño.

6) Desarrollar una aplicación cliente-servidor que utilice protocolo TCP, cuya función sea emular una calculadora distribuida. Al programa cliente se le debe proporcionarar en la llamada la dirección IP servidor y después se le proporcionarán los datosasí como la operación a realizar. Las operaciones disponibles serán las cuatro operaciones aritméticas básicas (+, -, *, /), las cuales serán procesadas por el programa servidor, el trabajo del cliente será suministrar los operandos y el operador al programa servidor. La calculadora debe tener la capacidad de procesar una operación aritmética a la vez hasta que se digite `s` (salir), lo cual hace que termina el proceso cliente. La finalización de dicho proceso debe ser informada al usuario por medio de un mensaje.

   La ejecución del programa cliente debe tener el aspecto siguiente:

```
> calculadora 169.168.0.5
169.168.0.5> 5*50
169.168.0.5> 250
169.168.0.5> 10-50
169.168.0.5> 40
169.168.0.5> s
> cliente terminado.
```

Referencias Bibliográficas

[1] Brian W. Kernighan.
"El Entorno de Programación UNIX."
Editorial Prentice Hall, 1992.
Primera Edición.

[2] W. Richard Stevens.
"Unix Network Programming, Networking API's Sockets and XTI".
Editorial Prentice Hall, 1998.
Segunda Edición.

[3] Páginas de Manual (Páginas `man`) de todas las funciones mencionadas en la guía.

[4] An Advanced 4.3BSD Interprocess Comunicación Tutorial
http://www.sparc.spb.su/jjb/Docs/internet/ipc-tutorial/index.html

[5] "An Advanced Socket Comunicación Tutorial"
http://viks.mvrop.org/networking/sock_advanced_tut.html

Guía de Laboratorio No. 2

Sockets No Orientados a Conexión

Introducción

Existen algunas diferencias entre aplicaciones escritas usando TCP y aquellas que usan UDP. Esto es debido a las diferencias entre los dos protocolos: UDP es un protocolo no orientado a conexión, no confiable, un protocolo de datagramas. Es diferente a TCP, el cual es orientado a la conexión, confiable, protocolo de corriente bytes. Sin embargo, hay circunstancias cuando tiene sentido usar UDP en lugar de TCP. Tales situaciones son tratadas con mayor detalle en guías posteriores. Algunas aplicaciones populares son construidas utilizando UDP son: DNS (Sistema de nombres de Dominio), NFS (Sistema de Archivos en Red), etc

Funciones Típicas en una Aplicación Cliente/Servidor UDP

En una aplicación cliente-servidor UDP, el cliente no establece una conexión con el servidor. En lugar de eso, el cliente solo envía un datagrama al servidor usando la función `sendto`, que requiere la dirección destino (dirección del servidor) como parámetro. Similarmente, el servidor no acepta una conexión desde el cliente. En lugar de eso, el servidor solo llama a la función `revcfrom` la cual espera hasta que los datos lleguen desde algún cliente. La función `recvfrom` devuelve la dirección del cliente, junto con el datagrama, y de esta manera el servidor pueda enviar respuesta al cliente actual. Debido a esto, las aplicaciones orientadas a la no conexión son mucho más senciallas que sus contrapartes orientadas a la conexión.

Envío y Recepción de Datos No Orientados a Conexión

La primitiva `sendto` permite realizar el envío de datos en la forma que se muestra a continuación:

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

int sendto(sockfd, msg, lg, opcion, p_dest, lg_dest)
```

Donde:

```
int sockfd;                 /* Descriptor del socket de emisión */
const void *msg;            /* Dirección del buffer de envío */
int lg;                     /* Longitud del buffer */
int opcion;                 /* Generalmente 0 */
struct sockaddr *pdest;     /* Longitud de la dirección del socket destino */
```

La llamada no es bloqueante, devuelve el control cuando se copian los datos del espacio de usuario al del núcleo. El valor de retorno es, en caso de éxito, el número de caracteres enviados y -1 en caso de fallo.

Para recibir datos existe la primitiva recíproca, `recvfrom()`:

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

int recvfrom (sockfd, msg, lg, opcion, p_exp, lg_exp)
```

Donde:

```
int sockfd;                 /* Descriptor del socket de recepción*/
void *msg;                  /* Dirección del buffer para recepción*/
int lg;                     /* Tamaño del buffer*/
int opción;                 /* Normalmente 0. Ver la página man */
struct sockaddr *p_exp;     /* Para recuperar la dirección de emisión*/
int *lg_exp;                /* Longitud de p_exp*/
```

Se extraerá del socket un mensaje completo, que corresponde a la información enviada en una sola operación de `sendto()`. La llamada es bloqueante hasta recibir datos. Devuelve, en caso de éxito, el número de bytes recibidos En caso de error devuelve -1.

La llamada también devuelve la dirección del que nos envía los datos, de esta forma podemos porporcionar una respuesta. Esta dirección es dejada en la estructura apuntada por `p_exp`. Si no se está interesado en saber la dirección del cliente, estos valores pueden tener un valor nulo (NULL).

Desarrollo

El siguiente ejemplo es la solución de la asignación #5 (calculadora distribuida) de la guía #1, utilizando el protocolo UDP. Al programa cliente se le proporcionará en la llamada la dirección IP del host servidor y después se le proporcionarán los datos y la operación a realizar. Las operaciones disponibles serán las cuatro operaciones aritmeticas básicas (+, - , *. /). Cuando ya no se deseen realizar cálculos, se puede digitar `s` (salir) para terminar el programa cliente.

Código del programa cliente (cliente.c):

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

#define   PORT     5004
#define   MAX      100

struct num
{
   float     a;
   float     b;
   char      oper;
};

struct resp
{
   int flag;
   float res;
};

int main( int argc, char *argv[])
{
   struct sockaddr_in addr;
   struct hostent *hp;
   int desc, lg;
   char cad[MAX], car, c, d, oper;
   struct num valor;
   float res;
```

```
    int flag = 0;
    struct resp result;

    system("clear");

    if(argc != 2)
    {
                printf("\n");
                printf("Uso: cliente nombre_host\n");
                printf("Ejemplo: cliente localhost\n");
                exit(2);
    }

    printf("  ********************CALCULADORA DISTRIBUIDA********************");
    printf("\nEste programa realiza con");
    printf(" dos números una de las operaciones siguientes:\n");
    printf("\n + Suma\n - Resta\n * Multiplicacion\n / Division\n");
    printf(" s Salir\n c Limpiar\n");
    printf("\nPrimer operando, operador, segundo operando:");
    printf("\nEjemplo: 10 * 30\n\n");

    desc = socket(AF_INET, SOCK_DGRAM, 0);

    if (desc == -1)
    {
       perror("Creación imposible del socket");
       exit(3);
    }

    lg = sizeof(addr);

    addr.sin_family = AF_INET;
    addr.sin_port = htons(PORT);
    addr.sin_addr.s_addr = inet_addr(argv[1]);
    bzero(&(serv_addr.sin_zero), 8);
    bind(desc, (struct sockaddr *)&addr, sizeof(addr));

    while(1)
    {
      printf("%s> ",argv[1]);
      gets(cad);
      if(!strcmp(cad,"s")) exit(1);
      if(!strcmp(cad,"c")) system("clear");
      else
      {
        sscanf(cad,"%f %c %f", &valor.a, &valor.oper, &valor.b);
        sendto(desc, &valor, sizeof(struct num), 0,
                    (struct sockaddr *)&addr, sizeof(addr));
        recvfrom(desc, &result, sizeof(struct resp), 0,
                 (struct sockaddr *)&addr, &lg);

        if(result.flag == -1)
        printf("Operacion no valida\n");
        else printf("%s> %f \n", argv[1], result.res);
        bzero(&(valor), sizeof(valor));
      }
    }
}
```

Código del programa servidor (servidor.c):

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

#define PORT                    5004
#define SUMA                    '+'
#define RESTA                   '-'
#define MULTIPLICACION  '*'
#define DIVISION        '/'

struct num
{
   float a;
   float b;
   char oper;
};

struct resp
{
   int flag;
   float res;
};

int main( int argc, char *argv[])
{
   struct sockaddr_in addr_1, addr_2;
   int lg, desc;
   struct num numeros;
   char op;
   float res;
   struct resp result;

   if((desc = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
   {
      perror("Imposible crear socket");
      exit(2);
   }

   addr_1.sin_family = AF_INET;
   addr_1.sin_port = htons(PORT);
   addr_1.sin_addr.s_addr = INADDR_ANY;

   if(bind(desc,(struct sockaddr *)&addr_1, sizeof(addr_1)))
   {
      perror("Imposible nombrar socket");
      exit(3);
   }

   while(1)
   {
      lg = sizeof(addr_2);
      bzero((char *)&numeros, sizeof(struct num));

      recvfrom(desc, &numeros, sizeof(struct num), 0,
               (struct sockaddr *)&addr_2, &lg);
```

```
    switch(numeros.oper)
    {
       case SUMA:
       result.res = numeros.a + numeros.b;
       result.flag = 0;
       break;
       case RESTA:
       result.res = numeros.a - numeros.b;
       result.flag = 0;
       break;

       result.res = numeros.a * numeros.b;
       result.flag = 0;
       break;
       case DIVISION:
       result.res = numeros.a / numeros.b;
       result.flag = 0;
       break;
       default:
       result.res = 0;
       result.flag = -1;
    }
    sendto(desc, &result, sizeof(struct resp), 0,
           (struct sockaddr *)&addr_2, lg);
  }
}
```

Asignaciones

1) Crear una aplicación cliente-servidor que utilice el protocolo UDP capaz de realizar un eco, es decir, el envío de un mensaje al servidor por parte del cliente y la espera por parte de este último del retorno del mensaje enviado. Ambos mensajes se deben imprimir en la pantalla del cliente. Esto debe repetirse por cada mensaje enviado hasta que se presiona **<INTRO>** (sin escribir ningún otro carácter en el mismo renglón) o fin de archivo **<Control+D>**. Al cliente se le debe proporcionar como parámetro la dirección IP del host servidor.

2) Desarrollar una aplicación cliente-servidor *"daytime"* (hora del día) utilizando protocolo UDP. El programa cliente debe imprimir en la terminal del host cliente la hora y fecha del host servidor obtenida por medio del programa servidor. Se recomienda utilizar la estructura `timeval`.

3) Desarrollar una aplicación cliente-servidor en la que el cliente transmita un comando a la vez al programa servidor para que éste se ejecute en dicho servidor.
   La respuesta de la ejecución de dicho comando debe ser devuelta al programa cliente para que sea mostrada en la pantalla del host cliente. Se debe utilizar protocolo UDP para la implementación de la aplicación. Se recomienda utilizar la función `dup2()` para redireccionar tanto la entrada, la salida y el error estándar al socket de comunicación en el programa servidor para que la salida del comando ejecutado en el servidor pueda ser mostrada apropiadamente en el cliente. Asimismo, puede valerse de la función `system` para la ejecución de comandos del sistema operativo.
   El cliente debe tener la capacidad de aceptar un comando o el carácter `s` (salir), el cual hace que el proceso cliente termine. Al programa cliente se le proporcionará en la llamada la dirección IP del host servidor.
   La ejecución del programa cliente debe tener el aspecto siguiente:

```
> remoto 169.168.0.5
169.168.0.5> date
169.168.0.5> Thu May 22 19:28:11 2000
169.168.0.5> s
> remoto terminado.
```

Referencias Bibliográficas

[1] W. Richard Stevens.
"Unix Network Programming, Networking API's Sockets and XTI".
Editorial Prentice Hall, 1998.
Segunda Edicion.

[2] Páginas del Manual (Páginas man) de todas funciones mencionadas en la guía. .

[3] "An Advanced 4.3BSD Interprocess Communication Tutorial"
http://www.sparc.spb.su/jjb/Docs/internet/ipc-tutorial/index.html

[4] "An Advanced Socket Communication Tutorial"
http://viks.nwrop.org/networking/sock_advanced_tut.html

Guía de Laboratorio No. 3

Conversión de Nombres y Direcciones

Introducción

Todos los programas tratados han usado, hasta este momento, direcciones numéricas en los hosts y números de puerto para identificar servidores. Sin embargo, es mejor utilizar nombres en lugar de números por las siguientes razones:

- Los nombres son más fáciles de recordar.
- Los números pueden cambiar mientras que los nombres pueden permanecer sin cambio.

Esta guía muestra como utilizar funciones que convierten nombres en valores numéricos y viceversa (Por ejemplo: nombres de dominio a direcciones IP , nombres de servicios a números de puertos, etc).

Archivos de Configuración

Todos estos ficheros se encuentran en el directorio `/etc` y su estructura es similar: cada entrada es una línea en donde se relaciona un nombre con un número. Además puede que haya más de un nombre para un número, es decir, un *alias*. Los ficheros son los siguientes:

- `/etc/hosts` contiene la tabla de correspondencia entre la dirección Internet y un nombre simbólico. A cada máquina le correponde una línea con la siguiente información:

  Dirección Internet.
  Nombre Oficial.
  Lista de alias.
  Un comentario.

- `/etc/networks` forma la base de datos de las redes conocidas. A cada red le corresponde una línea con la siguiente información:

  Nombre Oficial.
  Direccion Internet.
  Un comentario.

- `/etc/protocols` muestra la lista de protocolos conocidos.

  Nombre Oficial.
  Número del protocolo.
  Lista de alias
  Un comentario

- `/etc/services` contiene los servicios Internet. Cada servicio esta formado por:

  Nombre.
  Número de puerto.
  Protocolo.

Estructuras Relacionadas

Las llamadas que se veran en el siguiente apartado, hacen uso de las estructuras que a continuación se detallan.

Para utilizarlas es necesario incluir el fichero de cabecera <netdb.h>:

```
#include <netdb.h>
```

La estructura hostent consulta el archivo /etc/hosts:

```
struct hostent
{
        char    *h_name;                    /*nombre oficial de la máquina*/
        char    **h_aliases;                /*lista de alias*/
        int     h_addrtype;                 /*tipo de dirección siempre AF_INET*/
        int     h_length;                   /*longitud de la dirección*/
        int     **h_addr_list;              /*lista de direcciones*/

        #define h_addr h_addr_list[0];    /*primera dirección de la lista*/
};
```

Los arrays de punteros siempre terminan con un puntero a NULL. La lista h_addr_list se compone de punteros a direcciones IP que pueden ser introducidas en estructuras tipo in_addr. En el caso de un ordenador con varios interfaces a distintas redes, puede usarse la lista de punteros a direcciones IP para conseguir la dirección de cada interfaz.

La estructura netent consulta el archivo /etc/networks:

```
struct netent
{
        char            *n_name;        /*nombre oficial de la red*/
        char            **n_aliases:    /*lista de alias*/
        int             addrtype;       /*tipo de dirección de la red*/
        unsigned long   n_net;          /*dirección de la red*/
};
```

La estructura protoent consulta el archivo /etc/protocols:

```
struct protoent
{
        char    *p_name;        /*nombre oficial*/
        char    **p_aliases;    /*lista de alias*/
        int     n_net;          /*número de protocolo*/
};
```

La estructura servent consulta el archivo /etc/services:

ruct servent

```
        char    *s_name;          /*nombre oficial del servicio*/
        char    **s_aliases;      /*lista de los alias*/
        int     s_port;           /*número del puerto*/
        char    *p_proto;         /*protocolo utilizado*/
```

Funciones Base

devuelven un puntero a una estructura de las vistas anteriormente, que contiene información recopilada de directorio /etc.

hivo /etc/hosts, o bien al servidor de nombres (DNS) cal que tiene acceso la red a través del odemos obtener el nombre y la dirección IP de cualquier ordenador. Existen dos funciones que

```
struct hostent *gethostbyname(nombre)
        char *nombre;

struct hostent *gethostbyaddr(dirección, longitud, tipo)
        char    *dirección;
        int     longitud;
        int     tipo;
```

También hay una función para conocer el nombre y dirección de la máquina local.

```
int gethostname (nombre, lg)
        char    *nombre;
        int     lg;
```

Coloca el nombre del sistema local en nombre. El espacio a reservar para la dirección del nombre se indica por medio de `lg`. En caso de error se devuelve -1.

Para el acceso a `/etc/network`, tenemos las funciones:

```
struct netent     *getnetbyname(nombre)
            char  *nombre;

struct netent     *getnetbyaddr(dirección, tipo)
            char  dirección;
            char  tipo;
```

De manera similar para `/etc/protocols` están las funciones:

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

struct protoent  *getprotobynumber(número)
            int número;
```

Para el acceso a `/etc/services`, están las funciones:

```
struct servent  *getservbyname(nombre, proto)
            char *nombre:
            char *proto;

struct servent    *getservbyport(puerto, proto)
            int puerto;
            char *proto;
```

La tabla 1 es un cuadro resúmen de toda la información presentada anteriormente:

Información	*Archivo*	*Estructura*	*Funciones*
Nodos	`/etc/hosts`	`hostent`	`gethostbyaddr, gethostbyname`
Redes	`/etc/networks`	`netent`	`getnetbyaddr, getnetbyname`
Protocolos	`/etc/protocolos`	`protoent`	`getprotobyname, getprotobynumber`
Servicios	`/etc/services`	`servent`	`getservbyname, getservbyport`

Tabla 1

Desarrollo

Se desea implementar un programa que obtenga la dirección IP de un ordenador a partir de su nombre de dominio. En este programa de ejemplo se muestra la utilidad de la estructura `hostent` y la función `gethostbyname`.

Digitar el programa, compilarlo y ejecurlo. Probar el desempeño del programa proporcionando diferentes direcciones IP (Internet) como argumento. Es necesario recordar al lector que si la entrada respectiva del host que se solicita está especificada en el archivo `/etc/hosts` o en el Servidor de Nombres de Dominio (DNS) al cual tiene acceso la red, la respuesta del programa será satifactoria. Se recomienda hacer las pruebas de este programa en un ordenador con acceso a Internet, aunque esto no es un requisito indispensable.

Código del programa getip.c

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>

int main(int argc, char *argv[])
{
  struct hostent *h;

  if (argc != 2) {
    fprintf(stderr,"uso: getip dirección\n");
    exit(1);
  }

  if ((h=gethostbyname(argv[1])) == NULL) {
    herror("gethostbyname");
    exit(1);
  }

  printf("Nombre del Host : %s\n", h->h_name);
  printf("Direccion IP : %s\n",inet_ntoa(*((struct in_addr *)h->h_addr)));

  return 0;
}
```

El programa anterior generaría, si se le introduce como parámetro la dirección de interfaz Loopback, la salida siguiente:

```
> get_info localhost
  Nombre del Host : localhost
  Dirección IP : 127.0.0.1
```

Asignaciones

1) Desarrollar un programa que pueda imprimir el nombre (también los alias) y dirección IP del ordenador local (el programa no debe soportar ningún parámetro).

2) Desarrollar un programa que imprima información de la estructura `hostent` de todos los nombres de hosts que sean suministrados desde la línea de comandos.

   La salida del programa (El nombre supuesto del programa es `gethostent`) debe tener el aspecto siguiente:

```
> hostent localhost roco
Nombre Oficial del host: localhost
Alias: localhost.localdomain
Tipo de dirección = 2
Tamaño de dirección (bytes) = 4
Dirección IP: 127.0.0.1
```

```
Nombre Oficial del host: rascabuches
Alias: rascabuches.pocaspulgas.com
Tipo de dirección = 2
Tamaño de dirección (bytes) = 4
Dirección IP: 168.120.10.5
```

3) Modificar el programa *daytime* cliente (UDP) para que tenga la capacidad de tomar como parámetros el nombre del host servidor y el servicio ("daytime") que se solicita.
El único miembro de la estructura `sockaddr_in` en el programa cliente que hace referencia al host servidor, al cual le puede asignar informacion directamente, es `sin_family` con la constante AF_INET. Los demás miembros de la estructura deben ser llenados por medio de la información que puede ser adquirida de alguna de las funciones indicadas en la Tabla 1, apoyándose en los parámetros introducidos desde la línea de comandos. En el host cliente y servidor modificar los archivos de configuración del directorio `/etc` que sean necesarios.
Además el programa *daytime* servidor debe mostrar la dirección IP y el nombre de dominio de cada host cliente que utilice sus servicios. Para tal objetivo utilizar la función `getpeername()`.

Referencias Bibliográficas

[1] W. Richard Stevens.
"Unix Network Programming, Networking API's Sockets and XTI".
Editorial Prentice Hall, 1998.
Segunda Edicion.

[2] Uday O. Pabrai
"UNIX: Interconexión de Redes"
Editorial RA-MA, 1997.
Segunda Edición.

[3] Páginas del Manual de todas funciones mencionadas en la guía.

[4] "An Advanced 4.3BSD Interprocess Communication Tutorial"
http://www.sparc.spb.su/jjb/Docs/internet/ipc-tutorial/index.html

[5] "An Advanced Socket Communication Tutorial"
http://viks.mvrop.org/networking/sock_advanced_tut.html

Guía de Laboratorio de No. 4

Señales en Aplicaciones Cliente-Servidor

Introducción

En esta guía se introducen dos funciones elementales de gran utilidad como son `fork()` y `signal()` que son utilizadas para crear programas servidores TCP concurrentes. Un servidor TCP concurrente es aquel que tiene la capacidad de atender más de un cliente a la vez, debido a la creación de procesos servidores hijos desde el proceso padre. Los tópicos concernientes a los tipos de servidores que existen serán tratados con mayor detalle en la guía # 9, la cual cubre también a los servidores concurrentes.
También se introduce el concepto de *manejo de señales*, el cual es muy importarte para los procesos servidores, ya que de esta manera, cuando se crea un proceso servidor hijo por parte del proceso servidor padre, éste debe dar el tratamiento adecuado cuando la conexión respectiva del proceso hijo haya terminado.

La primitiva fork()

Esta primitiva crea un nuevo proceso idéntico al proceso que ha realizado la llamada. El proceso que realiza la llamada recibe el nombre de *padre*, mientras que el nuevo proceso recibe el nombre de *hijo*. Ambos procesos continúan ejecutándose al volver de la llamada. El valor devuelto por `fork()` es cero (0) en el proceso hijo, y en el proceso padre el valor el identificador del proceso hijo (PID: *Process ID*). El prototipo de la función `fork()` es el siguiente:

```
#include <unistd.h>
pid_t fork(void);
```

Como una demostración del uso de `fork()`, a continuación se presenta la solución a la asignación # 4 de la guía 1. Se debe observar que el programa servidor hace uso de función `fork()` con el fin de crear un servidor concurrente.
Compilar el código cliente y servidor. Ejecutar el servidor y el cliente en hosts distintos, probar el funcionamiento de la aplicación.

Código del archivo remoto.h

```
# include <signal.h>
# include <unistd.h>
# include <stdio.h>
# include <string.h>
# include <sys/types.h>
# include <sys/socket.h>
# include <sys/wait.h>
# include <netinet/in.h>
# include <netdb.h>
# include <wait.h>
# define error(msg) { perror(msg) ; exit(1) ; }
# define PORTNUM 4093
```

Código del programa cliente (remoto.c)

```
#include "remoto.h"

int main(int argc, char* argv[])
{
  struct sockaddr_in bba ;
```

```
  struct hostent *hp ;
  int s, i=0 ;
  char comando[100], buffer[5000000];

  if (argc < 2)
  {
    fprintf(stderr, "Uso: %remoto [nombre_de_host]\n", argv[0]);
    exit(1) ;
  }

  buffer[i] = comando[0] = '\0' ;

  memset(&bba, 0 , sizeof(bba)) ;
  bba.sin_family = AF_INET ;
  hp = gethostbyname(argv[1]) ;

  if (hp == NULL) error ("no tal host");
  memcpy(&bba.sin_addr, hp->h_addr, hp->h_length) ;
  bba.sin_port = htons(PORTNUM) ;

  s = socket (AF_INET, SOCK_STREAM, 0) ;
  if (s == -1) error ("socket") ;
  if (connect (s, &bba, sizeof(bba)) != 0) error ("connect") ;

  memset(&comando, '\0', sizeof(comando));
  printf("%s> ", hp->h_name);
  gets(comando);
  write(s, comando, sizeof(comando));

  while (strcmp(comando,"s") != 0)
  {
    memset(&buffer, '\0', sizeof(buffer));
    sleep(1);
    i = read(s, buffer, sizeof(buffer));
    puts(buffer);
    printf("%s> ", hp->h_name);
    gets(comando);
    write(s, comando, sizeof(comando));
  }
  return(0);
}
```

Código del programa servidor (remotos.c)

```
#include "remoto.h"

int main(int argc,char* argv[])
{
  struct sockaddr_in saddr ;
  struct hostent *hp ;
  char buffer[20000], hostname[256];
  int s = -1,i=0, sfd=0, slen=0;

  memset(&saddr, 0, sizeof(saddr)) ;
  saddr.sin_family = AF_INET ;
  gethostname(hostname, sizeof(hostname)) ;
  hp = gethostbyname(hostname) ;
  memcpy(&saddr.sin_addr, hp->h_addr, hp->h_length) ;
```

```
    saddr.sin_port = htons(PORTNUM) ;

    s = socket (AF_INET, SOCK_STREAM, 0) ;

    if (s == -1) error("socket") ;
    if (bind (s, &saddr, sizeof(saddr)) != 0) error("bind") ;

    slen = sizeof(saddr) ;
    if (getsockname(s, &saddr, &slen) != 0) error("getsockname") ;

    if (listen(s,1) != 0) error("listen") ;

    while(1)
    {
      buffer[0] = '\0' ;
      sfd = accept(s, NULL, NULL) ;

      if (sfd == -1) error("accept") ;
      i = read (sfd, buffer, 1024) ;
      buffer[i-1] = '\0' ;

      dup2(sfd, 1) ;
      dup2(sfd, 2) ;

      if (fork() == 0)
      {
        close(s);
        while(strcmp(buffer,"s") != 0)
        {
          system(buffer);
          write(sfd, "\0",1);
          memset(&buffer,'\0', sizeof(buffer)) ;
          i = read (sfd, buffer, 1024) ;
        }
        close(sfd);
        exit(0);
      }
      close(sfd) ;
    }
    return(0);
}
```

Desde la línea de comandos del host servidork, digitar el comando ps (process status) desde la misma terminal en que se ejecuta en proceso servidor:

```
host-servidor> ps

 PID     TTY      STAT    TIME       COMMAND
 319     a0       S       0:00       gpm -t ms
 418     2        S       0:00      (mingetty)
 419     3        S       0:00      (mingetty)
 421     5        S       0:00      (mingetty)
 422     6        SW      0:00      (mingetty)
 479     p1       S       0:00      (bash)
 579     1        S       0:00      -bash
 985     1        S       0:00      ./remotos
 986     1        S       0:00      ./remotos
```

La salida del comando `ps` muestra los procesos que se están ejecutando. El campo STAT muestra el estado del proceso. PID muestra el identificador de proceso y COMMAND indica el proceso en cuestión.

Las banderas posibles para STAT así como el respectivo significado se muestran a continuación:

```
R    Ejecutándose
S    Durmiendo
D    Letargo Ininterruptible
T    Detenido
Z    Proceso Zombi
```

La salida anterior muestra que el proceso servidor padre con número PID de 985 tiene un STAT Durmiente (es decir, esperando respuesta o bloqueado). El proceso hijo tiene un número PID de 986 y un STAT Durmiente.

Cerrar el Cliente, ejecutar el comando `ps` otra vez en la misma terminal del servidor y observar su salida:

```
host-servidor> ps

 PID     TTY     STAT     TIME     COMMAND
 319     a0      S        0:00     gpm -t ms
 418     2       S        0:00     (mingetty)
 419     3       S        0:00     (mingetty)
 421     5       S        0:00     (mingetty)
 422     6       SW       0:00     (mingetty)
 479     p1      S        0:00     (bash)
 579     1       S        0:00     -bash
 985     1       S        0:00     ./remotos
 986     1       Z        0:00     (remotos <zombie>)
```

El único cambio que se puede observar es en el proceso hijo, el cual tiene un STAT zombi. Cuando un proceso hijo termina, el núcleo del sistema envía una señal SIGCHLD al proceso padre.
Si no se tiene la precaución de que el proceso padre consulte la finalización de sus procesos hijos, la tabla de procesos termina por llenarse. Debido a esto se hace necesario eliminar los procesos hijos cuando terminan. Para solucionar este problema es necesario saber cómo manipular las señales.

Manejo de Señales

Una señal es una notificación a un proceso de que un evento ha ocurrido. Las señales son llamadas a veces *interrupciones por software*. Las señales usualmente ocurren asincrónicamente, es decir, el proceso no sabe de antemano exactamente cuando una señal ocurrirá..

Las señales pueden ser enviadas de la manera siguiente:

- De un proceso a otro (o a el mismo).
- Del kernel a un proceso.

La señal SIGCHLD que se ha descrito anteriormente es enviada por el núcleo siempre que un proceso finaliza hacia el padre del proceso que termina.
Cada señal tiene una acción asociada, que es establecida llamando a la función `signal`. Cada vez que la señal `signal` es utilizada, se debe especificar la señal y el tratamiento a proporcionar a dicha señal, es decir, la acción asociada. Para tal efecto exiten tres opciones para la realización de la acción:

1. Se puede proveer una función que es llamada siempre que una señal ocurre. Esta función es conocida como un *manejador de señal* y dicha acción es llamada *captura de señal*. El prototipo de la función manejadora de señal, puede poseer un entero como argumento que es el número de señal y la función no debe regresar nada. Tal prototipo es el siguiente:

```
void manejar_señal(int señal);
```

2. Se puede indicar que la señal sea ignorada especificando la acción a realizar como SIG_IGN.
3. Se puede también especificar que la acción a realizar sea la asignada por defecto indicando que la acción a realizar como SIG_DFL.

Manejo de la Señal SIGCHLD

Un proceso Linux, una vez ha finalizado, pasa a ser un proceso *zombi*. El proceso padre puede utilizar el resultado (código de retorno o *exit status*) que el proceso hijo le suministra tras su finalización. Esta finalización se puede dar en cualquier momento. Mientras el proceso padre no consulte el código de retorno de sus procesos hijos, estos estarán dormidos (en estado zombi).

Para evitar que la tabla de procesos termina por llenarse, se debe capturar la señal SIGCHLD, que es enviada al proceso padre cada vez que los procesos hijos finalizan.
Como se ha mencionado anteriormente, para la captura de esta señal tenemos la función `signal()`, con la cual se debe especificar la acción a tomar para dicha señal y puede ser cualquiera de las mencionadas anteriormente según convenga como se muestra a continuación :

```
#include <sys/wait.h>
#include <signal.h>
...
main()
{
...
        signal(SIGCHLD, acción_a_realizar);
...
}
```

Como ejemplo de manejo de señales, se muestra a continuación la solución a la aplicación "transferencia archivos" que es la asignación # 5 de la guía #1.Esta versión que se presenta hace uso de `fork()` y `signal()` con el fin de mostrar cómo hacer uso de la función `signal()` utilizando la constante SIG_IGN como parámetro y de esta manera eliminar los procesos zombies en el host servidor. Compilar la aplicación y ejecutar los programas cliente y servidor respectivamente. Comprobar con el comando `ps` que los procesos zombis son eliminados en el host servidor cada vez que un cliente en ejecución cierra la conexión.

Código del programa cliente (transfer.c):

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/stat.h>

#define PORT            20000
#define MAXDATASIZE     4500

int main(int argc, char *argv[])
{
  int sockfd, numbytes, fd, size, n;
  char *buf;
  struct hostent *he;
  struct sockaddr_in serv_addr;
  char *name;
  struct stat file;
```

```
if (argc != 3)
{
   fprintf(stderr,"uso: transfer nombre_host nombre_archivo\n");
   exit(1);
}

if ((he=gethostbyname(argv[1])) == NULL)
{
   perror("gethostbyname");
   exit(1);
}

if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
{
   perror("socket");
   exit(1);
}

serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(PORT);
serv_addr.sin_addr = *((struct in_addr *)he->h_addr);
bzero(&(serv_addr.sin_zero), 8);

if(stat(argv[2], &file) == -1)
{
     printf("stat: Archivo %s no existe \n", argv[2]);
     exit(0);
}
if (connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(struct sockaddr))
     == -1)
{
     perror("connect");
     exit(1);
}

/*Envio del archivo*/

buf = (char *) calloc(file.st_size, sizeof(file.st_size));
writen(sockfd, &file.st_size, sizeof (file.st_size));
fd = open(argv[2], 0);

while ((n = readn(fd, buf, sizeof buf)) > 0) writen(sockfd, buf, n);

free(buf);
close(fd);

/*Recepcion del archivo*/

readn(sockfd, &size, sizeof(size));
buf = (char *) calloc(size, sizeof(char));
if ((numbytes=readn(sockfd, buf, size)) == -1)
{
   perror("read");
   exit(1);
}

buf[numbytes] = '\0';
fd = creat("comprimido.gz",1);
writen(fd, buf, numbytes);
close(sockfd);
```

```
  return 0;
}
```

Código del programa servidor (transfers.c):

```
#include <stdio.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <time.h>
#include <signal.h>
#include <sys/wait.h>

#define  MAXLINE 4096
#define MYPORT    20000
#define COLA 10

main(int argc, char *argv[])
{
  char      *buf;
  time_t    ticks;
  int sockfd, sock, n;
  struct sockaddr_in serv_addr;
  struct sockaddr_in cli_addr;
  int sin_size;
  struct stat file;
  char *name;
  int size;
  int numbytes, fd;

  if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
  {
    perror("socket");
    exit(1);
  }

  serv_addr.sin_family = AF_INET;
  serv_addr.sin_port = htons(MYPORT);
  serv_addr.sin_addr.s_addr = INADDR_ANY;
  bzero(&(serv_addr.sin_zero), 8);

  if (bind(sockfd, (struct sockaddr *)&serv_addr,sizeof(struct sockaddr)) == -1)
  {
    perror("bind");
    exit(1);
  }

  if (listen(sockfd, COLA) == -1)
  {
    perror("listen");
    exit(1);
  }
```

```
  while(1)
  {
    sin_size = sizeof(struct sockaddr_in);
    if ((sock = accept(sockfd, (struct sockaddr *)&cli_addr,&sin_size)) == -1)
    {
      perror("accept");
      continue;
    }

    printf("servidor: conexión obtenida desde %s\n",
    inet_ntoa(cli_addr.sin_addr));

    signal(SIGCHLD, SIG_IGN);

    if (!fork())
    {
      /*Recepcion de archivo*/
      readn(sock, &size, sizeof(size));
      buf = (char *) calloc(size, sizeof(char));

      if ((numbytes = readn(sock, buf, size)) == -1)
                {
      perror("read");
      exit(1);

      buf[numbytes] = '\0';
      fd = creat("temp",1);
      writen(fd, buf, numbytes);

      free(buf);
      close(fd);

      /*Envío de archivo*/

      stat("temp", &file);
      buf = (char *) calloc(file.st_size, sizeof(file.st_size));

      writen(sock, &file.st_size, sizeof (file.st_size));
      system("gzip temp");
      fd = open("temp.gz", 0);

      while ((n = readn(fd, buf, sizeof buf)) > 0) write(sock, buf, n);

      remove("temp.gz");
      free(buf);
      close(fd);
      close(sock);

      exit(0);
    }

    close(sock);
  }

  exit(0);
}
```

Código de la función writen(), archivo writen.c :

```
#include <stdio.h>
```

```
#include <unistd.h>
#include <errno.h>

int writen(int fd, const void *vptr, size_t n)
{
  size_t      nleft;
  int         nwritten;
  const char  *ptr;
  ptr = vptr;
  nleft = n;
  while (nleft > 0)
  {
    if ((nwritten = write(fd, ptr, nleft)) <= 0)
    {
      if (errno == EINTR)
      nwritten = 0;
      else
      return(-1);
    }

  nleft -= nwritten;
  ptr  += nwritten;

  }
  return(n);
}
```

Código de la función readn(), archivo readn.c :

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>

int readn(int fd, void *vptr, size_t n)
{
  size_t    nleft;
  int       nread;
  char      *ptr;

  ptr = vptr;
  nleft = n;
  while (nleft > 0)
  {
    if ((nread = read(fd, ptr, nleft)) < 0)
    {
      if (errno == EINTR)
      nread = 0;
      else
      return(-1);
    } else if (nread == 0)
    break;

    nleft -= nread;
    ptr  += nread;
  }
  return(n - nleft);
}
```

Asignaciones

1) Modificar el programa servidor remoto de la práctica para que pueda manejar las señales SIGCHLD de los clientes. Para tal efecto indicar que dicha señal sea ignorada. Verificar con el comando `ps` que los procesos hijos del servidor son correctamente eliminados.

2) Investigar el funcionamiento y las diferencias entre las funciones `wait()` y `waitpid()`, y como estas pueden ser utilizadas junto con la función `signal()` para el manejo de la señal SIGCHLD. Modificar el código del programa servidor `remotos` para que tenga la capacidad de manejar las señales SIGCHLD haciendo uso de las funciones `wait()` y `waitpid()`. Verificar con el comando `ps`, que los procesos hijos del servidor son correctamente eliminados.

3) Estudiar detalladamente el código de las funciones `readn()` y `writen()` de la guía. Explicar el funcionamiento de estas funciones y explicar porque razón no se utilizan directamente las funciones `read()` y `write()`. Probar el funcionamiento del programa "transferenciade archivos" con las funciones `read()` y `write()` en lugar de `readn()` y `writen()` respectivamente. Si observa alguna variente en el funcionamiento del programa, explicar con el mayor detalle posible.

Referencias Bibliográficas

[1] Brian W. Kernighan.
"El Entorno de Programación UNIX."
Editorial Prentice Hall, 1992.
Primera Edicion.

[2] W. Richard Stevens.
"Unix Network Programming, Networking API's Sockets and XTI".
Editorial Prentice Hall, 1998.
Segunda Edicion.

[3] Jim Frost
"UNIX Signals and Process Groups"
Agosto 17 de 1994.
http://world.std.com/~jimf/papers/signals/signals.html

[4] Páginas del Manual de todas las funciones que aparecen en la guía.

Guía de Laboratorio No. 5

Multiplexación de Entrada-Salida

Introducción

Cuando se desarrollan aplicaciones cliente-servidor es necesario que las mismas tengan la capacidad de decirle al núcleo del sistema que necesitan ser notificados si una o más condiciones de entrada-salida están listas (es decir, si la entrada estándar esta lista para leer, o si un descriptor tiene datos para ser enviados o recibidos). Esta capacidad es conocida como *multiplexacion de entrada-salida* y es suministrada por algunas funciones tales como `select()`, `poll()` y `alarm()`. Esta guía solo cubre la función `select()`.

Modelo de Multiplexado de E/S

Este modelo espera a que se tengan datos válidos en un canal de E/S, pero se desbloquea si recibe información valedera por cualquiera de sus puntos de comunicación (por cualquiera de sus sockets, en este caso).

La función `select` es la más usada cuando se usa el modelo de multiplexado de E/S, ya que permite decirle al kernel (núcleo) que espere por cualquiera de múltiples eventos, despertando al proceso solamente cuando uno de ellos ocurre o cuando haya transcurrido una cierta cantidad de tiempo. Esta función está descrita en los archivos de cabecera `<sys/select.h>` y `<sys/time.h>`. Su formato es:

```
int   select (int maxdesc, fd_set *desc_lectura, fd_set *desc_escritura,
              fd_set *desc_excepcion, const struct timeval *t_limite)
```

Valor Devuelto: Número de descriptores listos.0 si se sobrepasa el tiempo de espera y -1 en caso de error.

maxdesc: Especifica el número máximo de descriptores a ser evaluados. Su valor es el máximo descriptor a ser probado mas uno. Esto se debe a que la numeración de los descriptores inicia en cero (0). Usualmente, este valor no supera a 10.

desc_lectura, desc_escritura, desc_excepción: Especifican los descriptores que deseamos el kernel pruebe para verificar sus condiciones de lectura, escritura o excepción. Esta función solamente soporta 2 tipos de excepciones: La llegada de información fuera de banda al socket y la presencia de información de control de estado. Nótese que el tipo de variables de estos argumentos es `fd_set` (descriptor set) que se define como una cadena de enteros, en la cual cada bit en cada entero corresponde a un descriptor. Por ejemplo, si se usan enteros de 16 bits, el primer elemento de la cadena correspondería a los descriptores 0 al 15. El borrado o puesta de algún valor en este tipo de datos se hace por medio de las siguientes macros:

`void FD_ZERO (fd_set *desc_set):`	Pone todos los bits en `desc_set` a cero (0).
`void FD_SET (int desc, fd_set *desc_set):`	Activa el bit `desc` de la cadena `desc_set`.
`void FD_CLR (int desc, fd_set *desc_set):`	Desactiva el bit `desc` de la cadena `desc_set`.
`int FD_ISSET (int desc, fd_set *desc_set):`	Prueba si el bit `desc` de la cadena `desc_set` se encuentra activado.

Si no interesa el contenido de `desc_lectura`, `desc_escritura` ó `desc_excepción`, puede dárseles el valor de punteros nulos (NULL).

t_limite: Este argumento le dice al kernel cuanto tiempo esperar como máximo a que uno se los descriptores especificados se encuentre listo. Una estructura de tipo *timeval* define dicho tiempo en segundos y microsegundos.

```
struct timeval
{
long  tv_sec;     //segundos.
long  tv_usec;    //microsegundos.
};
```

Se pueden dar 3 casos posibles:
1- Esperar por siempre. En este caso el valor de este argumento es un puntero nulo (NULL).
2- Esperar una cantidad fija de tiempo. Esto dependerá de los valores dados a la estructura.
3- No esperar. En este caso se le da a ambos elementos de la estructura un valor de cero. Cuando se hace esto la función devuelve inmediatamente después de chequear los descriptores.

Desarrollo

El siguiente programa muestra cómo la funcion `select()` puede ser utilizada para determinar si un descriptor de archivo tiene datos a ser leídos y la forma como se puede establecer un tiempo de espera para una operación de entrada-salida.
El programa acepta como parámetros el número de segundos y microsegundos que se deben esperar a que la entrada estándar se le suminstren datos antes de imprimir el mensaje "tiempo fuera" indicando que dicho tiempo ha expirado.
Digitar el programa, compilarlo y probar su funcionamiento.

Código del programa select.c

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

main(int argc, char *argv[])
{
   struct timeval tv;
   fd_set readfds;
   char c;
   int val;

   if(argc != 3)
   {
      printf("Uso: %s #segundos #microsegundos\n", argv[0]);
      exit(1);
   }

   FD_ZERO(&readfds);

   while(1)
   {
      tv.tv_sec = atol(argv[1]);
      tv.tv_usec = atol(argv[2]);

      FD_SET(0, &readfds);

      val = select(1, &readfds, NULL, NULL, &tv);

      if (FD_ISSET(0, &readfds))
      {
        getchar(c); getchar(c);
```

```
        printf("¡Datos leídos!\n");
      }
      else
      printf("¡Tiempo fuera!.\n");
    }
}
```

Asignaciones

1) Modificar la aplicación `remoto` introducida en la guía # 1, asignación 4, para que el servidor tenga la capacidad de cerrar una conexión con un cliente específico cuando no reciba comandos de dicho cliente después de 20 segundos, mostrando en pantalla el siguiente mensaje:

   `El cliente` *`"Nombre_host "`* `con Dirección IP` *`"Dirección_IP"`* `ha sido desconectado.`

   Tambien modificar el programa cliente para que tenga la capacidad de imprimir el mensaje *"Servidor no responde"* y terminar la conexión con el servidor cuando este no responda pasados 15 segundos después de que un comando haya sido introducido.

2) Investigar el funcionamiento de la función `alarm()`. Repetir la asignación anterior con la variante de utilizar la función `alarm()` en lugar de `select()` para establecer el tiempo de espera de las operaciones de entrada/salida que la requieran.

Referencias Bibliográficas

[1] W. Richard Stevens.
"Unix Network Programming, Networking API's Sockets and XTI".
Editorial Prentice Hall, 1998.
Segunda Edicion.

[2] Páginas del Manual de todas funciones mencionadas en la guía. .

[3] An Advanced 4.3BSD Interprocess Communication Tutorial
http://www.sparc.spb.su/jib/Docs/internet/ipc-tutorial/index.html

[4] An Advanced Socket Communication Tutorial
http://yiks.nwrop.org/networking/sock_advanced_tut.html

Guía de Laboratorio No. 6

El superusuario Inetd

Introducción

Un proceso puede ser ejecutado en segundo plano e independientemente del control de todas las teminales. Linux tiene muchas aplicaciones que son ejecutadas como procesos en segundo plano e independientes de las terminales entre las cuales puedemos mencionar a TELNET, FTP, SENDMAIL, etc.
Si bien es cierto que es fácil ejecutar servidores en segundo plano colocando el signo de ampersand (&) al final de la línea de comandos, se debe recalcar que a veces lo que se busca es que nuestras aplicaciones por sí solas se ejecuten en segundo plano automáticamente cada vez que el sistema se inicializa, y a la vez es necesario que sean independientes de cualquier terminal para así poder utilizarla para otras tareas (hay que recordar que se supone los programas servidores serán ejecutados en un ordenador reservado como servidor, aunque esto no es necesario).

Superusuario Inetd

Se ha explicado cómo escribir un programa servidor que escuche las peticiones de los clientes y que dicho servidor sea ejecutado en segundo plano.
Otro camino para proveer servicio para un puerto Internet es permitir que el demonio `inetd` escuche las peticiones de los clientes. `inetd` es un demonio* que espera (usando `select`) por mensajes en un número de puertos específicos. Cuando este programa recibe un mesaje, acepta la conexión y entonces bifurca un proceso hijo para correr el programa servidor correspondiente. Se debe especificar el puerto y el programa en el archivo `/etc/inetd.conf`.

Servicio Inetd

Escribir un programa servidor para ser ejecutado por `inetd` es fácil. Cada vez que alguien requiere una conexión al puerto apropiado, un nuevo proceso servidor es comenzado. La conexión ya existe en este momento, de tal manera que el programa servidor puede comenzar a escribir y leer datos inmediatamente.

También se puede usar `inetd` para servidores que usan sockets orientados a no conexión. Para estos servidores, `inetd` no trata de aceptar una conexión, debido a que la realizacion de una conexión no es posible en este caso. El programa servidor puede manejar una respuesta y después se dedica a atender otra conexión o se puede escoger que se mantenga leyendo más peticiones hasta que no hayan más y después terminar. Se debe especificar cuál de estas dos técnicas el servidor debe usar, cuando `inetd` es configurado.

Configuración de inetd

El archivo `/etc/inetd.conf` le dice a `inetd` en cual puerto debe escuchar y cual programa servidor debe ejecutar. Normalmente cada entrada en el archivo es una línea. Las líneas que comienzan con "#" son comentarios.

A continuación se muestran dos entradas estándares de `/etc/inetd.conf` :

```
ftp    stream  ,tcp    nowait  root    /libexec/ftpd     ftpd
talk   dgram    udp    wait    root    /libexec/talkd    talkd
```

* Un demonio es un programa que se encuentra siempre en ejecución. Los nombres de procesos demonio generalmente terminan con la letra "d", como es el caso de `syslogd` e `inetd`.

Cada entrada tiene el siguiente formato:

SERVICIO ESTILO PROTOCOLO ESPERA NOMBRE_USUARIO PROGRAMA ARGUMENTOS

El campo SERVICIO dice qué servicio este provee el respectivo programa. Debe ser el nombre de un servicio definido en `/etc/services`. `inetd` usa SERVICIO para decidir en qué puerto escuchar esta entrada.

El campo ESTILO y PROTOCOLO especifican el estilo de comunicación y el protocolo a usar para el socket de escucha. Debe ser el nombre de un estilo de comunicación como por ejemplo *"stream"* o *"dgram"*. PROTOCOLO debe ser uno de los protocolos listados en `/etc/protocols`. Los nombres típicos de protocolos son *"tcp"* para comunicaciones orientadas a conexión y *"udp"* para comunicaciones no orientadas a conexión.

El campo ESPERA puede ser *"wait"* o *"nowait"*. Se debe usar *"wait"* si ESTILO especifica una comunicación no orientada a conexión y el servidor, una vez iniciado, maneja muchas peticiones. Se debe usar *"nowait"* si `inetd` debe iniciar un nuevo proceso por cada petición que le llegue. Si ESTILO hace referencia a una comunicación orientada a la conexión, entonces ESPERA debe ser *"nowait"*.

NOMBRE_USUARIO es el nombre de usuario bajo el cual el servidor se ejecuta. Debe recordarse que `inetd` puede correr como `root` (superusuario), por lo tanto puede establecer el ID de usuario de los hijos arbitrariamente. Sin embargo es mejor evitar usar `root` para NOMBRE_USUARIO si es posible y utilizar otro nombre de usuario.

PROGRAMA en conjunto con ARGUMENTOS especifica el programa servidor a ejecutar. PROGRAMA debe ser un nombre absoluto de archivo especificando el ejecutable a correr (suele ser necesario incluir la ruta de búsqueda). ARGUMENTOS consiste de cualquier número de palabras, separadas por espacios, que se convierten en la línea de comandos de PROGRAMA, el cual generalmente es solamente el mismo nombre del programa.

La función syslog

Debido a que un servidor ejecutado por `inetd` no tiene una terminal de control, se necesita una manera de mostrar mensajes cuando algo sucede, ya sea mensajes de información normal o mensajes de emergencia que necesitan ser atendidos por un administrador. La función `openlog()` se utiliza para indicar como se desean que se registren estos mensajes para ser tratados porel demonio `syslogd`[#].

```
#include <syslog.h>

void openlog(const char *ident, int options, int facility);
```

Donde:

`ident:` Generalmente apunta el nombre del proceso que invoca a openlog().

`options:` Puede ser uno o varias de las siguientes (combinándolas usando el operador OR) :

LOG_CONS:	Enviar mensajes a consola si no puede enviarlas al demonio `syslogd`.
LOG_NDELAY:	No retardar apertura, crear socket ahora.
LOG_PERROR:	Conectarse a error estándar (`stderr`) y también enviar mensajes al proceso `syslogd`.
LOG_PID:	Incluir el ID de proceso con cada mensaje.

Para nuestras aplicaciones, utilizar siempre la constante LOG_PID en `options` (ver página man de `openlog()`).

`facility:` Identifica el tipo de proceso que envía el mensaje. Utilizarlo siempre como tipo cero (0) el cual es el valor por defecto.

Por lo general, el nombre de un demonio siempre finaliza con la letra "d".

El proceso syslogd

Cuando el sistema se inicializa, el demonio `syslogd` es ejecutado, y lee al archivo `/etc/syslog.conf`, el cual especifica qué hacer con cada tipo de mensaje que el proceso `syslogd` recibe. Estos mensajes pueden ser añadidos a un archivo, escritos a un usuario específico, si esta registrado, o enviados a el proceso `syslogd` en otro host.

Funcionamiento de inetd

1. Durante la inicializacion, `inetd` lee el archivo `/etc/inetd.conf` y crea un socket del tipo apropiado (stream o dgram) para el servicio especificado en el archivo.

2. `bind()` es llamado para el socket, especificando el puerto para el servidor y la dirección IP comodín. Este número de puerto TCP o UDP es obtenido llamando a la función `getservbyname` con el nombre de servicio y protocolo de los campos de los archivos de configuración como argumentos.

3. En los sockets TCP, `listen()` es llamado para que las peticiones de conexiones que arriven sean aceptadas. Este paso no es realizado para sockets UDP.

4. Después de que todos los sockets son creados, `select()` es llamado para esperar por cualquiera de los sockets que puedan ser leidos.

5. Cuando `select()` indica que un socket está disponible para ser leído, si el socket es TCP, `accept()` es llamado para aceptar una nueva conexión.

6. `inetd` crea un proceso hijo por medio de `fork()` para manipular el servicio requerido similarmente como a un servidor estándar concurrente.

Desarrollo

Digitar el código para el programa servidor `remotos`, el cual incluye las modificaciones necesarias para que el servidor sea ejecutado por `inetd`. El cliente es el mismo de la guía # 3, así como el archivo `remoto.h`. Recompilar el programa servidor.

Modificar los archivos `/etc/inetd.conf` y `/etc/services` de la manera siguiente:

Incluir la siguiente línea en el archivo `/etc/services`

```
remoto          9900/tcp
```

Incluir la siguiente línea en el archivo `/etc/inetd.conf` (solicitar la ayuda del administrador de la red en caso de ser necesario para la modificación de estos archivos)

```
remoto  stream  tcp  nowait  eienet  /home/eienet/remotos  remotos
```

Reinicializar `inetd` en el host servidor para que los cambios surtan efecto.

Código del archivo remoto.h

```
# include <signal.h>
# include <unistd.h>
# include <stdio.h>
# include <string.h>
# include <sys/types.h>
# include <sys/socket.h>
# include <sys/wait.h>
```

```
# include <netinet/in.h>
# include <netdb.h>
# include <wait.h>
# define error(msg) { perror(msg) ; exit(1) ; }
# define PORTNUM        9900
# define MAXSOCKADDR    128
```

Código del programa servidor (remotos.c)

```
#include "remoto.h"
#include <syslog.h>

int main(int argc, char* argv[])
{
   char buffer[20000];
   int i, sfd;
   socklen_t len;
   struct sockaddr *cliaddr;

   /* Añadir nombre e ID del proceso actual por cada mensaje syslogd
      que sea registrado*/

   openlog(argv[0], LOG_PID, 0);

   cliaddr = malloc(MAXSOCKADDR);

   len = MAXSOCKADDR;
   getpeername(0, cliaddr, &len);

   i = read (sfd, buffer, 1024) ;

   buffer[i-1] = '\0' ;

   dup2(sfd, 1) ;
   dup2(sfd, 2) ;

   while(strcmp(buffer,"s") != 0)
   {
      system(buffer);
      write(sfd, "\0",1);

      memset(&buffer,'\0', sizeof(buffer)) ;
      i = read (sfd, buffer, 1024) ;
   }
   close(0);
   exit(0);
}
```

Código del programa cliente (remoto.c)

```
#include "remoto.h"

int main(int argc, char* argv[])
{
   struct sockaddr_in serv ;
   struct hostent *hp ;
   int s, i=0 ;
   char command[100],buffer[5000000];
```

```
  if (argc < 2)
  {
     fprintf(stderr, "Uso: %remoto [nombre_host]\n", argv[0]);
     exit(1) ;
  }
  buffer[i] = command[0] = '\0' ;
  memset(&serv, 0 , sizeof(serv)) ;
  serv.sin_family = AF_INET ;
  hp = gethostbyname(argv[1]) ;

  if (hp == NULL) error ("No tal host");
  memcpy(&serv.sin_addr, hp->h_addr, hp->h_length) ;
  serv.sin_port = htons(PORTNUM) ;

  s = socket (AF_INET, SOCK_STREAM, 0) ;
  if (s == -1) error ("socket") ;
  if (connect (s, &serv, sizeof(serv)) != 0) error ("connect") ;

  system("clear");
  printf("**Digitar "s" para salir**\n\n");
  memset(&command, '\0', sizeof(command));
  printf("%s> ", hp->h_name);
  gets(command);
  write(s, command, sizeof(command));

  while (strcmp(command,"s") != 0)
  {
     memset(&buffer, '\0', sizeof(buffer));
     sleep(1);
     i = read(s, buffer, sizeof(buffer));
     puts(buffer);
     printf("%s> ", hp->h_name);
     gets(command);
     write(s, command, sizeof(command));
  }
  printf("Conexion finalizada\n");
  return(0);
}
```

Asignaciones

1. Modificar los servidores de las aplicaciones *"calculadora"* (utilizando protocolo TCP) y *"tranferencia de archivos""* (utilizando protocolo TCP) introducidas en guías anteriores para que utilicen el demonio `inetd` según se ha explicado en la presente guía. Probar el funcionamiento de los servidores con sus respectivos programas clientes. Asegurarse de utilizar números de puertos apropiados.

Referencias Bibliográficas

[1] W. Richard Stevens.
"Unix Network Programming, Networking API's Sockets and XTI".
Editorial Prentice Hall, 1998.
Segunda Edicion.

[2] Páginas del Manual de todas funciones mencionadas en la guía. .

[3] "An Advanced 4.3BSD Interprocess Communication Tutorial"
http://www.sparc.spb.su/jjb/Docs/internet/ipc-tutorial/index.html

[4] "An Advanced Socket Communication Tutorial"
http://viks.mvrop.org/networking/sock_advanced_tut.html

Guía de Laboratorio No. 7

Opciones de Sockets

Introducción

Hay varias maneras para especificar y obtener información de las opciones que afectan a los sockets.

- Las funciones `getsockopt` y `setsockopt`.
- La función `fcntl`.
- La función `ioctl`.

Esta guía cubre las funciones `setsockopt` y `getsockopt`, seguido de un ejemplo que muestra como modificar y obtener información de las opciones actuales del socket en uso. Las funciones `fcntl` e `ioctl` no serán cubiertas en esta guía. En caso de interés por parte del lector, ver las paginas man respectivas.

Funciones getsockopt y setsockopt

Estas dos funciones solo son aplicables a sockets:

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

int getsockopt(int sockfd, int level, int optname, void *optval,
               socklen_t *optlen);

int setsockopt(int sockfd, int level, int optname, const void *optval,
               socklen_t optlen);
```

Donde:

sockfd: Debe ser referencia a un descriptor socket abierto.

level: Especifica el nivel de protocolo en el sistema para interpretar la opción: Para manipular opciones a nivel de socket, `level` debé especificarse como SOL_SOCKET; para manipular opciones a nivel de protocolo TCP, `level` se especifica como IPPROTO_TCP; y para manipular opciones a nivel de protocolo IP, `level` se debe especificar como IPPROTO_IP.

optval: Es un puntero a una variable desde el cual el nuevo valor de la opción es establecida por medio de `setsockopt`, o dentro del cual el actual valor de la opción es almacenada por `getsockopt`.

optlen: Es el tamaño de la variable especificada por el parámetro `optval`.

Opciones de Sockets Soportadas por LINUX

Opciones de Sockets Genéricas

Las opciones que se discuten a continuación son opciones de sockets genéricas e independientes de protocolos. Aunque sin embargo, como se verá, la opción de socket SO_BROADCAST por ejemplo, es tomada como genérica, aunque solo es aplicable a sockets tipo datagramas.

SO_BROADCAST habilita o deshabilita la habilidad al proceso para enviar mensajes broadcast. Broadcasting es soportado solamente por sockets no orientados a conexión y sólo en redes que soportan el concepto de un mensaje broadcast (por ejemplo Ethernet, token ring, etc).

SO_DONTROUTE indica que los mensajes salientes deben desviarse del medio estándar de enrutamiento. En lugar de eso, los mensajes son direccionados a la interfaz de red apropiada, de acuerdo a la porción de red de la dirección destino.

SO_DEBUG es soportada sólo por aplicaciones TCP. Cuando se habilita, el núcleo mantiene informaciún detallada acerca de todos los paquetes enviados o recibidos por el socket TCP.

SO_ERROR es utilizada solo por `getsockopt`. Es usada para establecer el estado de error del socket. Cuando un error ocurre en un socket, el proceso puede saber el tipo de error que se produjo consultando SO_ERROR.

SO_KEEPALIVE habilita la transmisión de mensajes en un socket orientado a conexión. Cuando no se ha dado intercambio de datos en cualquier dirección por dos horas, TCP automáticamente envía una prueba "keepalive" al host remoto. Esta prueba es un segmento TCP para el cual el host remoto debe responder. El propósito de esta opción es detectar si el host remoto esta caído.

SO_LINGER especifica cómo la función `close()` debe operar para protocolos orientados a conexión. Por defecto, `close()` retorna inmediatamente, pero si todavía hay datos en el buffer de envio del socket, el sistema trata de enviar los datos al host remoto. Esta opción permite modificar este comportamiento por defecto y utiliza la siguiente estructura a ser pasada por el proceso de usuario al núcleo:

```
struct linger
{
        int     l_onoff;    /*0 = apagado, no cero = encendido*/
        int     l_linger;   /*tiempo de demora en segundos*/
};
```

`l_onoff` indica si hay demora o no. Si esta habilitado, `l_linger` contiene el tiempo que el proceso debe esperar para completar el envio de los datos pendiente en el buffer de envio del socket después de un `close()`.

SO_OOBINLINE habilita que los datos fuera de banda (out-band-data) sean colocados en la cola normal de entrada. Cuando esto ocurre, la bandera MSG_OOB de las funciones de recepción de datos no pueden ser usadas para leer los datos fuera de banda.

SO_RCVBUF ajusta el tamaño del buffer de entrada.

SO_SNDBUF ajusta el tamaño del buffer de entrada.

SO_REUSEADDR indica que las restricciones usadas en la validación de direcciones suministradas en una llamada a `bind()` deben permitir la reutilización de las direcciones locales.

SO_TYPE devuelve el tipo de socket (SOCK_STREAM o SOCK_DGRAM).

Opciones IP de sockets

IP_TOS permite establecer el tipo de servicio en el encabezado IP para un socket TCP o UDP. Puede ser establecida por cualquiera de las siguientes constantes:

Constante	Descripción
IPTOS_LOWDELAY	Minimizar retraso
IPTOS_THROUGHPUT	Máximo rendimiento
IPTOS_RELIABILITY	Maximizar fiabilidad
IPTOS_LOWCOST	Minimizar costo

IP_TTL es utilizada para obtener o establecer el valor actual para TTL (time-to-live).

IP_OPTIONS en caso de ser habilitado, permite establecer opciones IP en el encabezado.

Desarrollo

El siguiente programa muestra cómo obtener y establecer la opción de socket SO_SNDBUF. Se imprime el valor actual del buffer de envío y luego es ajustado a 16384 bytes. Luego del ajuste se obtiene el valor actual del buffer de envío, el cual debe ser 16384 bytes. Compilar el código y ejecutarlo.

Código del programa sockopt.c

```
#include    <sys/types.h>
#include    <sys/socket.h>
#include    <netinet/in.h>
#include    <netinet/tcp.h>

main()
{
  int   sockfd, maxseg, optlen;
  int   sendbuff;

  if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
       perror("socket");

  optlen = sizeof(maxseg);

  /*Obtener valor actual de SO_SNDBUF*/

  if (getsockopt(sockfd, SOL_SOCKET, SO_SNDBUF, (char *) &sendbuff,
                 &optlen) < 0)
  perror("SO_SNDBUF getsockopt error");

  printf("Tamaño de buffer de envío por defecto = %d\n", sendbuff);

  /*Modificar valor de SO_SNDBUF a 16384 */

  sendbuff = 16384;
  if (setsockopt(sockfd, SOL_SOCKET, SO_SNDBUF, (char *) &sendbuff,
                 sizeof(sendbuff)) < 0)
  perror("SO_SNDBUF setsockopt error");

  optlen = sizeof(sendbuff);

  /*Obtener valor de SO_SNDBUF moficado*/

  if (getsockopt(sockfd, SOL_SOCKET, SO_SNDBUF, (char *) &sendbuff, &optlen)
      < 0)
  perror("SO_SNDBUF getsockopt error");

  printf("Tamaño de buffer de envio (modificado) = %d\n", sendbuff);
}
```

Asignaciones

1) Desarrollar una aplicación que muestre el estado y/o valor actual de las opciones sockets mencionadas en la guía. La salida del programa debe ser como la mostrada a continuación (el nombre supuesto del programa es `ver_opt`), la cual muestra los valores típicos para un host Linux:

```
> ver_opt
SO_BROADCAST: defecto = desactivado
SO_DEBUG: defecto = desactivado
SO_DONTROUTE: defecto = desactivado
SO_ERROR: defecto = 0
SO_KEEPALIVE: defecto = desactivado
SO_LINGER: defecto = l_onoff = 0, l_linger = 0
SO_OOBINLINE: defecto = desactivado
SO_RCVBUF: defecto = 65535
SO_SNDBUF: defecto = 65535
SO_REUSEADDR: defecto = desactivado
SO_TYPE: defecto = 1
IP_TOS: defecto = 0
IP_TTL: defecto = 64
TCP_MAXSEG: defecto = 0
TCP_NODELAY: defecto = desactivado
```

2) De todas las aplicaciones siguientes, las cuales ha sido introducidas en guías anteriores: `remoto`, `calculadora`, `transferencia de archivos`, `echo` y `daytime`, indicar cuáles de las opciones (si acaso) vistas a lo largo de la guía el lector aplicaría y/o modificaría con el objetivo de mejorar el desempeño de estas aplicaciones. También indicar el estado y/o valor que las opciones deberían tener. Explicar su respuesta.

Referencias Bibliográficas

[1] Brian W. Kernighan.
"El Entorno de Programación UNIX."
Editorial Prentice Hall, 1992.
Primera Edicion.

[2] W. Richard Stevens.
"Unix Network Programming, Networking API's Sockets and XTI".
Editorial Prentice Hall. 1998.
Segunda Edicion.

[3] Páginas del Manual de todas funciones mencionadas en la guía. .

[4] "An Advanced 4.3BSD Interprocess Communication Tutorial"
http://www.sparc.spb.su/ijb/Docs/internet/ipc-tutorial/index.html

[5] "An Advanced Socket Communication Tutorial"
http://viks.mvrop.org/networking/sock_advanced_tut.html

Guía de Laboratorio No. 8

Sockets tipo Raw (Crudos)

Introducción

Los sockets "crudos" (Raw) proveen ventajas que no se pueden adquirir con sockets normales TCP y UDP. Los *raw sockets* permiten leer y escribir paquetes ICMP e IGMP, datagramas IP y hasta contruir el encabezado IP de un datagrama. Esta guía describe como crear sockets tipo raw, utilizando como ejemplo el programa `ping` de libre distribución.

NOTA: Para la total comprensión de los conceptos planteados en esta guía, es imperativo que el estudiante domine el formato de encabezado de los paquetes IPV4 e ICMP.

Creación de socket Raw

Los pasos que involucran la creación de sockets raw son los siguientes:

1. La función `socket()` crea un socket "crudo" (raw) cuando el segundo argumento es SOCK_RAW. El tercer argumento (protocolo) es normalmente distinto de cero. Por ejemplo, para crear un socket raw se debe escribir lo siguiente:

   ```
   int sockfd;
   sockfd = socket(AF_INET, SOCK_RAW, protocolo);
   ```

   Donde `protocolo` es una de las constantes IPPROTO_xxx definidas en el archivo `<netinet/in.h>`, tales como IPPROTO_ICMP.
   Solo el superusuario puede crear un socket tipo "raw" (crudo). Esto previene a usuarios normales de escribir sus datagramas IP en la red.

2. La opción socket IP_HDRINCL puede ser puesta en caso de que se desee crear un encabezado IP directamente:

   ```
   const int on = 1;
   if (setsockopt(sockfd, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on)) < 0)
   ```

2. `bind` puede ser llamada con los socket raw, pero es raro.

3. `connect` puede ser llamada con los socket raw, pero es raro.

Programa Ping

Como ejemplo de creación de sockets tipo raw se presenta el programa `ping` de libre distribución. Debido a que esta versión del programa `ping` soporta muchas opciones, lo cual complica el código y obscurece los conceptos que se desean presentar, fué necesario modificarlo con el objetivo de presentar una versión reducida sin soporte de ninguna opción. De esta manera el estudiante se concentra en los conceptos concernientes a los sockets tipo raw introducidos en la guía.

La operación del programa `ping` es muy simple: un petición de echo ICMP es enviada a una dirección IP y el nodo remoto responde con una respuesta echo ICMP. Estos dos mensajes ICMP son soportados bajo el protocolo IP.

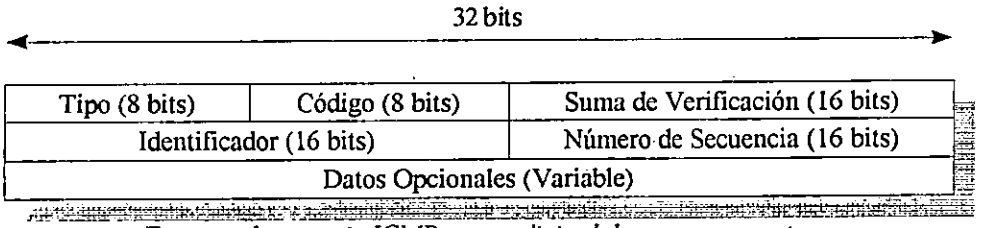
A continuación se muestra la salida típica del programa `ping` de libre distribución al consultar una dirección de Internet cualquiera.

```
PING www.navegante.com.sv (168.243.65.9) from 200.41.49.210 : 56(84) bytes of data.
64 bytes from 168.243.65.9: icmp_seq=0 ttl=108 time=1321.9 ms
64 bytes from 168.243.65.9: icmp_seq=1 ttl=108 time=1250.3 ms
64 bytes from 168.243.65.9: icmp_seq=2 ttl=108 time=1249.6 ms
...
```

Oprimimos ahora la combinación de teclas <CTRL+C> para finalizar el programa y mostrar resúmen:

```
--- www.navegante.com.sv ping statistics ---
21 packets transmitted, 20 packets received, 4% packet loss
round-trip min/avg/max = 1239.9/1259.9/1321.9 ms
```

La *Petición de Eco* y la *Respuesta de Eco* se usan para comprobar si un sistema está activo. Se usa *Tipo = 8* para la petición y *Tipo = 0* para la respuesta (ver apéndice). El destino debe devolver el mensaje recibido a su fuente. El campo *Identificador* se usa para hacer coincidir la respuesta con la petición original. En la siguiente figura se muestra el formato del mensaje eco:

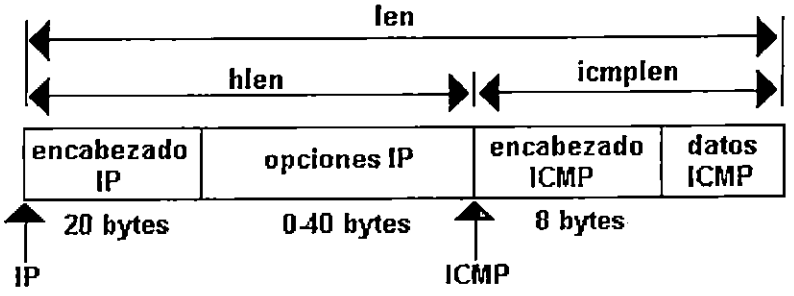


Formato de mensaje ICMP para solicitud de eco o respuesta.

Desarrollo

Digitar el código de los siguientes archivos que forman el programa `ping` de muestra. A continuación se proporcionan las debidas explicaciones para cada unos de los archivos, haciendo énfasis en los detalles concernientes al protocolo ICMP y *socket raw*, ya que como se podrá observar, dichos archivos hacen uso de funciones ya estudiadas en guías anteriores.

El archivo `ping.h` proporciona todas las definiciones de estructuras, variables y funciones de uso global en la aplicación. También hace referencia de todos los archivos de inclusión que se necesitan. Se debe observar la inclusión del archivo `ip_icmp.h` que proporciona la definición de la estructura `struct icmp` (ver apéndice), la cual proporciona los miembros necesarios para tener acceso a los campos de un mensaje ICMP.



Código del archivo ping.h

```
#include<netinet/in_systm.h>
#include<netinet/ip.h>
#include<netinet/ip_icmp.h>
#include<stdio.h>
#include<stdlib.h>
#include<errno.h>
```

```
#include<string.h>
#include<netdb.h>
#include<sys/types.h>
#include<netinet/in.h>
#include<sys/socket.h>
#include<signal.h>
#include<time.h>
#include<sys/time.h>
#define     BUFSIZE     1500
#define error(mesg)     { perror(mesg); exit(1); }

char   recvbuf[BUFSIZE];
char   sendbuf[BUFSIZE];

int          datalen;   /* #bytes de datos del header ICMP */
char        *host;      /*host hay que hacer ping*/
int          nsent;     /* adicionar 1 para cada sendto() */
pid_t        pid;       /* PID nuestro */
int          sockfd;    /*sockt raw

/* funciones prototipos */

void   proc_pack (char *, ssize_t, struct timeval *);
void   send_pack (void);
void   leer_lazo (void);
void   sig_alrm (int);
void   tv_sub (struct timeval *, struct timeval *);
unsigned short cksum( unsigned short *, int );

struct sockaddr_in   send_addr;          /* sockaddr_in{} para enviar */
struct sockaddr_in   recv_addr;          /* sockaddr_in{} para recibir */
struct hostent       *hp;
socklen_t            len_addr;           /* tamano de los sockaddr{}s */
```

Programa del programa ping (archivo ping.c)

```
#include    "ping.h"

int  main(int argc, char *argv[])
{
  if(argc != 2)
   {
     printf("Uso: ping <nombre_host>\n");
     exit(1);
   }

  datalen = 56;
  pid = getpid();
  signal(SIGALRM, sig_alrm);

  send_addr.sin_family = AF_INET;
  memset(&send_addr, 0, sizeof(send_addr));
  hp = gethostbyname(argv[1]);
  memcpy(&send_addr.sin_addr, hp->h_addr, hp->h_length);
  len_addr = sizeof(struct sockaddr_in);

  printf("PING %s (%s): %d bytes de datos\n", argv[1],
         inet_ntoa(send_addr.sin_addr) , datalen);

  leer_lazo();
```

```
  exit(0);
}
```

Código de la función leer_lazo() (archivo leer_lazo.c)

```
#include    "ping.h"

void leer_lazo(void)
{
  int              size;
  char             recvbuf[BUFSIZE];
  socklen_t        len;
  ssize_t          n;
  struct timeval   tval;

  sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
  if (sockfd < 0) error("socket");

  setuid(getuid());
  size = 60 * 1024;
  setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &size, sizeof(size));

  sig_alrm(SIGALRM);

  for (; ;)
  {
    len = len_addr;
    n = recvfrom(sockfd, recvbuf, sizeof(recvbuf), 0,
    (struct sockaddr *)&recv_addr, &len);

    if (n < 0) error("recvfrom error");

    if (gettimeofday(&tval, NULL) == -1)
    error("gettimeofday");

    proc_pack(recvbuf, n, &tval);
  }
}
```

Código de la función send_pack() (archivo send_pack.c)

```
#include    "ping.h"

void send_pack(void)
{
  int         len;
  struct icmp *icmp;

  icmp = (struct icmp *) sendbuf;
  icmp->icmp_type = ICMP_ECHO;
  icmp->icmp_code = 0;
  icmp->icmp_id = pid;
  icmp->icmp_seq = nsent++;
  if( gettimeofday((struct timeval *) icmp->icmp_data, NULL) == -1 )
  error("gettimeofday");

  len = 8 + datalen;       /* checksum para el header y datos ICMP */
  icmp->icmp_cksum = 0;
  icmp->icmp_cksum = cksum((u_short *) icmp, len);
```

```
   sendto(sockfd, sendbuf, len, 0, (struct sockaddr *)&send_addr, len_addr);
}
```

Código de la función proc_pack() (archivo proc_pack.c)

```
#include      "ping.h"

void proc_pack(char *recvbuf, ssize_t len, struct timeval *tvrecv)
{
  int             hlen, icmplen;
  double          rtt;
  struct ip       *ip;
  struct icmp     *icmp;
  struct timeval  *tvsend;

  ip = (struct ip *) recvbuf;         /* inicio del header IP */
  hlen = ip->ip_hl * 4;               /* longitud del header IP */

  icmp = (struct icmp *) (recvbuf + hlen); /* inicio del header ICMP */
  icmplen = len - hlen;

  if (icmp->icmp_type == ICMP_ECHOREPLY)
  {
     if (icmp->icmp_id != pid) return; /* Sin repuesta para el ECHO_REQUEST */

     tvsend = (struct timeval *) icmp->icmp_data;
     tv_sub(tvrecv, tvsend);
     rtt = tvrecv->tv_sec * 1000.0 + tvrecv->tv_usec / 1000.0;

     printf("%d bytes desde %s: icmp_sec=%u, ttl=%d, tiempo=%.3f ms\n",
            icmplen, inet_ntoa(send_addr.sin_addr), icmp->icmp_seq,
            ip->ip_ttl, rtt);
  }
}
```

Código de la función sig_alarm() (archivo sig_alarm.c)

```
#include    "ping.h"

void sig_alrm(int signo)
{
  send_pack();
  alarm(1);
  return;
}
```

Código de la función tv_sub() (archivo tv_sub.c)

```
include     "ping.h"

void tv_sub(struct timeval *out, struct timeval *in)
{
  if ((out->tv_usec -= in->tv_usec) < 0)
  {
     --out->tv_sec;
     out->tv_usec += 1000000;
  }
  out->tv_sec -= in->tv_sec;
}
```

Código de la función de la función cksum() (archivo cksum.c)

```
unsigned short cksum(unsigned short *addr, int len)
{
  int             nleft = len;
  int             sum = 0;
  unsigned short  *w = addr;
  unsigned short  answer = 0;

  while (nleft > 1)
  {
    sum += *w++;
    nleft -= 2;
  }

  if (nleft == 1)
  {
    *(unsigned char *)(&answer) = *(unsigned char *)w ;
    sum += answer;
  }

  sum = (sum >> 16) + (sum & 0xffff);
  sum += (sum >> 16);
  answer = ~sum;

  return(answer);
}
```

Asignaciones

1. Modificar el programa `ping` de la guía para que tenga la capacidad de mostrar el tipo de mensaje recibido (nombre del mensaje y su constante) y el tipo de código cuando el parámetro `-t` sea suministrado en la llamada al programa.
2. Modificar el programa `ping` de la guía, para que se le pueda indicar al programa por medio del parámetro `-c` el número de mensajes ECHO REQUEST a enviar al host destino.
3. Modificar el programa `ping` de la guía, para que se le pueda indicar al programa por medio del parámetro `-i` los segundos a esperar entre cada envío de mensajes *Petición de Echo.*

Referencias Bibliográficas

[1] W. Richard Stevens.
"Unix Network Programming, Networking API's Sockets and XTI".
Editorial Prentice Hall, 1998.
Segunda Edicion.

[2] Uyless Black
"Redes de Computadoras. Protocolos, Normas e Interfaces"
Editorial RA-MA, 1995.
Segunda Edición.

[3] Douglas E. Comer
"Redes Globales de Información con Internet y TCP/IP. Principios Básicos, Protocolos y Arquitectura".
Editorial Prentice-Hall Hispanoamericana, 1996.
Tercera Edición.

[4] Páginas del Manual de todas funciones mencionadas en la guía. .

[5] "The Story of the PING Program"
ftp://ftp.arl.mil/pub/ping.shar

APENDICE A: Definición de la Estructura IP

```
struct ip
{
  #if __BYTE_ORDER == __LITTLE_ENDIAN
    u_int8_t ip_hl:4;                   /* header length */
    u_int8_t ip_v:4;                    /* version */
  #endif

  #if __BYTE_ORDER == __BIG_ENDIAN
    u_int8_t ip_v:4;                    /* version */
    u_int8_t ip_hl:4;                   /* header length */
  #endif

  u_int8_t ip_tos;                      /* type of service */
  u_short ip_len;                       /* total length */
  u_short ip_id;                        /* identification */
  u_short ip_off;                       /* fragment offset field */

  #define   IP_RF 0x8000                /* reserved fragment flag */
  #define   IP_DF 0x4000                /* dont fragment flag */
  #define   IP_MF 0x2000                /* more fragments flag */
  #define   IP_OFFMASK 0x1fff           /* mask for fragmenting bits */

  u_int8_t ip_ttl;                      /* time to live */
  u_int8_t ip_p;                        /* protocol */
  u_short ip_sum;                       /* checksum */
struct in_addr ip_src, ip_dst;      /* source and dest address */
};
```

APENDICE B: Constantes para los Mensajes ICMP

```
#define ICMP_ECHOREPLY          0     /* Echo Reply */
#define ICMP_DEST_UNREACH       3     /* Destination Unreachable */
#define ICMP_SOURCE_QUENCH      4     /* Source Quench */
#define ICMP_REDIRECT           5     /* Redirect (change route) */
#define ICMP_ECHO               8     /* Echo Request */
#define ICMP_TIME_EXCEEDED      11    /* Time Exceeded */
#define ICMP_PARAMETERPROB      12    /* Parameter Problem */
#define ICMP_TIMESTAMP          13    /* Timestamp Request */
#define ICMP_TIMESTAMPREPLY     14    /* Timestamp Reply */
#define ICMP_INFO_REQUEST       15    /* Information Request */
#define ICMP_INFO_REPLY         16    /* Information Reply */
#define ICMP_ADDRESS            17    /* Address Mask Request */
#define ICMP_ADDRESSREPLY       18    /* Address Mask Reply */
```

```
        u_int8_t  id_data[1];
    } icmp_dun;

    #define  icmp_otime     icmp_dun.id_ts.its_otime
    #define  icmp_rtime     icmp_dun.id_ts.its_rtime
    #define  icmp_ttime     icmp_dun.id_ts.its_ttime
    #define  icmp_ip        icmp_dun.id_ip.idi_ip
    #define  icmp_radv      icmp_dun.id_radv
    #define  icmp_mask      icmp_dun.id_mask
    #define  icmp_data      icmp_dun.id_data
};
```

Guía de Laboratorio No. 9

Tipos de Servidor

Introducción

Una clasificación preliminar de los distintos tipos de servidores que pueden existir, atendiendo a 2 características importantes:

- Una primera característica es el tipo de comunicación empleada: orientada o no a la conexión.
- Una segunda característica, independiente de la primera, es si existe solo un proceso servidor que atienda a un solo cliente en cada instante (servidor *iterativo*), o si existen varios procesos servidores simultáneos, que atiendan a varios clientes (servidor *concurrente*).

Asi, podemos tener cuatro tipos distintos de servidores, según se indican en la tabla siguiente:

Servidor	***Iterativo***	***Concurrente***
Orientado a conexión	Anormal(daytime)	Normal(FTP)
No orientado a conexión	Normal(echo)	Anormal(tftp)

Un *servidor concurrente orientado a conexión* se emplea en aplicaciones típicas en las que el cliente establece una conexión con el servidor que dura mucho tiempo, por ejemplo: telnet, FTP, etc. En este caso hay un proceso servidor que atiende a cada cliente.

Un *servidor iterativo no orientado a conexión*, se utiliza cuando el servicio solicitado se atiende en poco tiempo, no interesando entonces ir creando servidores distintos. Las aplicaciones típicas son echo, time, finger, etc.

Las otras dos combinaciones son menos frecuentes, pero también hay aplicaciones estándar que las emplean.

Hay otros factores que hay que tener en cuenta para elegir un tipo de servidor, algunos se verán más adelante.

Se verán algoritmos para realizar cada uno de los cuatro tipos de servidores anteriores.

Servidores Iterativos No Orientados a Conexión

Los servidores iterativos son fáciles de diseñar: programar, depurar y modificar. Son los más empleados para dar una rápida respuesta a las peticiones.

El servidor iterativo no conectivo, siempre estará escuchando peticiones en un puerto conocido y todas las peticiones de los distintos clientes serán atendidas por el mismo proceso servidor. Si llegan peticiones mientras se atiende a otro cliente, el sistema operativo los guarda en una cola y después son atendidas.

El algoritmo de este servidor es muy sencillo: después de llamar a `socket()` y `bind()`, espera las peticiones con `recvfrom()` y devuelve respuesta con `sendto()` como se muestra en la siguiente figura:

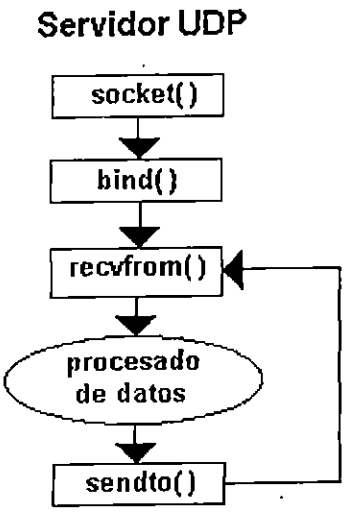


Figura 1.

Este tipo de servidores se empleará cuando el tiempo de procesamiento de una petición sea baja. Si además queremos dar una respuesta lo más rápida posible, utilizaremos un protocolo no orientado a conexión, que es mas rápido ya que no se pierde tiempo en establecer la conexión (el envío de paquetes es inmediato), las cabeceras son mas pequeñas y el tiempo de procesamiento (errores, asentimientos, control de flujo, etc) asociado al protocolo es menor.

Servidores Iterativos Orientados a Conexión

En el servidor iterativo orientado a conexión, abrimos un socket en un puerto conocido por todos los clientes esperando que lleguen peticiones. Al llegar una petición, creamos un socket nuevo para atender al cliente y nos conectamos con él mediante un puerto temporal, asignado por el sistema. Mientras se atiende a un cliente por el socket inicial, el sistema operativo almacena las nuevas peticiones de conexión que lleguan. Cuando termina un cliente, se cierra el socket actual y se abre uno nuevo si hay peticiones pendientes, que se procesan por medio de `accept()`. La figura 2 muestra el algoritmo para este tipo de servidores:

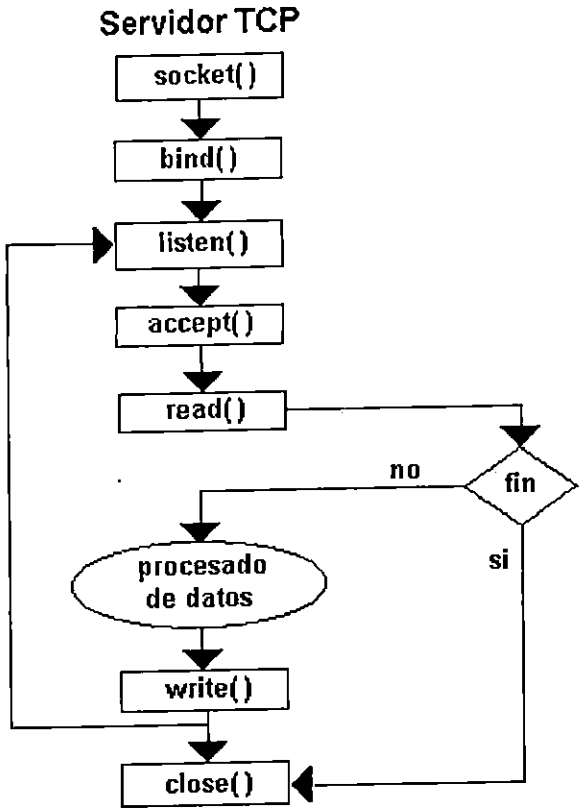


Figura 2.

Servidores Concurrentes Orientados a Conexión

Los servidores concurrentes se emplean cuando se dan algunas de las siguientes circunstancias:

- Es preciso dar respuesta rápida a las solicitudes de múltiples clientes.
- Cuando, para atender a un cliente, es preciso resolver ciertas funciones de entrada-salida de forma rápida y eficiente, atendiendo las distintas solicitudes de manera concurrente de forma que se pueda simultanear la atención por la CPU. con las transferencias de entrada-salida.
- Cuando el tiempo para atender una petición puede variar mucho. En estos casos, y con máquinas monoprocesadoras, las peticiones que requieran poco tiempo serán atendidas rápidamente, y las que requieran mucho tiempo sufrirán un breve retraso. Hay que tener en cuenta que el sistema operativo LINUX es multiproceso, de forma que todos los procesos reciben atencion.

En el servidor concurrente orientado a conexión tenemos un socket en un puerto conocido para atender a las peticiones de conexión de los clientes. Cuando llega una solicitud de conexión, el proceso padre que escucha crea un proceso hijo para atenderla, que es quien realmente se conecta con el cliente. Mientras los clientes interactúan con los procesos servidores hijos, el proceso servidor padre escucha nuevas peticiones de servicio. Cada cliente puede intercambiar varias peticiones y respuestas con su proceso servidor. Los procesos servidores hijos terminan cuando lo hacen los clientes. La figura 3 muestra el algoritmo para este tipo de servidores.

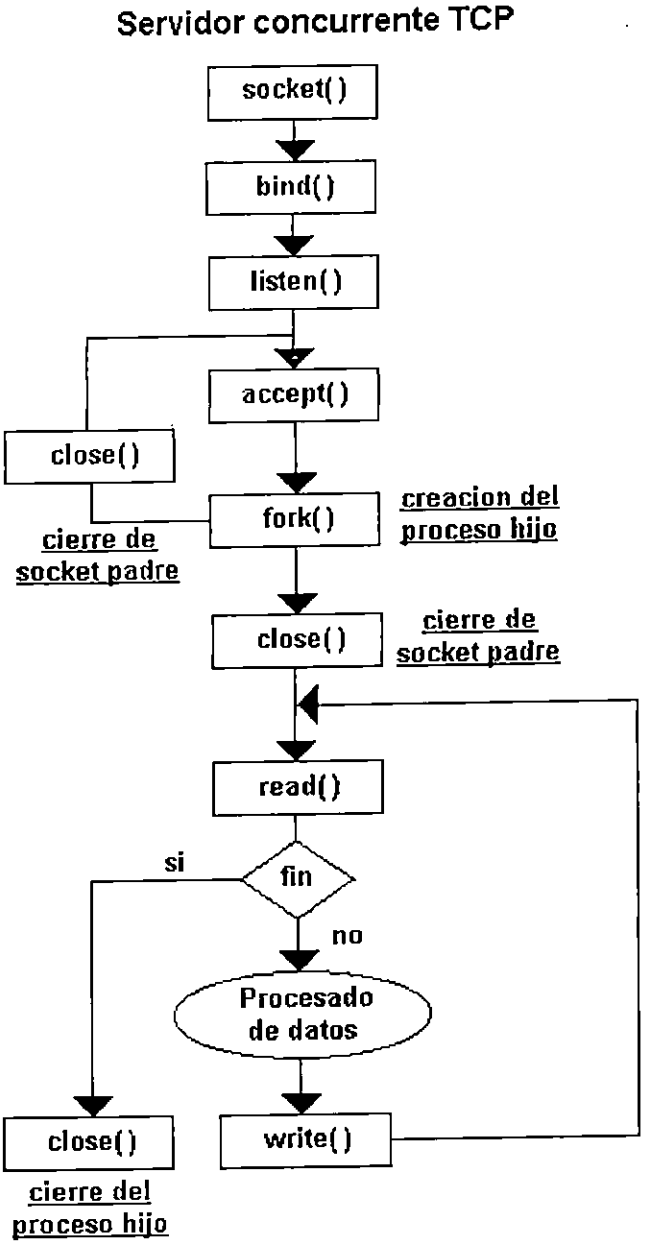


Figura 3.

Servidores Concurrentes No Orientados a Conexión

Este tipo de servidor no se utiliza mucho, ya que el algoritmo asociado a su implementación es ligermante más complicado. Su esquema de implementación se describe a continuación:

- Como siempre, se parte de un proceso servidor padre escuchando peticiones en un puerto conocido. En este caso no escucha peticiones de conexión, sino de servicio mediante `recvfrom()`.
- El cliente manda la primera petición en un datagrama UDP que contiene también su dirección IP y puerto, por lo que el proceso hijo servidor que se creará sabrá donde se encuentra el cliente.
- Tras esta primera petición, el proceso servidor padre crea un proceso servidor hijo, que toma un puerto local asignado por el sistema. Este proceso también debe cerrar la copia del socket del padre.
- El proceso servidor hijo elabora una respuesta a la primera petición del cliente, y le devuelve un datagrama UDP como respuesta, en donde figura la nueva dirección a donde el cliente debe dirigir las siguientes peticiones.

Operaciones sobre el flujo XDR

Especifica la operación que se va a realizar en el filtro con relación a los datos. Si se codifican, se utiliza XDR_ENCODE, o en el caso de la decodificación, se utiliza XDR_DECODE. El valor XDR_FREE libera en ciertas operaciones de decodificación el espacio de memoria reservado automáticamente en las operaciones anteriores.

```
enum xdr_op
{
      XDR_ENCODE = 0,
      XDR_DECODE = 1,
      XDR_FREE = 2
};
```

Creación del Flujo en Memoria

Para poder codificar o decodificar los datos, necesitamos una estructura especial que contenga los valores necesarios para su gestión: la estructura XDR.

Hay que declarar un puntero a dicha estructura al principio del fichero fuente de la aplicación para luego inicializarla. Esto previamente a cada codificación o decodificación, por medio de la siguiente función:

```
#include <rpc/types.h>
#include <rpc/xdr.h>

void xdrmen_create (xdrptr, buffer, len, operación)
       XDR          *xdrptr;     /* Dirección de la estructura XDR */
       char         *buffer;     /* Dirección del buffer (datos a tratar) */
       int          len;         /* Máxima longitud de los datos a tratar */
       enum xdr_op  operación;   /* Tipo de operación a realizar */
```

Donde el tipo de operación a realizar puede ser alguna de las tres constantes XDR_ENCODE, XDR_DECODE o XDR_FREE, vistas anteriormente.

Destrucción de un Flujo

Una vez realizada la operación con los datos, conviene liberar los recursos y datos de gestión empleados por el flujo, empleando la llamada:

```
void xdr_destroy(xdrp)
          XDR      *xdrp;
```

Longitud de los Datos Convertidos

Se obtiene utilizando uno de los procedimientos definidos sobre el flujo. El procedimiento es:

```
int xdr_getpos(xdrp)
          XDR      *xdrp;            /*puntero al descriptor XDR*/
```

Este devuelve la longitud de los datos convertidos. Debemos emplearlos para saber cuántos bytes hay que enviar. En ocasiones es sencillo saber cual es la longituid de los datos convertidos. Por ejemplo, un entero de 2 bytes se convertirá en 4 bytes, pero para estructuras complejas es útil esta función.

Operaciones de Codificación y Decodificación

Son operaciones que se realizan a través de un filtro, y en ellas se extrae o añade información de un flujo XDR. Es preciso recordar que los filtros son generados por el compilador especial **rpcgen**, con el nombre `xdr_nombreaplicación`.

Las llamadas a las funciones filtro son prácticamente iguales, en lo único que varían es en el nombre (*nombreaplicación*), y tienen la forma genérica siguiente:

```
bool_t    aplicación_xdr(xdrs, datos_aplicación)
          XDR     xdrs;              /* Dirección del descriptor XDR */
          <tipo>  datos_aplicación; /* Dirección de los datos a convertir */
```

Siempre pasaremos las direcciones del descriptor XDR y la de los datos a convertir, que es distinta a la del buffer asociado con el descriptor XDR.

La ejecución del filtro devuelve un tipo `bool_t`, que pueden tomar los valores, 1 (TRUE) ó 0 (FALSE), si se realizó correctamente o no. Un error en la ejecución puede deberse a errores en la conversión de datos o a fallos en el descriptor XDR por falta de espacio para hacer la conversión.

Filtros XDR de Conversión

Son los que vienen definidos en las librerías XDR, encargados de hacer las conversiones básicas. Los filtros creados por `rpcgen` se apoyan en ellos. Pueden dividirse en dos categorías:

- **Simples:** convierten directamente entre tipos en C y en XDR.
- **Complejos:** para convertir estructuras de más dificultad.

Los nombres de los filtros siguen una regla muy simple, siendo para un tipo determinado `xdr_tipo`. Lo primeros se resúmen en la tabla siguiente:

TIPO XDR	*FILTRO*
`bool`	`xdr_bool`
`char`	`xdr_char`
`short`	`xdr_short`
`unsigned short`	`xdr_u_short`
`int`	`xdr_int`
`unsigned int`	`xdr_u_int`
`long`	`xdr_long`
`unsigned long`	`xdr_u_long`
`float`	`xdr_float`
`double`	`xdr_double`
`void`	`xdr_void`
`enum`	`xdr_enum`

Tabla 1

Para los tipos complejos están disponibles los siguientes filtros:

TIPO XDR	*FILTRO*
Arreglo con longitud variable, y elementos de longitud variable	`xdr_array`
Arreglo de bytes de longitud variable	`xdr_bytes`
Datos opacos de longitud fija	`xdr_opaque`
Referencias a punteros	`xdr_pointer`
Referencias a objetos	`xdr_reference`
Arreglo de caracteres, terminados por NULL	`xdr_string`
Uniones discriminantes	`xdr_union`
Arreglo de longitud fija con elementos de longitud variable	`xdr_vector`
Arreglo de caracteres de longitud variable, terminados por NULL	`xdr_wrapstring`

Tabla 2

A modo de resúmen, cuando necesitamos enviar datos en formato de red hay que realizar los pasos siguientes:

- Crear un descriptor con la llamada `xdr<tipo>_create()`.
- Llamar al filtro para hacer la conversión.
- Calcular la longitud de los bytes convertidos, con `xdr_getpos()`.
- Destruir el descriptor con `xdr_destroy()`.

Cuando recibamos datos las operaciones son similares:

- Crear un descriptor con la llamada `xdr<tipo>_create()`.
- Llamar al filtro para hacer la conversión.
- Destruir el descriptor con `xdr_destroy()`.

Previamente hay que generar el archivo de cabecera y el archivo filtro con el compilador `rpcgen`. Los archivos que vamos a utilizar tendrán los siguientes nombres:

CONTENIDO	*ARCHIVO*
Especificaciones de estructuras en XDR	`aplicacion.x`
Cabeceras (*)	`aplicacion.h`
Filtros XDR (*)	`aplicacion_xdr.c`
Cliente	`aplicación_cliente,c`
Servidor	`aplicacione_servidor.c`

(*): Pueden ser generados por el compilador `rpcgen`.

Uso del compilador rpcgen

Previamente hay que generar el fichero de cabecera y el filtro con el compilador `rpcgen`.

Veamos un ejemplo para una aplicación:

Lo primero es definir la estructura anterior en el lenguaje XDR, archivo `banco.x`

```
struct transferencia
{
     string        beneficiario<20>;
     unsigned int  num_cuenta;
     float         cantidad;
     opaque        comprobación<30>;
};
```

Esta información hay que pasarla por un compilador especial: el `rpcgen` para traducirla al lenguaje en que escribimos la aplicación bancaria (lenguaje C). Ejecutaremos la orden:

```
$ rpcgen -h banco.x > banco.h
```

Obtendremos la estructura equivalente en C, archivo `banco.h` :

```
#include <rpc/rpc.h>

struct transferencia
{
      char *beneficiario;
      u_int num_cuenta;
      float cantidad;
```

```
        struct
        {
                u_int comprobacion_len;
                char *comprobacion_val;
        } comprobacion;
};

typedef struct transferencia transferencia;
extern bool_t xdr_transferencia(XDR *, transferencia*);
bool_t xdr_transferencia();
```

La traducción de los tipos XDR a C se puede intuir. El archivo `banco.h` suministra una declaración de la estructura transferencia, y la definición de la función filtro, `xdr_transferencia`, que devuelve un valor booleano que indica el resultado de la ejecución. Este archivo de cabecera debe ser incorporado a nuestra aplicación, insertandolo al principio de los archivos fuente del cliente y servidor, `banco.c` y `bancos.c`, es decir. la siguiente línea:

```
#include "banco.h"
```

Además. el compilador `rpcgen` nos suministra la función filtro para codificar y decodificar los datos. Ejecutaremos para ello la orden:

```
$ rpcgen -c banco.x > banco_xdr.c
```

y obtenemos el filtro correspondiente en el archivo `banco_xdr.c` :

```
#include <rpc/types.h>
#include <rpc/xdr.h>
#include "banco.h"

bool_t xdr_transferencia(XDR *xdrs, transferencia *objp)
{
        register long *buf;

        if (!xdr_string(xdrs, &objp->beneficiario, 20))
                return (FALSE);

        if (!xdr_u_int(xdrs, &objp->num_cuenta))
                return (FALSE);

        if (!xdr_float(xdrs, &objp->cantidad))
                return (FALSE);

        if (!xdr_bytes(xdrs, (char **)&objp->comprobacion.comprobacion_val,
                (u_int*)&objp->comprobacion.comprobacion_len, 30))
                return. (FALSE);

       return (TRUE);
}
```

Esta función será invocada desde nuestra aplicación cuando queramos enviar o recibir datos, para codificarlos o decodificarlos, y le pasaremos la dirección de los datos a tratar así como la operación a afectuar. La generación del filtro es trivial. cada tipo elemental tiene su filtro. Por ejemplo, para `float` esta `xdr_float`. Así, el filtro de la estructura se hace en base a los filtros de los tipos elementales que la componen.

En el archivo fuente del cliente utilizaremos la estructura y el filtro como sigue:

```
#include "banco.h"
...
/*introducir valores de la estructura transferencia*/
```

```
...
/*codificarlos para enviarlos*/
if(!xdr_transferencia(&operacion, &transferencia))
      {
            perror("fallo en la conversion);
            exit(1);
      }

sendto(...);

/*esperar respuesta*/
....
```

En el archivo fuente del servidor se hace el proceso inverso:

```
#include"banco.h"
...
recvfrom(...);
/*decodificación para procesarlos*/

if(!xdr_transferencia(&operacion, &transferencia))
      {
            perror("fallo en la conversion);
            exit(1);
      }

/*envio de respuesta*/
...
```

Los tres archivos `banco.h`, `banco_xdr.c` y `banco_cliente.c` serán compilados para generar el ejecutable cliente. mientras que los archivos `banco.h`, `banco_xdr.c` y `banco_servidor.c` se compilan para generar el servidor.

Desarrollo

La aplicación cliente-servidor siguiente utiliza el protocolo XDR para la transferencia de información. El programa Cliente envía una cadena al programa servidor para que éste lo imprima en pantalla. A este programa cliente se le debe proporcionar la dirección IP del servidor en notación de puntos y la cadena o mensaje a enviar.

Digitar el código del programa cliente y del programa servidor, compilarlos y probar la aplicación.

Código del programa cliente (archivo cliente.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <rpc/types.h>
#include <rpc/xdr.h>

#define MIPORT    4950
#define MAXLINEA  100
#define error(mesg){ perror(mesg); exit(1); }
```

```
int main(int argc, char *argv[])
{
  int sockfd;
  struct sockaddr_in serv_addr;
  int numbytes;
  XDR p;
  char buf[MAXLINEA];
  char *ptr = argv[2];

  if (argc != 3)
  {
      fprintf(stderr,"Uso: cliente <Direccion IP> \"mensaje\"\n");
      exit(1);
  }

  if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) error("socket");

  serv_addr.sin_family = AF_INET;
  serv_addr.sin_port = htons(MIPORT);
  serv_addr.sin_addr.s_addr = inet_addr(argv[1]);
  bzero(&(serv_addr.sin_zero), 8);

  xdrmem_create(&p, buf, MAXLINEA, XDR_ENCODE);
  xdr_string(&p, &ptr, strlen(argv[2]));

  if ((numbytes=sendto(sockfd, argv[2], xdr_getpos(&p) , 0,
       (struct sockaddr*)&serv_addr, sizeof(struct  sockaddr))) == -1)
  error("sendto");

  xdr_destroy(&p);

  printf("Enviando %d bytes a %s\n",numbytes,inet_ntoa(serv_addr.sin_addr));
  close(sockfd);
  return 0;
}
```

Código del programa servidor (archivo servidor.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <rpc/types.h>
#include <rpc/xdr.h>
#define MIPORT    4950
#define MAXBUFLEN       100
#define error(mesg){ perror(mesg); exit(1); }

main()
{
  int sockfd;
  struct sockaddr_in serv_addr;
  struct sockaddr_in cli_addr;
  int addr_len, numbytes;
  char buf[MAXBUFLEN], buffer[MAXBUFLEN];
  XDR p;
```

```
char *ptr = buffer;

if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) error("socket");

serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(MIPORT);
serv_addr.sin_addr.s_addr = INADDR_ANY;
bzero(&(serv_addr.sin_zero), 8);

if (bind(sockfd, (struct sockaddr *)&serv_addr, sizeof(struct sockaddr)) == -1)
error("bind");

for( ; ;)
{
   addr_len = sizeof(struct sockaddr);
   xdrmem_create(&p, buf, MAXBUFLEN , XDR_DECODE);

   if ((numbytes=recvfrom(sockfd, buffer , MAXBUFLEN,0,
        (struct sockaddr*)&cli_addr, &addr_len)) == -1)
   error("recvfrom");

   xdr_string(&p, &ptr,strlen(buffer));
   xdr_destroy(&p);

   printf("Paquete obtenido desde %s\n",inet_ntoa(cli_addr.sin_addr));
   printf("El paquete es %d bytes de largo\n",numbytes);
   buf[numbytes] = '\0';
   printf("El paquete contiene \"%s\"\n",buffer);
```

```
char *ptr = buffer;

if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) error("socket");

serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(MIPORT);
serv_addr.sin_addr.s_addr = INADDR_ANY;
bzero(&(serv_addr.sin_zero), 8);

if (bind(sockfd, (struct sockaddr *)&serv_addr, sizeof(struct sockaddr)) == -1)
error("bind");

for( ; ;)
{
   addr_len = sizeof(struct sockaddr);
   xdrmem_create(&p, buf, MAXBUFLEN , XDR_DECODE);

   if ((numbytes=recvfrom(sockfd, buffer , MAXBUFLEN,0,
        (struct sockaddr*)&cli_addr, &addr_len)) == -1)
   error("recvfrom");

   xdr_string(&p, &ptr,strlen(buffer));
   xdr_destroy(&p);

   printf("Paquete obtenido desde %s\n",inet_ntoa(cli_addr.sin_addr));
   printf("El paquete es %d bytes de largo\n",numbytes);
   buf[numbytes] = '\0';
   printf("El paquete contiene \"%s\"\n",buffer);
}
close(sockfd);
return(0);
```

Asignaciones

Modificar las programas distribuidos "remoto", "calculadora", "copia de archivo" y "eco" para que sean independientes del tipo de representación con el protocolo XDR.
Se recomienda utilizar el compilador `rpcgen` para la conversión de formato en caso de utilizarse estructuras en los programas anteriores.

Referencias Bibliográficas

[1] RFC 1832 (Request for Comments). Especificación del estandar XDR.

[2] Jean-Marie Rifflet
"Comunicaciones en UNIX".
Editorial McGraw-Hill, 1992.
Primera Edición.

[3] Páginas del Manual `rpcgen` y `xdr`.

[4] Cisco IOS for S/390 RPC/XDR Programmer's Reference"
http://www.cisco.com/univercd/cc/td/doc/product/software/ios390/hos390rp

[5] Communications Programming Concepts". Primera Edición.
http://anguilla.u arizona.edu/doc_link_en_US/a_doc_lib/aixprggd/progcoms

[6] "DIGITAL TCP/IP Services for OpenVMS ONC RPC Programming"
http://cetr.umkc.edu/vms/72final/6528/

APENDICE A: Prototipos de los Filtros Suministrados por XDR

```
bool_t  xdr_void ((void));
bool_t  xdr_int ((XDR *__xdrs, int *__ip));
bool_t  xdr_u_int ((XDR *__xdrs, u_int *__up));
bool_t  xdr_long ((XDR *__xdrs, long *__lp));
bool_t  xdr_u_long ((XDR *__xdrs, u_long *__ulp));
bool_t  xdr_short ((XDR *__xdrs, short *__sp));
bool_t  xdr_u_short ((XDR *__xdrs, u_short *__usp));
bool_t  xdr_bool ((XDR *__xdrs, bool_t *__bp));
bool_t  xdr_enum ((XDR *__xdrs, enum_t *__ep));
bool_t  xdr_array ((XDR *_xdrs, caddr_t *__addrp, u_int *__sizep, u_int __maxsize,
                    u_int __elsize, xdrproc_t __elproc));
bool_t  xdr_bytes ((XDR *__xdrs, char **__cpp, u_int *__sizep, u_int __maxsize));
bool_t  xdr_opaque ((XDR *__xdrs, caddr_t __cp, u_int __cnt));
bool_t  xdr_string ((XDR *__xdrs, char **__cpp, u_int __maxsize));
bool_t  xdr_union ((XDR *__xdrs, enum_t *__dscmp, char *__unp,
                   struct xdr_discrim *__choices, xdrproc_t dfault));
bool_t  xdr_char ((XDR *__xdrs, char *__cp));
bool_t  xdr_u_char ((XDR *__xdrs, u_char *__cp));
bool_t  xdr_vector ((XDR *__xdrs, char *__basep, u_int __nelem, u_int __elemsize,
                    xdrproc_t __xdr_elem));
bool_t  xdr_float ((XDR *__xdrs, float *__fp));
bool_t  xdr_double ((XDR *__xdrs, double *__dp));
bool_t  xdr_reference ((XDR *__xdrs, caddr_t *__xpp, u_int __size,
                   xdrproc_t __proc));
bool_t  xdr_pointer ((XDR *__xdrs, char **__objpp, u_int __obj_size,
                   xdrproc_t __xdr_obj));
bool_t  xdr_wrapstring ((XDR *__xdrs, char **__cpp));
```

Guía de Laboratorio # 11

Programación con RPC en Nivel Medio

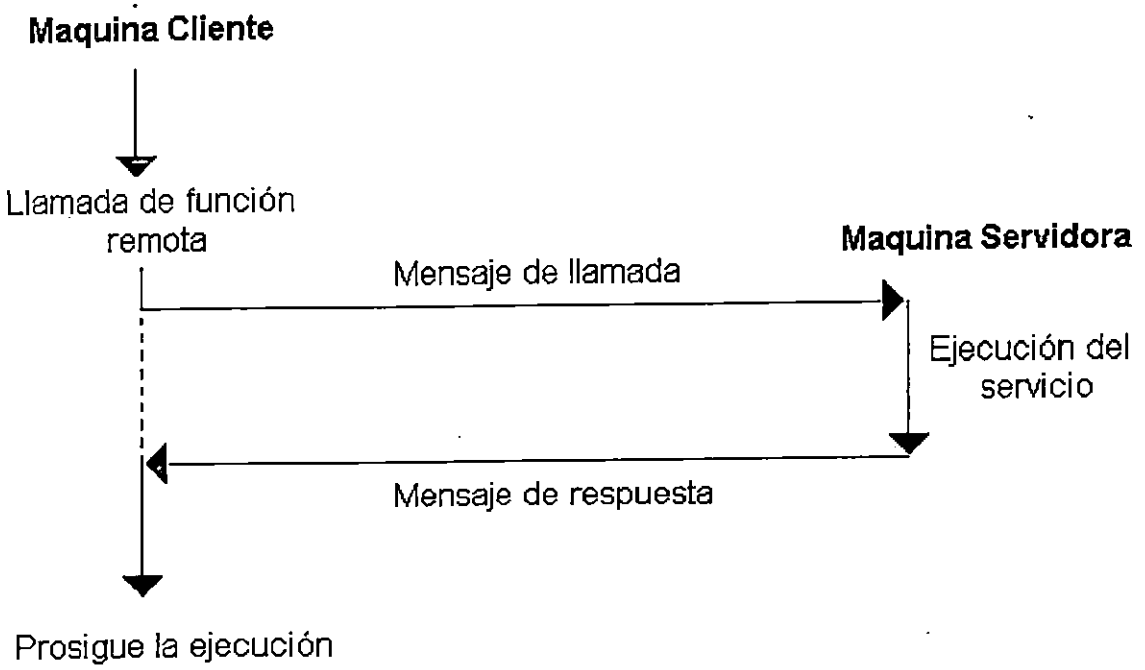
Introducción

Este protocolo ha sido definido por Sun Microsystems con vista a la implementación del sistema de archivos distribuidos NFS (Network File System). El concepto general de RPC (Remote Procedure Call) constituye una extensión de una llamada a un procedimiento local: el principio es permitir a un proceso, en el curso de su ejecución, una llamada de procedimiento o función, eventualmente con parámetros y valores de vuelta, cuya ejecución tenga lugar en otra máquina. Las llamadas a funciones remotas dan lugar a intercambios de mensajes sobre la red. La transmisión de los parámetros y de los resultados de las funciones se apoyan sobre la representación estandar XDR estudiada en la guía #10.

Uno de los intereses, y no el menor, es el de evitar al programador la manipulación de los sockets para la mayoría de las aplicaciones.

Principio General

La ejecución de la llamada remota se lleva a cabo según se muestra en la siguiente figura:



El proceso servidor estará dormido, escuchando con `select()` en un puerto determinado, esperando a que se le soliciten servicios. Cuando llegua una petición la atiende, ejecutando un servicio y devolviendo los resultados, volviendo al estado inicial, en espera de nuevas solicitudess.

Por su parte, el cliente esperará bloqueado hasta que el servidor le devuelva la respuesta.

Todas las llamadas del nivel XDR están ocultas en unos procedimientos llamados ***cabos*** (***stubs***). Estos cabos, uno para el cliente y otro para el servidor, se generán mediante un compilador de RPC a partir de una especificación en lenguaje RPC. En dicha especificación se indican algunas características del servidor, como por ejemplo las estructuras que se van a utilizar. El lenguaje RPC es un extensión al lenguaje XDR.

Números de Programa, Versión y Procedimiento

Bajo el protocolo RPC, a un programa se le identifica con un número entero de 32 bits, y a cada uno de los procedimientos del programa con otro entero de 32 bits. Este también es el caso del número de procedimiento.

Así pues, para solicitar una llamada a un procedimiento remoto necesitamos especificar las siguientes coordenadas RPC:

- Número de programa
- Número de version
- Número de procedimiento

En todo programa servidor el procedimiento número 0 está reservado, y se invoca sólo para comprobar que el servidor esta disponible.

Números de Programas

Son números enteros largos (long int) comprendidos entre los valores hexadecimales 00000000 y FFFFFFFF. No todos los valores pueden utilizarse, ya que existen varios sub-segmentos reservados como lo muestra la siguiente tabla:

RANGO EN HEXADECIMAL	*USO*
00000000 a 1FFFFFFF	Definidos por Sun
20000000 a 3FFFFFFF	A definir por los usuarios
40000000 a FFFFFFFF	Reservado

El Proceso portmap

Existe un problema ante un programa servidor RPC dado que consiste en averiguar por cuál puerto estará el servidor escuchando. Para resolver está incertidumbre se utiliza el proceso `portmap()`, que asocia un número de puerto a un servicio RPC.
El proceso `portmap` es, en sí mismo, un servicio RPC, cuyo número es el 100000, y cuyo puerto es el 111. Este servicio debe estar activo para que el mecanismo general RPC sea accesible. Los pasos que se siguen para el uso de `portmap` son:

- El servidor solicita la dirección de escucha (el número de puerto) a `portmap`, y éste le asigna una.
- El cliente, antes de solicitar el servicio, consulta `portmap` en la máquina servidora para saber el número de puerto del servicio.
- El cliente y el servidor establecen una comunicación. El cliente realiza una peticion y el servidor envía la respuesta.

Un servicio RPC activo está disponible para cualquier cliente. Previamente el proceso servidor indicará esta disponibilidad al proceso `portmap` de la máquina servidora.

El proceso `portmap` entrará en funcionamiento en la secuencia de arranque antes que el proceso `inetd`, ya que existen servicios RPC que utilizan este proceso.

Niveles de Utilización de RPC.

Pueden utilizarse tres niveles distintos a la hora de implemetar programas cliente-servidor mediante mecanismos RPC. Cada nivel ofrece mayor o menor tranparencia al programador, y será este, en función de las necesidades y conocimientos que tenga, el encargado de escoger el nivel mas adecuado.

- **Nivel alto.**
  Es el que oculta más detalles al programador. Unicamente se relizará una llamada a las funciones de biblioteca indicando el nombre de la máquina destino, asi como otros parámetros según la función en cuestión. Este nivel tiene una limitación importante: el programador no puede desarrollar sus propios servicios, sino solo utilizar los que hay disponibles de manera estándar.

- **Nivel medio.**
  Evita al programador el tener que trabajar con sockets y/o XDR. Contrariamente, pernite utilizar únicamente el protocolo UDP. Tampoco se puede cambiar el valor por defecto de ciertos parámetros. Requiere conocimientos mínimos de los protocolos XDR y RPC. Es el nivel más interesante y desde donde se realizan la mayoría de aplicaciones RPC.

- **Nivel bajo.**
  La programación es más compleja., pero permite el mayor grado de libertad permitiendo el desarrollo de programas en UDP y TCP, cambiar valores de los parámetros por defecto, etc. Este nivel será cubierto con mayor detalle en la guía siguiente.

Nivel Alto

Permite ocultar al máximo los detalles al usuario. Se dispone de un número de funciones estándar que permiten la realización de programas simples. Para ilustrar una llamada a nivel alto, un programa puede simplemente llamar a la rutina `rnusers` (función RPC estándar disponible), una rutina en C que devuelve el número de usuarios en una estación remota.

Algunas otras rutinas de librería de servicio RPC disponibles a los programadores en C son:

rusers	Devuelve información acerca de usuarios en una estación remota.
havedisk	Determina si la estación remota tiene disco.
rstat	Obtiene datos de desempeño desde un kernel remoto.
rwall	Envía un mensaje a una estación remota dada.

Nivel intermedio

Permite escribir nuevos servicios de manera simple apoyándose solo en el protocolo UDP, por lo que el tamaño de los mensajes intercambiados estarán limitados. Así, cuando se desee trabajar con un protocolo orientado a conexión, y/o utilizar mensajes de intercambio de mayor longitud, y/o cambiar los valores por defecto de ciertos parámetros (por ejemplo el *timeout*) se deberá utilizar el nivel bajo de programación.

Las operaciones a realizar para implementar una aplicación cliente-servidor en este nivel serán:

En el servidor:

- Escribir los diversos procedimientos del servicio.
- Asignar un número de programa y servicio.
- Registrar los distintos procedimientos del programa servidor en el proceso `portmap`, con la función `registerrpc()`.
- Esperar las peticiones de los clientes, por medio de `svcrun()`.

En el cliente:

- Realizar la llamada mediante `callrpc()`.

Llamadas en el servidor

El Registro de un Procedimiento por Medio de registerrpc().

Mediante esta llamada se indica la existencia de un procedimiento al proceso portmap. Cada función debe anotarse de forma individual por medio de la llamada a esta función como sigue:

```
#include<rpc/rpc.h>

int registerrpc   (num_programa, num_version, num_procedimiento,
                   función, xdr,_arg, xdr_res)

unsigned long    num_programa, num_version, num_procedimiento;
char             *(*funcion);
bool_t           (*xdr_arg)(), (*xdr_res)();
```

Con esta función se registrará la función `*funcion`, con número `num_procedimiento`, con la versión `num_version` del número de programa `num_programa`. Con `*xdr_arg` y `*xdr_res` indicamos el filtro que deseamos aplicar a los argumentos de la llamada y a los resultados respectivamente. Devuelve 0 en caso de éxito y -1 en caso contrario.

Una vez registrado, cualquier cliente puede llamar a este procedimiento preguntándole a `portmap` usando el número de procedimiento, de versiún y de programa, quien nos responderá con el puerto en donde esta escuchando.

Espera de solicitudes con la llamada svc_run()

El registrar un procedimiento no significa que esté disponible. Un vez el servidor comienza y registra sus distintos procedimientos mediante `registerrpc()`, con la función `svc_run()` éste se "duerme" quedando a la escucha por un socket, esperando a que se efectúe alguna llamada a los procedimientos registrados. Cuando llegue alguna petición de ejecución de un procedimiento, se despertara para atender la petición y volverá de nuevo al estado dormido.

La llamada a la función es muy sencilla:

```
void svc_run();
```

Esta función no devuelve nada, es decir, el programa servidor se queda bloqueado en este punto salvo que ocurra un error en la llamada, devolviendo el control.

Llamadas en el cliente

En el lado del cliente la programación no puede ser mas sencilla, tan solo es hacer la llamada al procedimiento.

La llamada al procedimiento remoto con callrpc()

Veamos cómo es la llamada para solicitar la ejecución de un procedimiento remoto :

```
int callrpc(máquina, num_programa, num_version, num_procedimiento,
            xdr_arg, arg, xdr_res, res);

char             *máquina;
unsigned long    num_programa, num_version, num_procedimiento;
char             *arg, *res;
bool_t           (*xdr_arg)(), (*xdr_res)();
```

Hay que indicar en qué ordenador se encuentra el servidor, su número de programa y versión, así como el procedimiento a ejecutar mediante `num_procedimiento`. El argumento `arg` apunta a los parámetros del procedimiento cuyo filtro de conversión viene indicado por la funcion `*xdr_arg`. De la misma forma, `res` es el puntero donde quedará la dirección de los resultados de la llamada y el filtro de conversión de los mismos será la función especificada `*xdr_res`.

La llamada a esta función es bloqueante. El valor devuelto es 0 en caso de éxito, y un valor distinto de cero en caso contrario.

Desarrollo

El siguiente ejemplo implementa una aplicación cliente-servidor utilizando programación RPC nivel medio cuya función es emular una calculadora distribuida. Al programa cliente se le proporcionará en la llamada el nombre del servidor y después se le proporcionarán los datos y la operación a realizar. Las operaciones disponibles serán las cuatro operaciones aritméticas básicas (+, -, *, /), las cuales serán procesadas por el programa servidor. El trabajo del cliente será suministrar los operandos y el operador al programa servidor. La calculadora debe tener la capacidad de procesar una operación aritmética a la vez hasta que se digite `s` (salir), lo cual hace que termine el proceso cliente.

La ejecución del programa cliente debe tener el aspecto siguiente:

```
> calcu 169.168.0.5
169.168.0.5> 5*50
169.168.0.5> 250
169.168.0.5> 10-50
169.168.0.5> 40
169.168.0.5> s
> cliente terminado.
```

El archivo `calcu.x` de definición de estructuras de datos, es el siguiente:

Archivo de definición XDR calcu.x

```
struct num
{
      float a;
      float b;
};
```

Utilizar el compilador `rpcgen` para generar el archivo `calcu.h`. Modificarlo de la siguiente manera para incluir el número de programa, versión de programa, cabeceras para RPC, y constantes globales. Las líneas resaltadas es el código a digitar (no es generado por `rpcgen`).

```
#rpcgen -h calcu.x > calcu.h
```

Archivo de inclusión calcu.h

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#ifndef _CALCU_H_RPCGEN
#define _CALCU_H_RPCGEN

#include <rpc/rpc.h>
#include <rpc/xdr.h>
#include <rpc/types.h>
```

```
#include <stdio.h>
#define     CALCU              0x32000000
#define     CALCU_VERSION      1
#define     SUMAR              1
#define     RESTAR             2
#define     MULTIPLICAR        3
#define     DIVIDIR            4

struct num
{
      float a;
      float b;
};

typedef struct num num;
#ifdef __cplusplus
extern "C" bool_t xdr_num(XDR *, num*);
#elif __STDC__
extern bool_t xdr_num(XDR *, num*);
#else /* Old Style C */
bool_t xdr_num();

char *sumar(struct num *);
char *restar(struct num *);
char *multiplicar(struct num *);
char *dividir(struct num *);

#endif /* Old Style C */

#endif /* !_CALCU_H_RPCGEN */
```

Archivo filtro calcu_xdr.c

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include <rpc/types.h>
#include <rpc/xdr.h>

#include "calcu.h"

bool_t
xdr_num(XDR *xdrs, num *objp)
{
   register long *buf;

   if (!xdr_float(xdrs, &objp->a))
   {
      return (FALSE);
   }
   if (!xdr_float(xdrs, &objp->b))
   {
      return (FALSE);
   }
   return (TRUE);
}
```

Digitar el código de los programas cliente y servidor de la calculadora RPC. Crear los programas ejecutables cliente y servidor compilando los archivos necesarios. Ejecutar apropiadamente la aplicación en la red para verificar su correcto funcionamiento.

Código del cliente calcu_cli.c

```
#include "calcu.h"
#include <stdlib.h>

#define     MAXIMO sizeof(struct num)
#define     SUM    '+'
#define     RES    '-'
#define     MUL    '*'
#define     DIV    '/'

main(int argc, char *argv[])
{
  struct      num valor;
  int   op, n;
  float       res;
  char  cad[MAXIMO], car;

  system("clear");
  if(argc != 2)
  {
     printf("\n");
     printf("Uso: calcu nombre_host\n");
     printf("Ejemplo: calcu localhost\n");
     exit(2);
  }

  printf("  ********************CALCULADORA DISTRIBUIDA*********************");
  printf("\nEste programa realiza con"
         " dos operandos una de las siguientes operaciones:\n"
         "\n + Suma\n - Resta\n * Multiplicación\n / División\n"
         " s Salir\n c Limpiar\n"
         "\nPrimer operando, operador, segundo operando:"
         "\nEjemplo: 10 * 30\n\n");

  while(1)
  {
     printf("%s> ", argv[1]);
     gets(cad);

     if(!strcmp(cad,"s")) exit(1);
     if(!strcmp(cad,"c")) system("clear");
     else
     {
       sscanf(cad,"%f %c %f", &valor.a, &car, &valor.b);

     switch(car)
     {
       case SUM:
             n =   callrpc(argv[1], CALCU, CALCU_VERSION, SUMAR,
              xdr_num, (char *)&valor, xdr_float, (char *)&res);
              break;
       case RES:
              n =   callrpc(argv[1], CALCU, CALCU_VERSION, RESTAR,
              xdr_num, (char *)&valor, xdr_float, (char *)&res);
              break;
```

```
        case MUL:
                n =   callrpc(argv[1], CALCU, CALCU_VERSION, MULTIPLICAR,
                xdr_num, (char *)&valor, xdr_float, (char *)&res);
                break;
        case DIV:
                n =   callrpc(argv[1], CALCU, CALCU_VERSION, DIVIDIR,
                xdr_num, (char *)&valor, xdr_float, (char *)&res);
                break;
        default:
                printf("\n%s> Operacion no valida\n", argv[1]);
                exit(1);
        }

        if(n == 0)
        printf("%s> %f \n", argv[1], res);

        else fprintf(stderr, "error callrpc");
      }
   }
   return(0);
}
```

Código del servidor calcu_srv.c

```
include "calcu.h"

char *sumar(struct num*);
char *restar(struct num*);
char *multiplicar(struct num*);
char *dividir(struct num*);

main()
{
   int n;

   n = registerrpc(CALCU, CALCU_VERSION, SUMAR, sumar, xdr_num, xdr_num);
   if(n == -1)
   {
      fprintf(stderr, "error en registerrpc sumar");
      exit(1);
   }

   n = registerrpc(CALCU, CALCU_VERSION, RESTAR, restar, xdr_num, xdr_num);
   if(n == -1)
   {
      fprintf(stderr, "error en registerrpc restar");
      exit(1);
   }

   n = registerrpc(CALCU, CALCU_VERSION, MULTIPLICAR, multiplicar,
   xdr_num, xdr_num);
   if(n == -1)
   {
      fprintf(stderr, "error en registerrpc multiplicar");
      exit(1);
   }

   n = registerrpc(CALCU, CALCU_VERSION, DIVIDIR, dividir, xdr_num, xdr_num);
   if(n == -1)
```

```
    {
      fprintf(stderr, "error en registerrpc dividir");
      exit(1);
    }

    svc_run();
    fprintf(stderr,"error en svc_run\n");
    exit(2);
}

char *sumar(struct num *p)
{
    static struct num res;
    res.a = p->a + p->b;
    return((char *)&res);
}

char *restar(struct num *p)
    {
    static struct num res;
    res.a = p->a - p->b;
    return((char *)&res);
}

char *multiplicar(struct num *p)
{
    static struct num res;
    res.a = p->a * p->b;
    return((char *)&res);
}

char *dividir(struct num *p)
{
    static struct num res;
    res.a = p->a / p->b;
    return((char *)&res);
}
```

Asignaciones

Para las siguientes asignaciones utilizar programación RPC nivel intermedio :

1. Escribir un servicio RPC que permita llamar a una función remota que realice el eco (es decir, la función recibirá como parámetro una cadena y el valor devuelto será la misma cadena). Esto se repite por cada solicitud de servicio por parte del cliente hasta que se presiona **<ENTER>** o *Fin de archivo* **<Control+D>**. Al cliente se le debe proporcionar como parámetro el nombre del servidor remoto.

2. Escribir un servicio RPC que permita llamar a una función remota que devuelva la fecha y hora (servicio `daytime`) de una máquina remota (la función tendrá como parámetro el nombre de la máquina y como resultado devolverá la fecha del servidor).

3. Escribir un servicio RPC que, dado un nombre de máquina y una cadena de caracteres que identifiquen a un usuario de esa máquina, permita:

    - Obtener el conjunto de información concerniente a este usuario (la estructura `passwd` predefinida en <pwd.h> y obtenida, por ejemplo, por la función `getwnam`).
    - Obtener la estructura arborescente de los archivos del usuario (la organización general y no el contenido de los archivos).

Referencias Bibliográficas

[1] Jean-Marie Rifflet
"Comunicaciones en UNIX".
Editorial McGraw-Hill, 1992.
Primera Edición.

[2] "Communications Programming Concepts".
http://anguilla.u.arizona.edu/doc_link/en_US/a_doc_lib/aixprggd/progcomc/

[3] Cisco Guides
http://www.cisco.com/univercd/cc/td/doc/product/software/ioss390/ios390rp/

[4] Web Documents
http://webdocs.sequent.com/docs/rpcpat01/

[5] "RPC Programming Documents"
http://webdocs.sequent.com/docs/rpcpat01/book_toc.htm

Guía de Laboratorio # 12

Programación con RPC Nivel Bajo

Introducción

En este nivel tenemos todas las posibilidades con una secuencia de llamada similar pero con mayor dificultad que el nivel intermedio, por lo que no se suele emplear a no ser que se requiera una opción que no se disponga en el nivel intermedio.

Hay dos posibilidades para programar en este nivel: una es hacer directamente la llamada a las funciones que implementan los cabos, y otra es utilizar el compilador `rpcgen` para generarlos, con lo que se reduce el número de llamadas a efectuar, pero complica la programación del cliente.

Elección entre Transporte UDP o TCP

Depende de varios factores, inherentes a ambas formas de transporte.

- Usando UDP no se puede estar seguros con respecto a si se ejecutó o no el procedimiento remoto. Lo que se hace con UDP, en caso de no recibir respuestas, es reenvíos periódicos durante un tiempo, y si no se recibe respuesta, podemos decir que el procedimiento remoto se ejecutó cero o más veces. Si recibimos una respuesta, podemos decir que se ejecutó una o mas veces.

- Usando TCP, cuando se recibe respuesta, se dice que el procedimiento se ejecutó exactamente una vez. En caso contrario, no podemos asegurar que no se ejecutó.

  La longitud del mensaje influye en la elección del protocolo de transporte: si es de más de 8Kbytes se utilizará TCP.

  En aplicaciones pensadas para correr en redes de área local, como por ejemplo NFS, es mejor emplear UDP; va más rápido y no suele haber problemas en el transporte de datos.

Llamadas en el Servidor

La principal diferencia, desde el punto de vista del código en el servidor, es que hay que reescribir todos los procedimientos bajo una única función servidora que será invocada cuando llegue cualquier petición. Cuando se llama, esta función analizará de qué petición se trata y ejecutará el procedimiento correspondiente, devolviendo resultados.

Los pasos que hay que seguir en la función principal del servidor son:

- Crear un descriptor de transporte.
- Registrar solamente la función servidora que agrupa todos los procedimientos en el proceso `portmap`.
- Quedar a la espera de peticiones con `svc_run()`.

Los pasos que hay que seguir en la función servidora son:

- Analizar la petición con un `switch()`.
- Leer los datos con `svc_getargs()`.
- Ejecutar el procedimiento.
- Devolver los resultados con `svc_sendreply()`.
- Regresar a esperar peticiones con `return()`.

Creación de un Descriptor de Transporte

Este paso consiste en crear un socket, enlazarlo y crear una estructura especial en donde se anotarán las características del transporte seleccionado.

La estructura mencionada es del tipo SVCXPRT definida en `<rpc/rpc.h>` y posee, entre otros, a `xp_sock` que tiene el descriptor del socket creado y `xp_port`, que tiene el número de puerto asociado.

Para poder registrar la función servidora, es necesario crear un objeto del tipo SVCXPRT, además de un socket enlazado. Existen dos funciones que permiten crear los objetos de tipo SVCXPRT y a la vez crean y enlazan un socket. Cada función correspondiente a uno de los protocolos de transporte: UDP o TCP.

Para cada una de las funciones, el número del descriptor de socket a crear será uno de los parámetros (el único en el caso del protocolo UDP). Si previamente no se ha realizado la creación del socket, puede utilizarse la constante RPC_ANYSOCK. Las funciones devuelven NULL en caso de fallo, y un puntero a un objeto SVCXPRT si todo va bien. Para utilizar transporte UDP tenemos la función:

```
SVCXPRT *svcudp_create(sockfd)
          int sockfd;
```

El único parámetro es el descriptor del socket, que normalmente es la constante RPC_ANYSOCK. Si queremos que el transporte se haga orientado a conexión, emplearemos la llamada:

```
SVCXPRT * svctcp_create(sockfd, tamaño_envio, tamaño_recepción)
            int               sockfd;
            unsigned int      tamaño_envio, tamaño_recepción;
```

Los parámetros `tamaño_envio` y `tamaño_recepción` indican el tamaño de los buffers utilizados en la emisión y recepción en las operaciones sobre el socket. Un valor nulo corresponde al tamaño por defecto.

Registro de un Servicio y Espera de Peticiones

Se trata de relacionar un número de programa y de versión con un puerto comunicando esta asociación a `portmap`. La función en cuestión es:

```
int svc_register(p_svc, num_programa, num_versión, f_servidora, protocolo)

SVCXPRT             *p_svc;     /*puntero a estructura SVCXPRT*/
unsigned long       num_programa, num_versión;
void                (*f_servidora)();
unsigned long       protocolo;
```

Registra el programa con número `num_programa`, versión `num_version` con el número de puerto `p_svc` del protocolo `protocolo`. Los valores de protocolo pueden ser:

- IPPROTO_UDP
- IPPROTO_TCP
- 0 (no se registra)

La Función Servidora

La función servidora agrupa a todos los procedimientos y será llamada cuando llegue cualquier petición. Los parámetros de la llamada a esta función serán dos punteros: uno apuntando a una estructura que tiene información sobre el procedimiento que hay que ejecutar (estructura `svc_req`) y el otro al descriptor de transporte un objeto SVCXPRT.

La estructura del tipo `svc_req` está definida en `<rpc/rpc.h>`, y el campo que nos interesa es el `rq_proc`, que contiene el número del procedimiento llamado.

Analizando los posibles valores del campo `rq_proc` (con un `switch`) ejecutamos los distintos procedimientos. Sin embargo, hay que considerar 2 casos especiales:

- El procedimiento es 0 (caso NULLPROC), que no devuelve ningún valor, así el cliente sabe que el servidor está esperando peticiones. Después volvemos a la escucha con `return()`.
- El procedimiento encargado de tratar los casos de número de procedimiento erróneo (`case default`), llamando a la función `svcerr_noproc`, y regresando a la escucha con `return()`.

Una vez analizado el procedimiento a ejecutar, lo primero que hay que hacer en el mismo es leer los datos que nos envía el cliente, con la función `svc_getargs()`:

```
int svc_getargs(p_svc, xdr_don, p_don)
      SVCXPRT     *p_svc;
      bool_t      (*xdr_don)();
      char        *p_don;
```

Los datos extraídos del flujo se decodifican por medio de la función `xdr_don` y se ponen en la dirección `p_don`. Devuelve 1 en caso de éxito y 0 en caso contrario.

El siguiente paso es la ejecución del procedimiento con los datos del cliente, y por último enviar los datos de respuesta al mismo con la llamada `svc_sendreply()`:

```
int svc_sendreply(p_svc, xdr_res, p_res)
      SVCXPRT     *p_svc;
      bool_t            (*xdr_res)();
      char        *p_res;
```

Los resultados en la dirección `p_res` se codifican aplicando la función `xdr_res` y se transmiten por el puerto de servicio designado por la estructura apuntada por `p_svc`. El valor devuelto por la función es 1, en caso de éxito, y 0 en caso contrario.

Lo último a incluir en el procedimiento es el retorno a la situación de escucha con `return()`.

Llamadas en el Cliente

En el cliente se hacen dos llamadas en vez de una: la primera para buscar la dirección del servidor y la segunda para ejecutar el procedimiento remoto. Este planteamiento, tiene la ventaja, de que en futuras ejecuciones sólo es necesario realizar la segunda llamada, ya que conocemos la dirección, evitando tráfico innecesario en la red.

Hay dos tipos de llamadas para buscar la dirección del servidor en función del transporte escogido. Las llamadas devuelven un puntero a una estructura tipo `CLIENT` definida en `<rpc/clnt.h>`.

Ambas llamadas crean y asocian un socket e inicializan la estructura `sockaddr_in` con el puerto y la dirección IP del servidor. La función para el transporte UDP es:

```
CLIENT *clntudp_create(dir_máquina, num_programa, num_version, tiempo, sockfd)
       struct sockaddr_in   *dir_máquina;
       unsigned long        num_programa, num_version;
       struct timeval       tiempo;
       int                  *sockfd;
```

La estructura del tipo `svc_req` está definida en `<rpc/rpc.h>`, y el campo que nos interesa es el `rq_proc`, que contiene el número del procedimiento llamado.

Analizando los posibles valores del campo `rq_proc` (con un `switch`) ejecutamos los distintos procedimientos. Sin embargo, hay que considerar 2 casos especiales:

- El procedimiento es 0 (caso NULLPROC), que no devuelve ningún valor, así el cliente sabe que el servidor está esperando peticiones. Despúes volvemos a la escucha con `return()`.
- El procedimiento encargado de tratar los casos de número de procedimiento erróneo (`case default`), llamando a la función `svcerr_noproc`, y regresando a la escucha con `return()`.

Una vez analizado el procedimiento a ejecutar, lo primero que hay que hacer en el mismo es leer los datos que nos envía el cliente, con la función `svc_getargs()`:

```
int svc_getargs(p_svc, xdr_don, p_don)
      SVCXPRT     *p_svc;
      bool_t      (*xdr_don)();
      char        *p_don;
```

Los datos extraídos del flujo se decodifican por medio de la función `xdr_don` y se ponen en la dirección `p_don`. Devuelve 1 en caso de éxito y 0 en caso contrario.

El siguiente paso es la ejecución del procedimiento con los datos del cliente, y por último enviar los datos de respuesta al mismo con la llamada `svc_sendreply()`:

```
int svc_sendreply(p_svc, xdr_res, p_res)
      SVCXPRT     *p_svc;
      bool_t            (*xdr_res)();
      char        *p_res;
```

Los resultados en la dirección `p_res` se codifican aplicando la función `xdr_res` y se transmiten por el puerto de servicio designado por la estructura apuntada por `p_svc`. El valor devuelto por la función es 1, en caso de éxito, y 0 en caso contrario.

Lo último a incluir en el procedimiento es el retorno a la situación de escucha con `return()`.

Llamadas en el Cliente

En el cliente se hacen dos llamadas en vez de una: la primera para buscar la dirección del servidor y la segunda para ejecutar el procedimiento remoto. Este planteamiento, tiene la ventaja, de que en futuras ejecuciones sólo es necesario realizar la segunda llamada, ya que conocemos la dirección, evitando tráfico innecesario en la red.

Hay dos tipos de llamadas para buscar la dirección del servidor en función del transporte escogido. Las llamadas devuelven un puntero a una estructura tipo `CLIENT` definida en `<rpc/clnt.h>`.

Ambas llamadas crean y asocian un socket e inicializan la estructura `sockaddr_in` con el puerto y la dirección IP del servidor. La función para el transporte UDP es:

```
CLIENT *clntudp_create(dir_máquina, num_programa, num_version, tiempo, sockfd)
       struct sockaddr_in   *dir_máquina;
       unsigned long        num_programa, num_version;
       struct timeval       tiempo;
       int                  *sockfd;
```

Con esta función se crea un cliente para el programa num_programa con la versión num_versión en la máquina dir_máquina. El descriptor del socket que se utiliza en la comunicación es sockfd, que se crea eventualmente en la propia llamada. Si *sockfd = RPC_ANYSOCK. tiempo indica el espacio de tiempo al cabo del cual se reenvía la petición en caso de que no haya llegado la respuesta.

El número de puerto del servidor puede ser nulo. inicializándose cuando se produce una llamada mediante la consulta del servicio portmap remoto.

Cuando el transporte elegido sea orientado a conexión, usaremos la función:

```
CLIENT *clnttcp_create(dir_máquina, num_programa, num_versión, sockfd,
                       tamaño_envío, tamaño_recepción)

struct sockaddr_in      *dir_máquina;
unsigned long           num_programa, num_versión;
int                     *sockfd;
unsigned int            tamaño_envío, tamaño_recepción;
```

Si los parámetros tamaño_envío y tamaño_recepción toman el valor NULL. el sistema fija el tamaño por defecto . Caso contrario. toma el valor de estos parámetros para fijar sus tamaños.

Para llamar al procedimiento remoto se emplea la función:

```
enum clnt_stat clnt_call (p_clnt, num_procedimiento, xdr_don, p_don, xdr_res,
                          p_res, total)

CLIENT                  *p_clnt;
unsigned long           num_procedimiento;
struct timeval          total;
char                    *p_don, *p_res;
bool_t                  (*xdr_don)(), (*xdr_res)();
```

Donde total proporciona el tiempo total máximo de espera por un resultado. Para el caso de UDP, en la llamada clntudp_create() se especifica el tiempo entre petición y petición así como el intervalo durante el que se reenviarán las peticiones. Para TCP indica el tiempo máximo en el que se estará reenviando. Si durante el intervalo de tiempo especificado por este parámetro el servidor no responde, se asume la existencia de un fallo, situación que también se produce si el valor devuelto es distinto de cero. En caso de éxito el valor devuelto será RPC_SUCCESS (valor 0).

Por último, cuando el cliente deje de hacer llamadas debe eliminar la estructura CLIENT con la función:

```
clnt_destroy(p_clnt)
        CLIENT *p_clnt;
```

Nivel Bajo con el Compilador rpcgen

Con la ayuda del compilador se simplifica bastante la programación del cliente y el servidor. Se especifican en el lenguaje RPCL las estructuras a emplear en el programa, versión y procedimiento. La especificación se pasa por el compilador rpcgen, que genera los archivos de cabecera, de filtro, cabo cliente, cabo servidor y los esqueletos del programa cliente y servidor. En los cabos se ocultan prácticamente todas las llamadas y pasos vistos en el nivel bajo, por lo que sólo resta editar los procedimientos en el servidor, es decir, personalizar el código esqueleto del servidor. En el cliente solamente hay que averiguar el puerto del servidor y después llamar directamente a cada procedimiento remoto tantas veces y en el orden que se requiera.

Desarrollo

El siguiente ejemplo implementa una aplicación cliente-servidor utilizando programación RPC nivel bajo, cuya función es emular una calculadora distribuida. Al programa cliente se le proporcionará (dentro de su llamada) el nombre del servidor, los datos y la operación a realizar. Las operaciones disponibles serán las cuatro operaciones aritméticas básicas (+, -, *, /) que serán procesadas por el programa servidor. El trabajo del cliente será suministrar los operandos y el operador al programa servidor. La calculadora debe tener la capacidad de procesar una operación aritmética a la vez, hasta que se digite `s` (salir), lo cual hace que termine el proceso cliente.

La ejecución del programa cliente debe tener el aspecto siguiente:

```
> calcu 169.168.0.5
169.168.0.5> 5*50
169.168.0.5> 250
169.168.0.5> 10-50
169.168.0.5> 40
169.168.0.5> s
> cliente terminado.
```

Escribir la especificación de nuestra aplicación en el lenguaje RPCL. Dicha especificación debe contener los nombres de los programas, versiones, procedimientos y estructuras de los datos que van a intercambiar cliente y servidor. Todo esto se introduce en un archivo de texto con el nombre de nuestra aplicación y `extensión x`. Para nuestro caso se llamará `calcu.x`, cuyo contenido es el siguiente :

Contenido del archivo calcu.x en lenguaje RPCL:

```
struct operandos
{
   float a;
   float b;
};

typedef   float resultado;

program CALCU
{
   version CALCU_VERS
   {
      resultado SUMAR         (struct operandos) = 1;
      resultado RESTAR        (struct operandos) = 2;
      resultado MULTIPLICAR   (struct operandos) = 3;
      resultado DIVIDIR       (struct operandos) = 4;
   } = 1;
} = 0x38000000;
```

Se compila el archivo `calcu.x` con el siguiente comando `rpcgen`:

```
# rpcgen -a calcu.x
```

Se generan los archivos con los siguientes nombres:

- **`calcu.h`** Archivo de encabezamiento.
- **`calcu_xdr.c`** Archivo de codificación y decodificación XDR.
- **`calcu_svc.c`** Cabo servidor.
- **`calcu_clnt.c`** Cabo cliente.
- **`calcu_client.c`** Esqueleto de archivo fuente del cliente.
- **`calcu_server.c`** Esqueleto de archivo fuente con los procedimientos a ejecutar en el servidor.

En `calcu_client.c` se realiza la llamada a los procedimientos remotos. Se debe introducir código para conseguir los parámetros y para procesar el resultado. En la llamada a la función `clnt_create()`, podemos seleccionar en el cuarto parámetro si el cliente utilizará protocolo UDP o TCP.

En `calcu_server.c` sólo debemos completar el proceso que se realiza en cada uno de los procedimientos remotos. Se implementan y registran un servidor TCP y otro UDP.

Por último se compilan todos los fichero fuente cabos y filtros de cada parte, para generar los ejecutables siguientes:

- **`cliente`** Archivo ejecutable del cliente.
- **`servidor`** Archivo ejecutable del servidor.

Las órdenes de compilación serán:

```
gcc calcu_xdr.c calcu_svc.c calcu_server.c -o servidor
gcc calcu_xdr.c calcu_clnt.c calcu_client.c -o cliente
```

También se puede utilizar para la compilación el siguiente archivo `make`, el cual generalmente se nombre como `Makefile`:

Contenido del archivo Makefile:

```
PROGS =     cliente servidor

all:  ${PROGS}

cliente:    calcu_client.o calcu_clnt.o calcu_xdr.o
            ${CC} ${CFLAGS} -o $@ calcu_client.o calcu_clnt.o calcu_xdr.o

servidor:   calcu_server.o calcu_svc.o calcu_xdr.o
            ${CC} ${CFLAGS} -o $@ calcu_server.o calcu_svc.o calcu_xdr.o

clean:
            rm -f ${PROGS} ${CLEANFILES}
```

Las líneas de código resaltadas en los archivos `calcu_client.c` y `calcu_server.c`. son las que han sido necesarias editar para que la aplicación funcione según las especificaciones de desempeño de la calculadora distribuida. Como podrá notar, la gran mayoría del código lo genera el compilador `rpcgen`.

A continuación se muestra el código de todos los archivos necesarios para compilar satisfactoriamente la aplicación con los cambios anteriormente mencionados:

Código del archivo calcu.h

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#ifndef _CALCU_H_RPCGEN
#define _CALCU_H_RPCGEN

#include <rpc/rpc.h>
```

```
struct operandos
{
  float a;
  float b;
};
typedef struct operandos operandos;
#ifdef __cplusplus
extern "C" bool_t xdr_operandos(XDR *, operandos*);
#elif __STDC__
extern bool_t xdr_operandos(XDR *, operandos*);
#else /* Old Style C */
bool_t xdr_operandos();
#endif /* Old Style C */

typedef float resultado;
#ifdef __cplusplus
extern "C" bool_t xdr_resultado(XDR *, resultado*);
#elif __STDC__
extern bool_t xdr_resultado(XDR *, resultado*);
#else /* Old Style C */
bool_t xdr_resultado();
#endif /* Old Style C */

#define CALCU ((u_long)0x38000000)
#define CALCU_VERS ((u_long)1)

#ifdef __cplusplus
#define SUMAR ((u_long)1)
extern "C" resultado * sumar_1(struct operandos *, CLIENT *);
extern "C" resultado * sumar_1_svc(struct operandos *, struct svc_req *);
#define RESTAR ((u_long)2)
extern "C" resultado * restar_1(struct operandos *, CLIENT *);
extern "C" resultado * restar_1_svc(struct operandos *, struct svc_req *);
#define MULTIPLICAR ((u_long)3)
extern "C" resultado * multiplicar_1(struct operandos *, CLIENT *);
extern "C" resultado * multiplicar_1_svc(struct operandos *, struct svc_req *);
#define DIVIDIR ((u_long)4)
extern "C" resultado * dividir_1(struct operandos *, CLIENT *);
extern "C" resultado * dividir_1_svc(struct operandos *, struct svc_req *);

#elif __STDC__
#define SUMAR ((u_long)1)
extern resultado * sumar_1(struct operandos *, CLIENT *);
extern resultado * sumar_1_svc(struct operandos *, struct svc_req *);
#define RESTAR ((u_long)2)
extern resultado * restar_1(struct operandos *, CLIENT *);
extern resultado * restar_1_svc(struct operandos *, struct svc_req *);
#define MULTIPLICAR ((u_long)3)
extern resultado * multiplicar_1(struct operandos *, CLIENT *);
extern resultado * multiplicar_1_svc(struct operandos *, struct svc_req *);
#define DIVIDIR ((u_long)4)
extern resultado * dividir_1(struct operandos *, CLIENT *);
extern resultado * dividir_1_svc(struct operandos *, struct svc_req *);

#else /* Old Style C */
#define SUMAR ((u_long)1)
extern resultado * sumar_1();
extern resultado * sumar_1_svc();
#define RESTAR ((u_long)2)
extern resultado * restar_1();
extern resultado * restar_1_svc();
```

```
#define MULTIPLICAR ((u_long)3)
extern resultado * multiplicar_1();
extern resultado * multiplicar_1_svc();
#define DIVIDIR ((u_long)4)
extern resultado * dividir_1();
extern resultado * dividir_1_svc();
#endif /* Old Style C */

#endif /* !_CALCU_H_RPCGEN */
```

Código del archivo calcu_xdr.c

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include <rpc/types.h>
#include <rpc/xdr.h>

#include "calcu.h"

bool_t
xdr_operandos(XDR *xdrs, operandos *objp)
{
  register long *buf;

  if (!xdr_float(xdrs, &objp->a))
  {
    return (FALSE);
  }
  if (!xdr_float(xdrs, &objp->b))
  {
    return (FALSE);
  }
  return (TRUE);
}

bool_t
xdr_resultado(XDR *xdrs, resultado *objp)
{
  register long *buf;

  if (!xdr_float(xdrs, objp))
  {
    return (FALSE);
  }
  return (TRUE);
}
```

Código del archivo cabo calcu_clnt.c

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */
```

```
#include <memory.h> /* for memset */
#include "calcu.h"

/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = { 25, 0 };

resultado *sumar_1(struct operandos *argp, CLIENT *clnt)
{
  static resultado clnt_res;

  memset((char *)&clnt_res, 0, sizeof(clnt_res));
  if (clnt_call(clnt, SUMAR, xdr_operandos, argp, xdr_resultado,
      &clnt_res, TIMEOUT) != RPC_SUCCESS)
  {
    return (NULL);
  }
  return (&clnt_res);
}

resultado *restar_1(struct operandos *argp, CLIENT *clnt)
{
  static resultado clnt_res;

  memset((char *)&clnt_res, 0, sizeof(clnt_res));
  if (clnt_call(clnt, RESTAR, xdr_operandos, argp, xdr_resultado,
      &clnt_res, TIMEOUT) != RPC_SUCCESS)
  {
    return (NULL);
  }
  return (&clnt_res);
}

resultado *multiplicar_1(struct operandos *argp, CLIENT *clnt)
{
  static resultado clnt_res;

  memset((char *)&clnt_res, 0, sizeof(clnt_res));
  if (clnt_call(clnt, MULTIPLICAR, xdr_operandos, argp, xdr_resultado,
      &clnt_res, TIMEOUT) != RPC_SUCCESS)
  {
    return (NULL);
  }
  return (&clnt_res);
}

resultado *dividir_1(struct operandos *argp, CLIENT *clnt)
{
  static resultado clnt_res;

  memset((char *)&clnt_res, 0, sizeof(clnt_res));
  if (clnt_call(clnt, DIVIDIR, xdr_operandos, argp, xdr_resultado,
      &clnt_res, TIMEOUT) != RPC_SUCCESS)
  {
    return (NULL);
  }
  return (&clnt_res);
}
```

Código del archivo cabo calcu_svc.c

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include "calcu.h"
#include <stdio.h>
#include <stdlib.h>/* getenv, exit */
#include <rpc/pmap_clnt.h> /* for pmap_unset */
#include <string.h> /* strcmp */
#include <memory.h>
#include <sys/socket.h>
#include <netinet/in.h>
#ifdef __STDC__
#define SIG_PF void(*)(int)
#endif

static void
calcu_1(struct svc_req *rqstp, register SVCXPRT *transp)
{
  union
  {
    struct operandos sumar_1_arg;
    struct operandos restar_1_arg;
    struct operandos multiplicar_1_arg;
    struct operandos dividir_1_arg;
  } argument;
  char *result;
  xdrproc_t xdr_argument, xdr_result;
  char *(*local)(char *, struct svc_req *);

  switch (rqstp->rq_proc)
  {
    case NULLPROC:
          (void) svc_sendreply(transp, (xdrproc_t) xdr_void, (char *)NULL);
          return;

    case SUMAR:
          xdr_argument = (xdrproc_t) xdr_operandos;
          xdr_result = (xdrproc_t) xdr_resultado;
          local = (char *(*)(char *, struct svc_req *)) sumar_1_svc;
          break;

    case RESTAR:
          xdr_argument = (xdrproc_t) xdr_operandos;
          xdr_result = (xdrproc_t) xdr_resultado;
          local = (char *(*)(char *, struct svc_req *)) restar_1_svc;
          break;

    case MULTIPLICAR:
          xdr_argument = (xdrproc_t) xdr_operandos;
          xdr_result = (xdrproc_t) xdr_resultado;
          local = (char *(*)(char *, struct svc_req *)) multiplicar_1_svc;
          break;

    case DIVIDIR:
          xdr_argument = (xdrproc_t) xdr_operandos;
          xdr_result = (xdrproc_t) xdr_resultado;
          local = (char *(*)(char *, struct svc_req *)) dividir_1_svc;
```

```
            break;

        default:
            svcerr_noproc(transp);
            return;
    }

    (void) memset((char *)&argument, 0, sizeof (argument));
    if (!svc_getargs(transp, xdr_argument, (caddr_t) &argument))
    {
        svcerr_decode(transp);
        return;
    }
    result = (*local)((char *)&argument, rqstp);
    if (result != NULL && !svc_sendreply(transp, xdr_result, result))
    {
        svcerr_systemerr(transp);
    }
    if (!svc_freeargs(transp, xdr_argument, (caddr_t) &argument))
    {
        fprintf(stderr, "unable to free arguments");
        exit(1);
    }
    return;
}

int main(int argc, char **argv)
{
    register SVCXPRT *transp;

    (void) pmap_unset(CALCU, CALCU_VERS);

    transp = svcudp_create(RPC_ANYSOCK);
    if (transp == NULL)
    {
        fprintf(stderr, "cannot create udp service.");
        exit(1);
    }
    if (!svc_register(transp, CALCU, CALCU_VERS, calcu_1, IPPROTO_UDP))
    {
        fprintf(stderr, "unable to register (CALCU, CALCU_VERS, udp).");
        exit(1);
    }

    transp = svctcp_create(RPC_ANYSOCK, 0, 0);
    if (transp == NULL)
    {
        fprintf(stderr, "cannot create tcp service.");
        exit(1);
    }
    if (!svc_register(transp, CALCU, CALCU_VERS, calcu_1, IPPROTO_TCP))
    {
        fprintf(stderr, "unable to register (CALCU, CALCU_VERS, tcp).");
        exit(1);
    }

    svc_run();
    fprintf(stderr, "svc_run returned");
    exit(1);
    /* NOTREACHED */
}
```

Código del archivo cliente calcu_client.c (modificado)

```
/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "calcu.h"

void
calcu_1( char* host )
{
      CLIENT *clnt;

      struct      operandos valor;
      resultado   *result;
      char        cadena[sizeof(struct operandos)];
      char        operador;

      clnt = clnt_create(host, CALCU, CALCU_VERS, "udp");
      if (clnt == NULL) {
            clnt_pcreateerror(host);
            exit(1);
      }

      printf("\n%s> ", host);
      gets(cadena);
      sscanf(cadena, "%f %c %f", &valor.a, &operador, &valor.b);

      if(!strcmp(cadena, "s")) exit(1);

      switch(operador)
      {
        case '+': result = sumar_1(&valor, clnt);
              if (result == NULL) clnt_perror(clnt, "call failed:");
              break;
        case '-': result = restar_1(&valor, clnt);
              if (result == NULL) clnt_perror(clnt, "call failed:");
              break;
        case '*': result = multiplicar_1(&valor, clnt);
             if (result == NULL) clnt_perror(clnt, "call failed:");
              break;
        case '/': result = dividir_1(&valor, clnt);
              if (result == NULL) clnt_perror(clnt, "call failed:");
              break;
      }
      printf("%s> %f", host, *result);

      clnt_destroy( clnt );
}

main(int argc, char* argv[])
{
      char *host;

      system("clear");
```

```
        if(argc != 2)
        {
          printf("\nUso: calcu nombre_host\n");
          printf("Ejemplo: calcu localhost\n");
          exit(2);
        }
        printf( "  ****************CALCULADORA DISTRIBUIDA****************"
                "\nEste programa realiza con dos números una de"
                " las operaciones" siguientes:\n"
                "\n + Suma\n - Resta\n * Multiplicacion\n / Division\n"
                " s Salir\n"
                "\nPrimer operando, operador, segundo operando:"
                "\nEjemplo: 10 * 30\n"
              );

        host = argv[1];

        for(;;) calcu_1(host);

}
```

Código del archivo servidor calcu_server.c

```
/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "calcu.h"

resultado *sumar_1_svc(struct operandos *argp, struct svc_req *rqstp)
{

        static resultado result;
        result = argp->a + argp->b;
        return(&result);
}

resultado *restar_1_svc(struct operandos *argp, struct svc_req *rqstp)
{

        static resultado result;
        result = argp->a - argp->b;
        return(&result);
}

resultado *multiplicar_1_svc(struct operandos *argp, struct svc_req *rqstp)
{

        static resultado result;
        result = argp->a * argp->b;
        return(&result);
}

resultado *dividir_1_svc(struct operandos *argp, struct svc_req *rqstp)
{
```

```
        static resultado result;
        result = argp->a / argp->b;
        return(&result);
}
```

Asignaciones

Para las siguientes asignaciones utilizar programación RPC nivel bajo:

1. Escribir un servicio RPC que permita llamar a una función remota que realice el eco (es decir, la función recibirá como parámetro una cadena y como resultado la misma cadena). Esto se repite por cada solicitud de servicio por parte del cliente hasta que se presiona <ENTER> o *Fin de archivo* <Control+D>. Al cliente se le debe proporcionar como parámetro el nombre del host remoto y la cadena a enviar. Utilizar protocolo UDP.

2) Escribir un servicio RPC que permita llamar a una función remota que devuelva la fecha y hora (servicio `daytime`) de una máquina remota (la función tendrá como parámetro el nombre de la máquina y como resultado la fecha en esta máquina). Utilizar protocolo UDP.

3) Escribir un servicio RPC que dado un nombre de máquina y una cadena de caracteres que identifiquen a un usuario de esa máquina, permita:

   - Obtener el conjunto de información concerniente a este usuario (estructura passwd predefinida en <pwd.h>) y obtenida, por ejemplo, por la función `getwnam`).
   - Obtener la estructura arborescente de los archivos del usuario (la organización general y no el contenido de los archivos).

   Utilizar protocolo UDP.

4) Escribir un servicio RPC que permita llamar a una función remota que reciba como parámetro un comando (y sus parámetros respectivos) dando como resultado la salida de la ejecución del comando en la máquina que presta el servicio.
   La respuesta de la ejecución de dicho comando debe ser devuelta al programa cliente para que sea mostrada en la pantalla del mismo. Se debe utilizar protocolo TCP para la implementación de la aplicación. Se recomienda utilizar la función `dup2()` para redireccionar tanto la entrada, la salida y el error estándar a un búffer que será utilizado como el valor devuelto por la función que presta el servicio en el servidor
   El cliente debe tener la capacidad de aceptar un comando (el cual será pasado como parámetro para la función que presta el servicio RPC) o el carácter `s`, el cual hace que el proceso cliente termine. Al programa cliente se le proporcionará el nombre del servidor.

   La ejecución del programa cliente debe tener el aspecto siguiente:

```
> remoto 169.168.0.5
169.168.0.5> date
169.168.0.5> Thu May 22 19:28:11 2000
169.168.0.5> s
remoto terminado.
```

Referencias Bibliográficas

[1] Jean-Marie Rifflet
"Comunicaciones en UNIX".
Editorial McGraw-Hill, 1992.
Primera Edición.

[2] "Communications Programming Concepts."
http://anguilla.u.arizona.edu/doc_link/en_US/a_doc_lib/aixprggd/progcomc/

[3] Cisco Guides
http://www.cisco.com/univercd/cc/td/doc/product/software/ios390/ios390rp/

[4] Web Documents
http://webdocs.sequent.com/docs/rpcpaa01

[5] RPC Programming Documents
http://webdocs.sequent.com/docs/rpcpaa01_book_toc.htm

ANEXO B: PRINCIPALES PROTOCOLOS DE CAPA 3 Y 4 (MODELO OSI)

Principales Protocolos de Capa 3 y 4 (Modelo OSI)

El Protocolo IP (Internet Protocol)

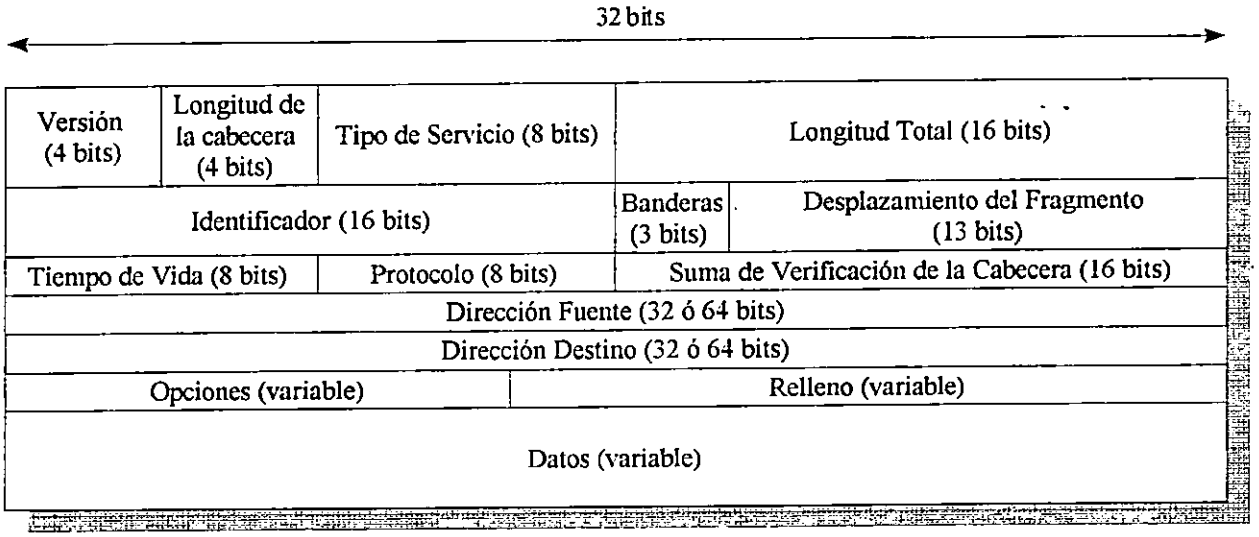
IP (Internet Protocol) es un protocolo de capa de red (nivel 3) no orientado a la conexión, por lo que permite el envío de datos entre dos ordenadores sin establecer una llamada previamente. Esto hace que IP, por sí solo, sea un protocolo de poca fiabilidad ya que no tiene procedimientos de recuperación de errores ni control de flujo. La mayoría de estos y otros problemas se dejan a cargo de protocolos de nivel superior.

La versión mas popular de este protocolo es el IPV4 (Protocolo de Internet versión 4), el cual es usado prácticamente en todo el mundo, ya que es el protocolo de direccionamiento usado por Internet. Sin embargo, este protocolo puede definir un número muy pequeño de direcciones (2^{32} direcciones)comparado con la gran cantidad de dispositivos conectados a la red mundial (internet). Este problema ha sido atenuado mediante técnicas como el reuso de direcciones, pero aun así esto no es una solución definitiva.

El vista de lo anterior, se creo la versión 6 de IP (IPV6). Se espera que en un futuro no muy lejano desplace al IPV4, ya que cuenta con una mucho mayor capacidad de direccionamiento (2^{64} direcciones).

El Datagrama IP

La cabecera del datagrama IP se compone de los siguientes campos:

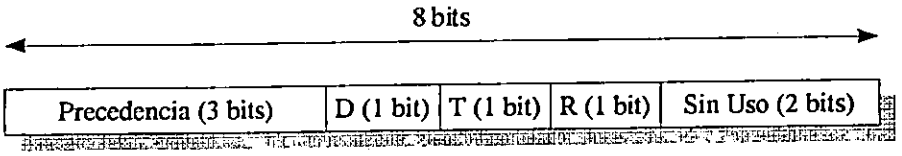


Formato de cabecera para el Protocolo IP

Versión: Este campo especifica la versión de IP que se usa. Esto se hace con el fin de que los datagramas IP de versiones posteriores tengan compatibilidad con aquellos creados por versiones previas. La versión de IP más usada actualmente es la 4.

Longitud de Cabecera: Este campo (HLEN) contiene 4 bits que definen la longitud de la cabecera del datagrama, expresada en palabras de 32 bits (1 bit = 4 bytes u octetos). Generalmente la longitud de la cabecera es 20 bytes (IPV4) o 40 bytes (IPV6), por lo que este campo suele tener un valor de 5 ó 10.

Tipo de Servicio: El tipo de servicio (TOS) se puede usar para identificar varias funciones de calidad de servicio (QoS) de Internet. Este campo esta compuesto de la siguiente forma:



*Formato del campo **Tipo de Servicio** de para la cabecera del protocolo IP*

Donde:

Precedencia: Indican el grado de prioridad de un datagrama. Existen 8 grados de precedencia, que se muestran a continuación según su orden de importancia:

Código	*Nivel de Precedencia*
000	Rutina
001	Prioridad
002	Inmediato
003	Flash
004	Flash Override
005	Crítico
006	Control Interred
007	Control de Red

D: Retardo (Delay). Cuando este bit se activa (D=1), el paquete solicita retardo.

T: Desempeño (Throughput). Cuando este bit es activado, el paquete busca la ruta menos congestionada.

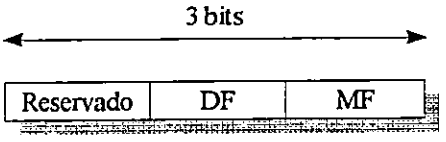
R: Confiabilidad (Reliability). Cuando se da una falla que implica el descarte de paquetes. aquellos que tengan este bit activo tendrán prioridad para ser salvados, descartándose primero los paquetes que tengan este bit desactivado.

Los dos siguientes bits no tienen un uso definido y por tanto no son utilizados por ahora.

Longitud Total: Especifica la longitud total del datagrama IP. Se mide en bytes e incluye tanto cabecera como datos. Para conocer el tamaño del campo de datos, IP resta a este campo el campo **Longitud de Cabecera**. Dado que este campo posee 16 bits. la longitud máxima para un datagrama IP se limita a 65535 octetos (2^{16})

Identificador: Se usa para identificar de manera única todos los fragmentos de un mismo datagrama (A veces, al viajar, el datagrama IP necesita ser fragmentado en unidades más pequeñas para, por ejemplo, viajar sobre una red X.25 donde el tamaño máximo del campo de información es de 4096 octetos). Este campo se utiliza en combinación del campo de **Dirección Fuente** para identificar el fragmento.

Banderas: Este campo de 3 bits está compuesto de la siguiente forma:



*Bits que componen el campo de **Banderas**.*

El primer bit se encuentra reservado para uso futuro. usualmente se le da valor de cero (0).

DF: No fragmentar (Don't Fragment). Cuando este bit es activado (DF=1), el datagrama IP no puede ser fragmentado.

MF: Más Fragmentos (More Fragments). Cuando este bit se encuentra inactivo (MF=0) indica que el datagrama es un fragmento. Cuando MF=1, el datagrama es el último fragmento de otro datagrama de mayor tamaño.

Desplazamiento del Fragmento: Indica la posición relativa del fragmento dentro del datagrama original, es decir, dentro del grupo de datagramas que tienen un mismo identificador. Se mide en unidades de 8 octetos. Si el último fragmento no tiene un número de bytes múltiplo de ocho, se rellena con ceros hasta llegar a un número de bytes que sea múltiplo de ocho. En vista de que gracias a los campos de ***Longitud de Cabecera*** y ***Longitud Total*** podemos saber la longitud del área de datos, podemos saber exactamente dónde termina la información útil y dónde comienza el relleno.
Debe recordarse que cuando un datagrama IP es fragmentado, toda la cabecera es copiada en cada uno de los fragmentos.

Tiempo de Vida: El Tiempo de Vida (TTL) se usa para medir el tiempo que un datagrama lleva en la red. Cuando un dispositivo de la red capaz de observar este campo detecta que su valor ha llegado a cero (0), el paquete es descartado. Dichos dispositivos decrementan el valor de este campo en todos los datagramas que procesan. En realidad, el campo TTL mide el número de saltos que un datagrama hace en la red. Algunas veces se usa un contador de tiempo en este campo, el cual decrementa su valor usualmente cada segundo. El valor inicial de este campo es configurable.

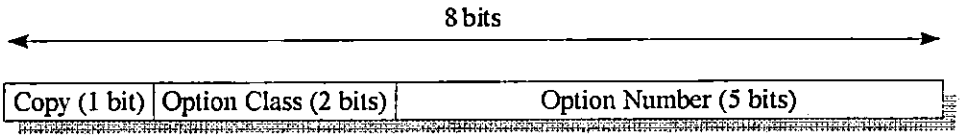
Protocolo: Este campo se utiliza para identificar el tipo de protocolo de capa superior (Capa 4, para este caso) que recibirá el datagrama en el receptor. Actualmente existe un sistema de numeración que identifica a los protocolos de nivel superior más usados (Por ejemplo, el código asignado a TCP es 6).

Suma de Verificación de la Cabecera: Se utiliza para la detección de errores en la cabecera. No se realizan comprobaciones en el área de datos del usuario.

Dirección Fuente: Es la dirección del transmisor u originador del datagrama. Este campo no se modifica en toda la vida del mismo y su longitud puede ser de 32 bits (IPV4) o 64 bits (IPV6).

Dirección Destino: Es la dirección del receptor o destinatario del datagrama. Este campo no se modifica en toda la vida del mismo y su longitud puede ser de 32 bits (IPV4) o 64 bits (IPV6).

Opciones: Este campo se emplea para identificar varios servicios adicionales y varía según la opción. No se usa en todos los datagramas. La mayoría de veces, este campo se usa para gestión de red y diagnóstico. Cada opción consiste de un octeto de código de opción y un conjunto de datos para cada opción. El octeto correspondiente al código de opción se divide en 3 campos que son:



*Bits que componen el campo de **Opciones**..*

Copy: Cuando este bit se activa, la opción es copiada a todos los fragmentos que componen un mismo datagrama. Cuando este bit vale cero, la opción sólo es copiada en el primer fragmento.

Option Class/Option Number: Especifican juntos la clase general de opción y una clase específica de la misma. Sus posibles combinaciones válidas se definen en la tabla siguiente:

Option Class	*Option Number*	*Descripción*
0	0	Fin de lista de opciones. Se usa si las opciones no terminan al final del encabezado.
0	1	No operación. Se usa para alinear octetos en una lista de opciones (como relleno)
0	2	Seguridad y restricciones de manejo (aplicaciones militares)
0	3	Ruteo no estricto de fuente. Se usa para enrutar un datagrama a través de una trayectoria específica de forma preferencial.
0	7	Registro de ruta. Se usa para registrar el trayecto seguido por un datagrama
0	8	Identificador de flujo. (Obsoleto)
0	9	Ruteo estricto de fuente. Se usa para establecer la ruta de un datagrama sobre un trayecto específico de forma estricta.
2	4	Sello de tiempo de Internet. Se usa para registrar sellos de hora a lo largo de una ruta.

El Protocolo ICMP (Internet Control Message Protocol)

Ya que el protocolo de Internet presenta un esquema no confiable y sin conexión para la entrega de datagramas, la coordinación entre los dispositivos que llevan dichos datagramas a su destino final (enrutadores) es prácticamente inexistente, por lo que el sistema trabaja bien solamente si todas las partes involucradas lo hacen, condición que no siempre se da. Para permitir que los dispositivos de comunicación (principalmente los enrutadores) en una red IP reporten errores o proporcionen información acerca de situaciones inesperadas, los diseñadores de IP agregaron un mecanismo de mensajes de propósito especial conocido como *Protocolo de Mensajes de Control de Internet* o ICMP. Este protocolo se considera parte obligatoria de IP y actualmente se incluye en todas las implementaciones del mismo.

Al igual que el resto del tráfico, los mensajes ICMP viajan por una red dentro del campo de datos de un datagrama IP, pero su destino no es un programa de aplicación o un usuario en la máquina destino, sino el software de Protocolo Internet en dicha máquina. En resúmen, el ICMP fué hecho pensando en los enrutadores, de modo que a través de dicho protocolo puedan enviar mensajes de error o control hacia otros enrutadores o anfitriones. ICMP proporciona comunicación entre el software IP en una máquina y el mismo software en otra.

Debe resaltarse que ICMP reporta errores, no los corrige. ICMP reporta los problemas a la fuente original y no a los dispositivos intermedios.

Es importante hacer notar que el ICMP necesita de 2 niveles de encapsulamiento: El mensaje ICMP se encapsula dentro de un datagrama IP, que a su vez se encapsula dentro de alguna trama de nivel 2 (como HDLC).

Por otro lado, los mensajes ICMP que reportan errores siempre regresan los primeros 64 bits de datos del datagrama causante del problema, esto para permitir que el receptor pueda determinar qué protocolos de nivel superior son responsables del datagrama. Es por esto que los protocolos de nivel superior están diseñados para codificar su información más importante en los primeros 64 bits.

Formato de Mensajes ICMP

Aunque cada mensaje ICMP tiene su propio formato, todos comienzan con los mismos 3 campos: Tipo, código y Suma de Verificación (CRC).

Tipo: Este campo de 8 bits define el significado del mensaje, así como su formato. Los códigos definidos para este campo son los siguientes:

Tipo	*Tipo de mensaje ICMP*
0	Respuesta de Eco
3	Destino Inaccesible
4	Disminución de Tasa al Origen
5	Redireccionar
8	Solicitud de Eco
11	Tiempo extendido para un datagrama
12	Problema de parámetros en un datagrama
13	Solicitud de marca de tiempo (timestamp)
14	Respuesta de marca de tiempo (timestamp)
15	Solicitud de información (obsoleto)
16	Respuesta de información (obsoleto)
17	Solicitud de máscara de dirección
18	Respuesta de máscara de dirección

Código: Este campo de 8 bits contiene información que complementa al campo de tipo. Su significado depende del tipo de mensaje ICMP.

Suma de Verificación: Este campo de 16 bits utiliza el mismo algoritmo de verificación que IP, pero dicha verificación se realiza solamente sobre el mensaje ICMP. Al igual que IP, esta campo se usa para buscar errores.

Formato de Mensajes de Solicitudes de Eco y Respuesta

La figura siguiente muestra el formato de mensajes de solicitud de eco y respuesta.

Tipo (8 bits)	Código (8 bits)	Suma de Verificación (16 bits)
Identificador (16 bits)		Número de Secuencia (16 bits)
Datos Opcionales (Variable)		

Formato de mensaje ICMP para solicitud de eco o respuesta.

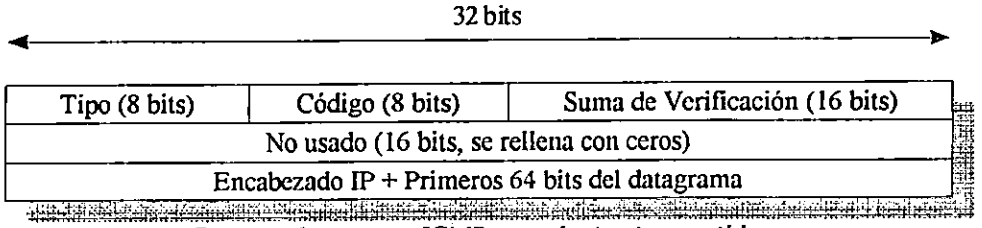
Para este tipo de mensajes, el campo TIPO únicamente puede tener los valores de 8 (Solicitud de eco) o 0 (Respuesta de eco). El campo CODIGO no significa nada en este tipo de mensajes, por lo que se rellena con ceros (0). Los campos de identificador y número de secuencia son usados para responder a las solicitudes, su función es similar a los campos de identificador y desplazamiento de fragmento de IP.

Los datos opcionales es un campo de longitud variable que contiene los datos que serán devueltos al transmisor.

Este tipo de mensajes de solicitud de eco y respuesta suelen usarse para saber si un destino es alcanzable y responde

Reporte de Destinos No Accesibles

Cuando un enrutador no puede direccionar o entregar un datagrama IP, envía un mensaje de *"destino no accesible"* a la fuente original, usando el formato siguiente.



Formato de mensaje ICMP para destino inaccesible.

Pare este caso, el campo TIPO vale 3 y el campo de CODIGO representa un número entero (entre 0 y 12) que describe con mayor detalle el problema. Los siguientes 16 bits no son usados, por lo que se rellenan con ceros.

Los posibles valores del campo CODIGO son:

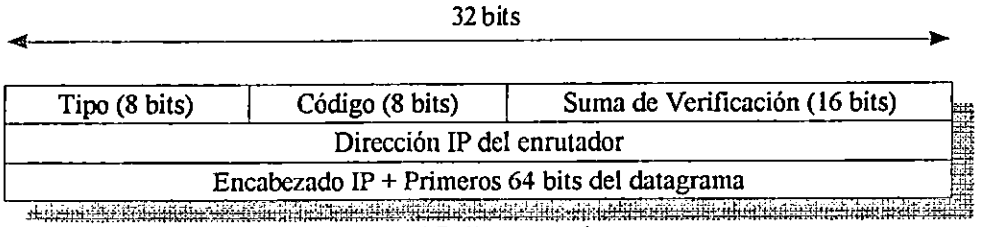
Código	*Significado*
0	Red inaccesible
1	Anfitrión (host) inaccesible
2	Protocolo inaccesible
3	Puerto inaccesible
4	Se necesita fragmentación
5	Falla en ruta de origen
6	Red de destino desconocida
7	Anfitrión de destino desconocido
8	Anfitrión de origen aislado
9	Comunicación con red destino prohibida por el administrador
10	Comunicación con anfitrión destino prohibida por el administrador
11	Red inaccesible por el tipo de servicio
12	Anfitrión inaccesible por el tipo de servicio

Disminución de Tasa al Origen

Este tipo de mensajes son usados por ICMP para reportar congestionamiento a la fuente. Este mensaje es a su vez una solicitud para que la fuente reduzca la tasa de transmisión de datagramas. Los enrutadores suelen enviar un mensaje de este tipo por cada datagrama que descartan. El formato de estos mensajes es igual al usado para reportar destinos no accesibles, con la variante de que el campo TIPO vale 4 y el campo CODIGO carece de significado, por lo que se rellena con ceros.

Solicitud de Redirección

En una red, los enrutadores necesitan de vez en cuando modificar su información de ruteo para mantener actualizadas sus rutas o bien para hacer cambios temporales en caso de que la ruta habitual se congestione o falle. El formato de este tipo de mensaje es el siguiente:



Formato de mensaje ICMP para redireccionamiento

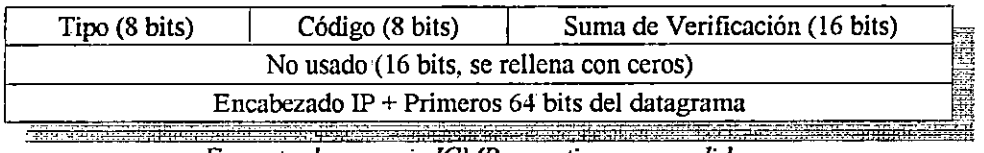
En este mensaje, el campo TIPO vale 5. Este mensaje incluye un campo de 32 bits que indica la dirección IP del enrutador por el que se redireccionarán los mensajes. El campo de CODIGO puede tomar los siguientes valores:

Código	*Significado*
0	Redireccionar datagramas para la red (obsoleto)
1	Redireccionar datagramas para el anfitrión
2	Redireccionar datagramas para el tipo de servicio y la red
3	Redireccionar datagramas para el tipo de servicio y el anfitrión

Como regla general, los enrutadores envían solicitudes de redireccionamiento solamente a los anfitriones y no a otros enrutadores.

Detección de Rutas Circulares o Muy Largas

Dado que los enrutadores calculan un salto al siguiente enrutador usando tablas locales, estas pueden producir errores ya sea creando una ruta que da vueltas en círculo o una que sea excesivamente larga. Para evitar que datagramas den vueltas en círculo indefinidamente o que recorran rutas demasiado prolongadas, consumiendo recursos de la red, cada datagrama contiene un *contador de tiempo de vida*, conocido también como *contador de saltos*. Un enrutador disminuye este contador siempre que procesa un datagrama y lo descarta cuando dicho datagrama llega a cero. Siempre que se descartan datagramas porque este contador llega a cero o porque ocurre una expiración de tiempo a la espera de fragmentos en un datagrama, envía un mensaje ICMP de *tiempo excedido* a la fuente, el cual posee el formato siguiente:



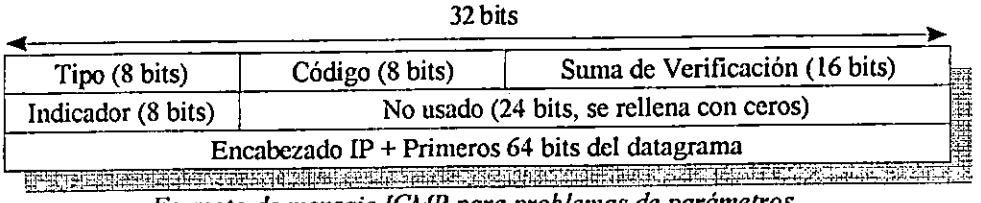
Formato de mensaje ICMP para tiempo excedido.

El campo TIPO vale 11, mientras que el campo CODIGO puede tomar uno de los siguientes valores:

Código	*Significado*
0	Se ha excedido el conteo de tiempo de vida (TTL: time to live)
1	Se ha excedido el tiempo para el reensamblado de fragmentos

Problemas de Parámetros (Otros Problemas)

Cuando un enrutador o anfitrión encuentran un problema no contemplado en laos casos anteriores (como un datagrama con encabezado incorrecto) envía un mensaje de *problema de parámetros* a la fuente. Este tipo se mensaje s se envían solamente cuando el problema es tan severo que se hace necesario descartar el datagrama. Su formato es:

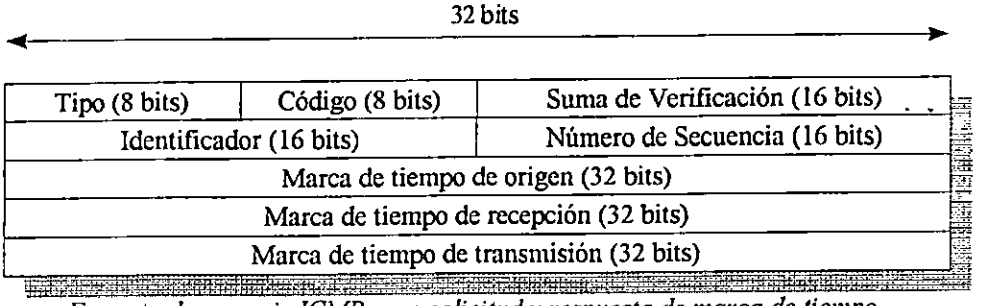


Formato de mensaje ICMP para problemas de parámetros.

El campo TIPO vale 12, mientras que el campo CODIGO puede valer cero (0) o uno (1: falta opción requerida). El campo INDICADOR se usa para identificar el octeto del datagrama que causó el problema. Solamente tiene significado si CODIGO vale cero (0).

Sincronización y Estimación de Tiempo de Tránsito

A pesar de que las máquinas en una red se comunican, cada una opera de forma independiente, manteniendo su propia noción de hora actual. Una de las formas más sencillas para obtener la hora de otra máquina es el envío de mensajes de solicitud y respuesta de marca de tiempo. El formato para este tipo de mensajes es el siguiente:



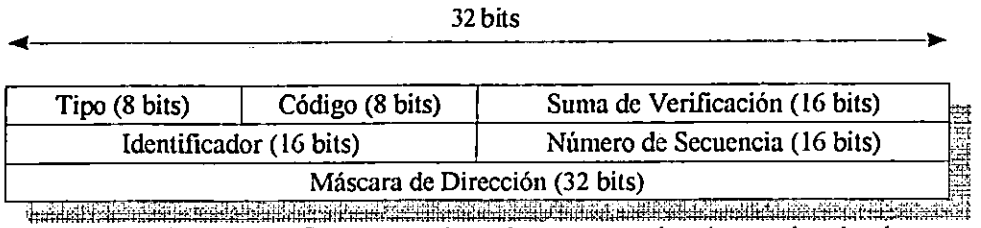
Formato de mensaje ICMP para solicitud y respuesta de marca de tiempo.

El campo TIPO puede valer 13 (solicitud de marca de tiempo) o 14 (respuesta de marca de tiempo). El campo de CODIGO carece de significado y se rellena con ceros. Por otro lado, los campos IDENTIFICADOR y NUMERO DE SECUENCIA son usado por la fuente para asociar las solicitudes con las respuestas. Los campos siguientes especifican la hora en milisegundos desde medianoche según el *tiempo universal* o *tiempo del Meridiano de Greenwich*. Estos campos son:

Marca de Tiempo de Origen: Es llenado por al fuente original justo antes de transmitir el paquete.
Marca de Tiempo de Recepción: Se llena en el receptor inmediatamente después de recibir un mensaje de solicitud de marca de tiempo.
Marca de Tiempo de Transmisión: Se llena en el receptor justo antes de transmitir la respuesta.

Obtención de Máscara de SubRed

Con este tipo de mensajes, se puede conocer qué bits corresponden a una red física y qué bits a los identificadores del anfitrión, es decir, se puede saber la máscara de subred con la que trabaja una red local. El formato para un mensaje de solicitud o respuesta de máscara de subred es el siguiente:



Formato de mensaje ICMP para solicitud y respuesta de máscara de subred.

El campo TIPO puede valer 17 (solicitud de máscara de subred) o 18 (respuesta de máscara de subred). El campo CODIGO carece de significado y se rellena con ceros. Los campos IDENTIFICADOR y NUMERO DE SECUENCIA permiten que una máquina asocie las solicitudes con las respuestas, mientras que la MASCARA DE DIRECCION se usa para indicar la máscara de dirección.

Este mensaje suele ser transmitido por difusión (broadcast), ya que no se sabe cuál enrutador responderá.

El Protocolo TCP (Transfer Control Protocol)

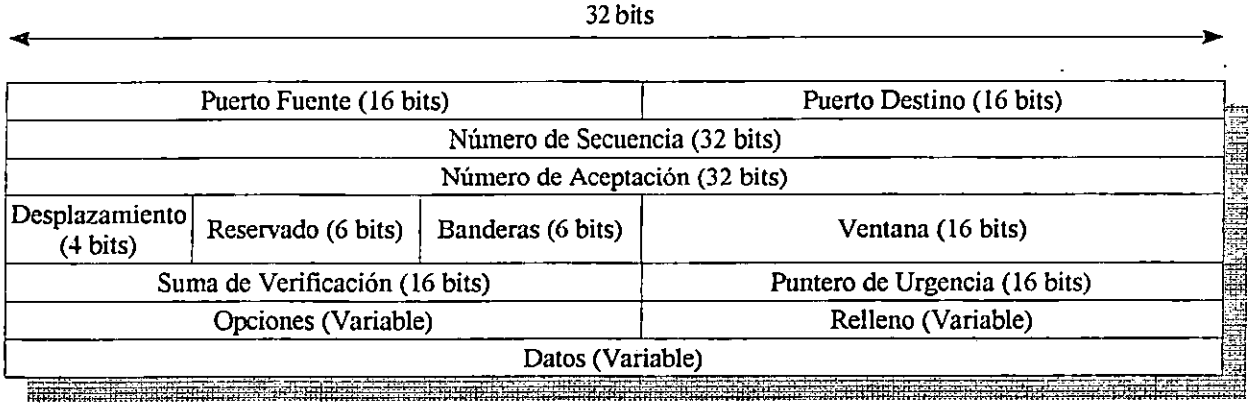
TCP es un protocolo del nivel de transporte (Capa 4). TCP está diseñado para residir en computadores o máquinas que se ocupan de conservar la integridad de la transferencia de datos entre extremos. Lo más común es que TCP resida en las computadores del usuario. TCP es quien se encarga de las tareas de fiabilidad, control de flujo, secuenciamiento, aperturas y cierres.

TCP es un protocolo orientado a la conexión, por lo que es responsable de la transferencia confiable de todos los bytes que recibe del nivel superior. Esto lo hace por medio de números de secuencia y aceptaciones/rechazos

La unidad de transferencia en el protocolo TCP se conoce como segmento.

El Segmento TCP

La figura mostrada a continuación muestra el formato del segmento TCP, el cual se compone cabecera y área de datos.



Segmento TCP.

Puerto Fuente/ Puerto Destino: Estos campos de 16 bits identifican a los programa de aplicación de nivel superior que utilizan la conexión TCP.

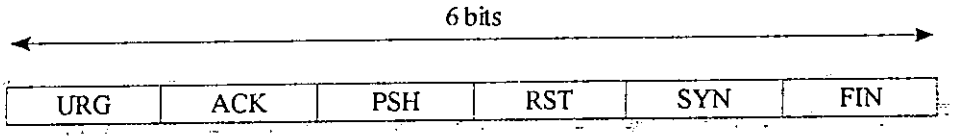
Número de Secuencia: Contiene el número de secuencia del primer octeto del campo de datos del usuario. Su valor define la posición de la cadena de bits del mensaje transmitido. Este campo se usa también durante la gestión de la conexión. Si dos entidades TCP usan el segmento de solicitud de conexión, entonces el número de secuencia especifica el *número de secuencia de envío inicial (ISS)* que se usará para la numeración de los datos de usuario.

Número de Aceptación: Permite aceptar los datos previamente recibidos. Este campo contiene el valor del número de secuencia del siguiente octeto que espera recibir del transmisor. TCP permite la aceptación inclusiva, lo que significa que permite la aceptación de todos los octetos hasta e incluyendo, el valor de ese número menos 1 (n-1).

Desplazamiento de Datos: Especifica el número de palabras alineadas de 32 bits de que consta la cabecera TCP. Este campo se usa para determinar dónde comienza el campo de datos.

Reservado: Este campo de 6 bits está reservado para usos futuros. Generalmente se rellena con ceros (0).

Banderas: Los 6 bits de éste campo son los siguientes:



Bits que componen el campo ***Banderas.***.

URG: Indica que el campo de puntero de urgencia es significativo.
ACK: Indica si el campo de aceptación es significativo.
PSH: Significa que el módulo usará la función PUSH.
RST: Indica que la conexión se va a reinicializar (RESET).
SYN: Indica que se van a sincronizar los números de secuencia. Se usa en los segmentos de establecimiento de la conexión para indicar que se realizarán algunas operaciones de preparación.
FIN: Indica que el remitente no tiene más datos que enviar (Fin de transmisión).

Ventana: Este campo indica cuántos octetos desea recibir el receptor como máximo. Este valor se establece teniendo en cuenta el valor del campo de aceptación (Número de Aceptación). La ventana se establece sumando los valores del campo de ventana y del campo de número de aceptación.

Suma de Verificación: Contiene el complemento a 1 de 16 bits del complemento a 1 de la suma de todas las palabras de 16 bits del segmento, incluyendo cabecera y texto. El objetivo de este campo es verificar si el segmento ha llegado sin errores al receptor.

Puntero de Urgencia: Tiene validez solamente si ella bandera URG está activa. El objeto de este puntero es identificar el octeto de datos al que le siguen datos urgentes. Los datos urgentes también se conocen como *datos fuera de banda*. TCP no dice lo que se debe hacer con los datos urgentes, por lo que su tratamiento depende de la aplicación. El valor de este campo es un desplazamiento del número de secuencia y apunta al octeto a partir del cual siguen los datos urgentes.

Opciones: El campo de opciones está hecho para implementar mejoras futuras a TCP. Cada opción está compuesta de 3 campos: el número de opción, la longitud de la opción y finalmente los valores de la opción propiamente dichos.

Actualmente, el campo de opción tiene un uso muy limitado y el estándar TCP sólo especifica 3 opciones:

0: Fin de lista de opciones.
1: No operación.
2: Tamaño máximo de segmento.

Relleno: Este campo asegura que la cabecera TCP tenga un tamaño que es un múltiplo de 32 bits.

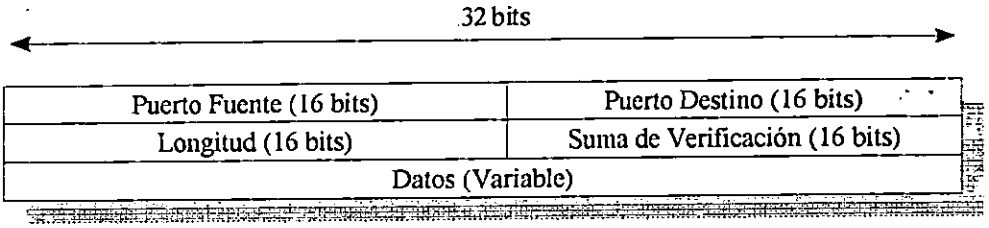
El Protocolo UDP (User Datagram Protocol)

UDP es un protocolo de nivel de transporte (Capa 4); pero, a diferencia de TCP, no es orientado a la conexión, por lo que no proporciona fiabilidad, mecanismos de control de flujo ni procedimientos de recuperación de errores. UDP se usa a veces como sustituto de TCP cuando no es necesario usar los servicios de este último.

UDP suele usarse junto con IP como Multiplexor/Demultiplexor del envío y recepción del tráfico IP. UDP hace uso del concepto de puerto para dirigir los datagramas hacia las aplicaciones de nivel superior.

Formato de Mensaje UDP

El protocolo UDP tiene el siguiente formato:



Formato de Mensaje UDP.

Puerto Fuente: Identifica el puerto del proceso de aplicación remitente. Este campo es opcional. Si no se usa, su valor es de cero (0).

Puerto Destino: Este campo identifica el proceso de recepción el el ordenador destino.

Longitud: Indica la longitud del datagrama de usuario, incluyendo cabecera y datos. La longitud mínima es de 8 octetos.

Suma de Verificación: Contiene el complemento a 1 de 16 bits del complemento a 1 de la suma de todas las palabras de 16 bits de la cabecera y los datos.

UDP representa el nivel de servicio mínimo que usan muchos sistemas de aplicación basados en transacciones. Sin embargo, es muy útil en los casos en los que no son necesarios los servicios de TCP.