IMPLEMENTACION Y DESARROLLO DE PRACTICAS DE APOYO PARA LA ENSEÑANZA EN LOS SISTEMAS DISTRIBUIDOS EN LA C.U.T.B.

EFRAIN ORTIZ DÍAZ

GABRIEL OLIVEROS VALENCIA

CORPORACION UNIVERSITARIA TECNOLOGICA DE BOLIVAR

FACULTAD DE INGENIERIA DE SISTEMAS

CARTAGENA D.T. Y C.

2000

IMPLEMENTACION Y DESARROLLO DE PRACTICAS DE APOYO PARA LA ENSEÑANZA EN LOS SISTEMAS DISTRIBUIDOS EN LA C.U.T.B.

EFRAIN ORTIZ DÍAZ

GABRIEL OLIVEROS VALENCIA

Trabajo de grado presentado para optar al título de Ingeniero de Sistemas

Asesor
CARLOS VALENCIA MTZ.
Ingeniero Electricista

CORPORACION UNIVERSITARIA TECNOLOGICA DE BOLIVAR

FACULTAD DE INGENIERIA DE SISTEMAS

CARTAGENA D.T. Y C.

2000

CONTENIDO

Pág

INTRODUCCION

Con el transcurrir de los días y el veloz adelanto tecnológico, el área de la ciencia de la computación se enriquece aceleradamente. Generando subáreas de interés y desarrollo como es el caso que hoy nos ocupa: Los Sistemas Distribuidos (Colección de computadoras independiente que aparecen ante los usuarios como una única computadora).

Los Sistemas Distribuidos surgen del avance tecnológico en los mecanismos de interconexión de los equipos de computo (LAN) y los nuevos paradigmas de programación (aplicados a los sistemas operativos) que al unirlos y usarlos eficientemente entran a resolver los problemas más comunes que se presentaban en los sistemas centralizados y por ende a mejorar los modelos de la mayoría de los sistemas informáticos de hoy.

Por ejemplo, los sistemas distribuidos:

Mejoran el rendimiento de los sistemas de información

Descentralizan los procesos

Permiten una mejor asignación de recursos

Son más tolerantes a fallas

Hacen transparentes los procesos de comunicación entre las máquinas

Permiten un crecimiento escalonado del sistema

... entre otros

Para un sistema distribuido es fundamental el hardware y el software. Por lo primero no existe mucho inconveniente, pero por lo segundo, se hace necesario de un software radicalmente diferente al de los sistemas centralizados. Sin embargo se necesita de un proceso continuo de cambio hasta llegar a un sistema distribuido 100%. En dicho proceso se han creado modelos de programación como el modelo cliente/servidor que permite la creación de sistemas con alguas de las características de un sistema distribuido.

Un ejemplo del uso de la filosofía cliente/servidor en la creación de un modelo aproximado a un sistema distribuido es la WWW (world wide web). En ella a través de una aplicación cliente, el usuario, puede navegar por un gran conjunto de sistemas de cómputo (servidores) e interactuar con ellos sín importar que tan cerca o distante se puedan encontrar (eso es transparente para el usuario).

Bien es cierto que no todo está dicho en cuanto a los sistemas distribuidos pero es la tendencia de los sistemas informáticos que hoy se desarrollan. Por tanto la C.U.T.B. comprometida con un proceso de formación de excelencia de sus estudiantes, ha impulsado la creación de un laboratorio para la asignatura Sistemas Distribuidos, que apoye el proceso de enseñanza/aprendizaje. Dicho laboratorio hace parte de un

proyecto macro en el departamento de Ingeniería de Sistemas al cual nos vinculamos a través de la figura de una Tesis de Grado y luego de varios meses de trabajo presentamos este documento y una serie de productos complementarios (relacionados más adelante) que esperamos contribuyan positivamente en el proceso académico para el cual fue diseñado.

JUSTIFICACION.

Consideramos la viabilidad de este proyecto que hemos iniciado, debido a que:

- Brindará apoyo a las actividades de investigación que deben acompañar el desarrollo teórico de la asignatura que nos ocupa, como es el caso de Sistemas Distribuidos.

- Este trabajo de grado hace parte de un proyecto macro de laboratorios en la facultad de Ingeniería de Sistemas de la CUTB, para beneficiar el proceso de enseñanza aprendizaje. Este proyecto de laboratorios involucra otras asignaturas como: sistemas operativos, comunicación de datos y redes. Donde también se adelantan proyectos paralelos a este, bajo el mismo rótulo de trabajos de grado.

- Se puede tomar como un punto de partida para proyectos futuros, tendientes a tocar otros aspectos relacionados con los sistemas distribuidos o proyectos relacionados con el campo de la Ingeniería de Sistemas en general, debido a lo extenso de esta área de la ciencia.

OBJETIVOS

GENERAL

Implementación y desarrollo de una serie de prácticas de laboratorio de Sistemas Distribuidos en el área de la comunicación entre procesos, como un complemento para la enseñanza, en el interior de dicha asignatura vinculada con la facultad de Ingeniería de sistemas de la C.U.T.B.

ESPECÍFICOS

- Montar un servidor de LINUX que sirva como base para la implementación de las prácticas del laboratorio de sistemas distribuidos y desarrollo de proyectos futuros.
- Diseñar e implementar un conjunto de pruebas para un laboratorio de sistemas distribuidos en las subáreas de:
  - ✓ Comunicación entre procesos: Usando los modelos de comunicación
    - RPC
    - Cliente/Servidor
    - Comunicación en grupo.
  - ✓ Programación Paralela:
    - PVM

- El número de prácticas que proponemos es de la siguiente manera:
  - ✓ RPC: 5 prácticas
  - ✓ Cliente/Servidor: 5 prácticas
  - ✓ Comunicación en grupo: 5 prácticas
  - ✓ PVM: 5 prácticas

  Para un gran total de 20 prácticas que estarán a disposición del docente de la materia para la asignación durante el semestre académico.
- Proponemos como base de procedimiento de estas prácticas una estructura que ayude al estudiante en el desarrollo de las mismas, de la siguiente forma:
  - ✓ Sustentación teórica
  - ✓ Objetivos por alcanzar
  - ✓ Contenido de la práctica en sí
  - ✓ Cuestionario de retroalimentación final.
- Lo anterior será recopilado en un manual de laboratorio de sistemas distribuidos, tanto para el docente como para el estudiante.

1 MARCO TEORICO

1.1 SISTEMAS DISTRIBUIDOS

Un sistema distribuido es un conjunto de computadoras autónomas ligadas en red que aparecen ante los usuarios del sistema como una única computadora.

1.2 COMUNICACIÓN ENTRE PROCESOS

1.2.1 Cliente/servidor

1.2.2 Conceptos y definiciones del modelo c/s: protocolos y conexiones.

La comunicación mediante sockets es una interfaz con la capa de transporte (nivel 4) de la jerarquía OSI.

Sin embargo, la interfaz de acceso a la capa de transporte del sistema UNIX no está totalmente aislada de las capas inferiores, por lo que a la hora de trabajar con sockets es necesario conocer algunos detalles sobre esas capas. En concreto, a la hora de establecer una conexión mediante sockets, es necesario conocer la familia o dominio de la conexión y el tipo de conexión.

- Una familia agrupa todos aquellos sockets que comparten características comunes, tales como protocolos, convenios para formar direcciones de red, convenios para formar nombre, etc.
- El tipo de conexión indica el tipo de circuito que se va a establecer entre los dos procesos que se están comunicando. El circuito puede ser virtual (orientado a la conexión) o datagrama (no orientado a conexión). Para establecer un circuito virtual, se realiza una búsqueda de enlaces libres que unan los computadores a conectar. Una vez establecida la conexión, se puede proceder al envío secuencial de los datos, ya que la conexión es permanente. Por el contrario, los datagramas no trabajan con conexiones permanentes. La transmisión por los datagramas es a nivel de paquetes, donde cada paquete puede seguir una ruta distinta y no se garantiza una recepción secuencial de la información.

1.2.2.1 ***Generalidades del manejo de sockets y esquema de funcionamiento de la filosofía cliente/servidor***. A la hora de referirse a un nodo de la red, cada protocolo implementa un mecanismo de direccionamiento. La dirección distingue de forma inequívoca a cada nodo o computador y es utilizada para encaminar los datos desde el nodo origen al nodo destino. La forma de construir direcciones depende de los protocolos que se empleen en la capa de transporte y de red. Hay muchas llamadas al sistema UNIX que necesitan un puntero a una estructura de dirección de socket para trabajar**.** Esta estructura se define el fichero de cabecera <sys/socket.h> y su forma es la siguiente:

```
struct sockaddr {
  u_short sa_family;  /*Familia de sockets. Se emplean las constantes de la forma
                          AF_***  */
  Char sa_data[14];    /*14 bytes que contiene la dirección. Su significado depende de
                          la familia de sockets que se esté empleando */
};
```

Si usamos una familia que emplea protocolos internet, la forma de las direcciones de red será la definida en el fichero de cabecera <netined/in.h>:

```
struct in_addr {
  u_long s_addr;  /*32 bits que contienen la identificación de la red y del host. */
};
```

```
struct sockaddr_in {
  short sin_family;   /*AF_INET */
  u_short sin_port;   /*16 bits con el número de puerto */
  struct in_addr sin_addr;  /*32 bits con la identificación de la red y del host. */
  char sin_zero[8];  /*8 bytes no usados.*/
};
```

La familia de sockets conocida como UNIX domain emplea direcciones con la forma definida en <sys/un.h>:

```
struct sokaddr_un {
  short sun_family;  /* AF_UNIX */
  char sun_path[108];  /* Path name. */
};
```

Estas direcciones se corresponden en realidad con path names de ficheros y su longitud (110 bytes) es superior a los 16 bytes que de forma estándar tienen las direcciones del resto de familias. Esto es posible debido a que esta familia de sockets se utiliza para comunicar procesos que se están ejecutando bajo el control de una misma máquina, por lo que no necesitan hacer accesos a la red.

Por otra parte, el modelo Cliente/Servidor es el modelo estándar de ejecución de aplicaciones en una red.

Un servidor es un proceso que se está ejecutando en un nodo de la red y que gestiona el acceso a un determinado recurso. Un cliente es un proceso que se ejecuta en el mismo o en diferente nodo y que realiza peticiones de servicio al servidor. Las peticiones están originadas por la necesidad de acceder al recurso que gestiona el servidor.

Las acciones que debe llevar a cabo el programa servidor son las siguientes:

1. Abrir el canal de comunicaciones e informar a la red tanto de la dirección a la que responderá como de su disposición para aceptar peticiones de servicio.
2. Esperar a que un cliente le pida servicio en la dirección que él tiene declarada.
3. Cuando recibe una petición de servicio, si es un servidor interactivo atenderá al cliente. Los servidores interactivos se suelen implementar cuando la respuesta que necesita el cliente es sencilla e implica poco tiempo de proceso. Si el servidor es concurrente creará un proceso mediante fork para que le de servicio al cliente.
4. Volver al punto 2 para esperar nuevas peticiones de servicio.

El programa cliente, por su parte, llevará a cabo las siguientes acciones:

1. Abrir el canal de comunicaciones y conectarse a la dirección de red atendida por el servidor.

2. Enviar al servidor un mensaje de petición de servicio y esperar hasta recibir la respuesta.
3. Cerrar el canal de comunicaciones y terminar la ejecución.

1.2.2.2 ***Métodos de direccionamiento***. Para que un cliente pueda enviar un mensaje a un servidor, debe conocer la dirección de éste.

Si sólo existe un proceso en ejecución en la máquina destino, el núcleo sabrá que hacer con el mensaje recibido (dárselo al único proceso en ejecución). Sin embargo, ¿qué ocurre si existen varios procesos en ejecución en la máquina destino?

Otro tipo de sistema de direccionamiento envía mensajes a los procesos en vez de a las máquinas. Un esquema común consiste en utilizar nombres con dos partes, para especificar tanto la máquina como el proceso.

Una ligera variación de este esquema de direccionamiento utiliza machine. Local–id en vez de machine.process. El campo local–id es por lo general, un entero aleatorio de 16 o 32 bits (o el siguiente de una serie).

Existe otro método más para la asignación de identificadores a los procesos, el cual consiste en dejar que cada proceso elija el propio identificador de un gran espacio de direcciones dispersas, como el espacio de enteros binarios de 64 bits. Sin embargo, aquí también existe un problema: ¿Cómo sabe el núcleo emisor a cuál máquina enviar el mensaje? En una LAN que soporte transmisiones, el emisor puede transmitir un **paquete especial de localización** con la dirección del proceso destino.

Aunque este esquema es transparente, incluso con ocultamiento, la transmisión provoca una carga adicional en el sistema. Esta carga se evita mediante una máquina

adicional para la asociación a alto nivel (es decir, en ASCII) de los nombres de servicios con las direcciones de las máquinas. Lo anterior se conoce a menudo como **servidor de nombres.**

Un método por completo distinto utiliza un hardware especial. Se deja que los procesos elijan su dirección en forma aleatoria.

1.2.2.3 LLamadas para el manejo de sockets.

Apertura de un punto terminal en un canal. Socket:

La llamada para abrir un canal bidireccional de comunicaciones es ***socket,*** y se usa de la siguiente manera:

socket (af, types, protocol);

Socket crea un punto terminal para conectarse a un canal y devuelve un descriptor. El descriptor de socket devuelto se usará en llamadas posteriores a funciones de la interfaz.

Af (address family) especifica la familia de sockets o familia de direcciones que se desea emplear. Las distintas familias están definidas en el fichero de cabecera <sys/socket.h>. Las dos familias siguientes suelen estar presentes en todos los sistemas:

AF_UNIX Protocolos internos UNIX. Es la familia de sockets empleada para comunicar procesos que se ejecutan en una misma máquina. Esta familia no requiere que esté presente un hardware especial de red

AF_INET Protocolos Internet. Es la familia de sockets que se comunican mediante protocolos, tales como TCP o UDP.

El argumento types indica la semántica de la comunicación para el socket. Puede ser:

SOCK_STREAM socket con un protocolo orientado a conexión.
SOCK_DGRAM socket con un protocolo no orientado a conexión o datagrama.

Protocol especifica el protocolo particular que se va a usar en el socket.

Nombre de un socket. Bind
La llamada bind se utiliza para unir un socket a una dirección de red determinada. Se usa de la siguiente manera:

bind (sfd, addr, addrlen);

Cuando se crea un socket con la llamada socket, se le asigna una familia de direcciones, pero no una dirección particular. Bind hace que el socket de descriptor sfd se una a la dirección de socket específica en la estructura apuntada por addr. Addrlen indica el tamaño de la dirección.

Disponibilidad para recibir peticiones de servicio. Listen

Cuando se abre un socket orientado a conexión, el programa servidor indica que está disponible para recibir peticiones de conexión mediante la llamada a listen. La cual se emplea de la siguiente forma:

listen (sfd, backlog)

Listen habilita una cola asociada al socket descrito por sfd. Esta cola se va a encargar de alojar peticiones de conexión procedentes de los procesos clientes. La longitud de esta cola es la especificada en el argumento backlog. Para que la llamada a listen tenga sentido, el socket debe ser del tipo SOCK_STREAM (socket orientado a conexión).

1.2.2.4 Primitivas de transferencia y recepción de mensajes.

En las primitivas de bloqueo (a veces llamadas primitivas síncronas) cuando un proceso llama a send, especifica un destino y buffer dónde enviar ese destino. Mientras se envía el mensaje, el proceso emisor se bloque (es decir, se suspende). La instrucción que sigue a la llamada a send no se ejecuta sino hasta que el mensaje se envía en su totalidad.

Una alternativa a las primitivas con bloqueo son las **primitivas sin bloqueo** (a veces llamadas **primitivas asíncronas**). Si send no tiene bloqueo, regresa de inmediato el control a quien hizo la llamada, antes de enviar el mensaje.

Así como los diseñadores de sistemas pueden elegir entre las primitivas con o sin bloqueo, también pueden elegir entre las primitivas almacenadas en buffer o no almacenadas.

Los mensajes se pueden perder, lo cual afecta la semántica del modelo de transferencia de mensajes. Supongamos que se utilizan las primitivas por bloqueo. Cuando un cliente envía un mensaje, se le suspende hasta que el mensaje ha sido enviado. Sin embargo, cuando vuelve a iniciar, no existe garantía alguna acerca de la entrega del mensaje.

Existen tres enfoques de este problema. El primero consiste en volver a definir la semántica de **send** para hacerla no confiable. El sistema no da garantía alguna acerca de la entrega de los mensajes. La implantación de una comunicación confiable se deja por completo en manos de los usuarios.

El segundo método exige que el núcleo de la maquina receptora envíe un reconocimiento a la maquina emisora. Sólo cuando se reciba este reconocimiento, el

núcleo emisor liberará el proceso usuario (cliente). El reconocimiento va de un núcleo al otro ; ni el cliente ni el servidor ven alguna vez un reconocimiento. De la misma forma que la solicitud de un cliente a un servidor es reconocida por el núcleo del servidor, la respuesta del servidor es reconocida por el núcleo del cliente. Así una solicitud de respuesta consta de cuatro mensajes.

El tercer método aprovecha el hecho de que la comunicación cliente/servidor se estructura como solicitud del cliente al servidor, seguida de una respuesta del servidor al cliente.

Petición de conexión. Connect.

Para que un proceso cliente inicie una conexión con un servidor a través de un socket, es necesario que haga una llamada a connect. Esta función se declara de la forma siguiente:

```
connect (sfd, addr, addrlen);
```

sfd es el descriptor del socket que da acceso al canal y addr es un puntero a una estructura que contiene la dirección del socket remoto al que queremos conectarnos. Addrlen es el tamaño en bytes. La estructura de la dirección dependerá de la familia de sockets con la que estemos trabajando.

Aceptación de una conexión. Accept

Los procesos servidores van a leer peticiones de servicios mediante la llamada accept.

Llamada:

accept (sfd, addr, addrlen)

Esta llamada se usa con sockets orientados a conexión como el tipo SOCK_STREAM. El argumento sfd es un descriptor de socket creado por una llamada previa a socket y unido a una dirección mediante bind. Accept extrae la primera petición de conexión que hay en la cola de peticiones pendientes creada con una llamada previa a listen. Una vez extraída la petición de conexión, accept crea un nuevo socket con las mismas propiedades que sfd y reserva un nuevo descriptor de fichero (nsfd) para él.

El socket original (sfd) permanece abierto y puede aceptar nuevas conexiones; sin embargo, el socket recién creado (nsfd) no puede usarse para aceptar más conexiones.

El argumento addr debe apuntar a una estructura local de dirección de socket. La llamada accept rellenará esa estructura con la dirección del socket remoto que pide la conexión. El argumento addrlen debe ser un puntero a int. Inicialmente, debe contener el tamaño en bytes de la estructura de dirección. La función sobreescribirá en addrlen el tamaño real de la dirección leída de la cola de direcciones.

Lectura o recepción de mensajes de un socket.

Una vez que el canal de comunicación entre los procesos servidor y cliente está correctamente inicializado y ambos procesos disponen de un conector (socket) con el canal, contamos con cinco llamadas al sistema para leer datos/mensajes de un socket y otras cinco llamadas para escribir datos/mensajes en él.

Las llamadas para leer datos de un socket son: read, readv, recv, recvfrom, recvmsg.

Escritura o envío de mensaje a un socket.

Las llamadas para escribir datos en un socket son: write, writev, send, sento, sendmsg.

Cierre del canal. Close.

Una vez que un proceso no necesita realizar más accesos a un socket, puede desconectarse del mismo. Para ello, y aprovechando que un socket es tratado sintácticamente como si fuera un fichero, podemos usar la llamada close. Esta llamada va a cerrar el sockets en sus dos sentido (servidor-cliente y cliente-servidor).

1.2.2.5 Tipos de servidores. El modelo Cliente/Servidor es el modelo estándar de ejecución de aplicaciones en una red**.**

Un servidor es un proceso que se está ejecutando en un nodo de la red y que gestiona el acceso a un determinado recurso. Un cliente es un proceso que se ejecuta en el mismo o en diferente nodo y que realiza peticiones de servicio al servidor. Las peticiones están originadas por la necesidad de acceder al recurso que gestiona el servidor.

El servidor está continuamente esperando peticiones de servicio. Cuando se produce una petición, el servidor despierta y atiende al cliente. Cuando el servicio concluye, el servidor vuelve al estado de espera. De acuerdo con la forma de prestar el servicio, podemos considerar dos tipos de servidores:

- Servidores interactivos: El servidor no solo recoge la petición de servicio, sino que él mismo se encarga de atenderla.

- Servidores concurrentes: El servidor recoge cada una de las peticiones de servicio y crea otros procesos para que se encarguen de atenderlas. Este tipo de servidores solo es aplicables en sistemas multiprocesos, como es UNIX. La ventaja que tiene este tipo de servicio es que el servidor puede recoger peticiones a muy alta velocidad, porque está descargado de la tarea de atención del cliente.

1.2.3 LLamada a procedimiento remoto (rpc)

1.2.3.1 ***Conceptos generales de las rpc.*** Aunque el modelo cliente/servidor es una forma conveniente de estructurar un sistema operativo distribuido, adolece de una enfermedad incurable; el paradigma básico en torno al cual se construye la comunicación es la entrada/salida**.** Los procedimientos send y receive están reservados para la realización de E/S no es uno de los conceptos fundamentales de los sistemas centralizados, el hecho de que sean la base del cómputo distribuido es visto por las personas que laboran en este campo como un grave error. Su objetivo es lograr que el cómputo distribuido se vea como el cómputo centralizado. La construcción de todo en torno de la E/S no es la forma de lograrlo.

La mayoría de los sistemas de ordenadores están conectados en red, soportando comunicación de datos entre ellos. Como resultado de esto, se han desarrollado muchas técnicas para soportar el desarrollo de aplicaciones que requieren procesos en diferentes sistemas, para comunicar y coordinar sus actividades. Una de estas técnicas son las RPC "Remote Procedure Call", (llamadas a procedimientos remotos)

El concepto de RPC es una sencilla técnica para desarrollar aplicaciones donde se requiere la comunicación entre procesadores que cooperan en un sistema distribuido. RPC es una técnica consistente, como evidencia la existencia de muchas especificaciones e implementaciones.

El mecanismo RPC proporciona un servicio para el programador de aplicaciones que le permite el uso transparente de un servidor para proporcionar alguna actividad por

parte de la aplicación. Esto efectivamente puede ser utilizado para interactuar con un servidor computacional o con una Base de Datos, y ha sido usado por algunos sistemas para proporcionar acceso a servicios del sistema operativo. El último uso conocido es en sistemas basados en microordenadores, que da un mejor resultado que en los sistemas tradicionales encontrados en la mayoría de sistemas UNIX, y es especialmente utilizado en sistemas operativos distribuidos.

Las RPC son expresadas como procedimientos ordinarios. Estas llamadas no requieren un compilador especial para el código fuente del programa. Las ventajas de esto incluyen:

- Transparencia: La capa RPC puede ser reemplazada con llamadas a funciones directas si llegan a estar disponibles.
- Familiaridad de la interface: La mayoría de programadores están acostumbrados a una forma de llamadas a procedimientos. Esto permite una fácil adaptación al mecanismo RPC en sistemas ya implantados.

1.2.3.2 Transferencia de parámetros.

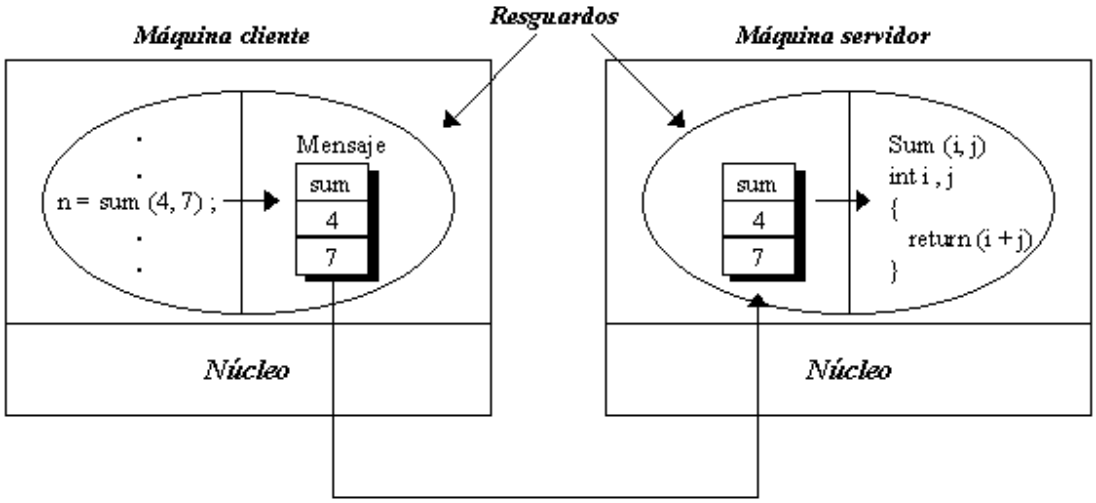
Las RPC son realmente una manera de ocultar el protocolo de pase de mensajes. Esto proporciona una manera adecuada de permitir una interpretación a alto nivel, abstrayendo el mecanismo de comunicación a bajo nivel, es decir haciendo transparente al programador el protocolo de red. Las RPC tienen la intención de ser parecidas a una llamada a procedimiento ordinario, pero atravesando la red transparentemente. Un proceso

realiza una RPC poniendo sus parámetros y una dirección de retorno en la pila, y salta al comienzo del procedimiento. El procedimiento es responsable de acceder y usar la red. Cuando la ejecución remota finaliza, el procedimiento salta hacia atrás a la dirección de retorno. El proceso de llamada (cliente) entonces continua**.**

La función del resguardo (figura 1.) del cliente es tomar sus parámetros, empacarlos en un mensaje y enviarlos al resguardo del servidor. Aunque esto parece directo, no es tan sencillo como aparenta. El empacamiento de parámetros en un mensaje se denomina **ordenamiento de parámetros.**

El ejemplo más sencillo es considerar un procedimiento remoto, *sum*(i,j), que toma dos parámetros enteros y regresa su suma aritmética.

La llamada a *sum*, con los parámetros 4 y 7, aparece en la parte izquierda del proceso cliente de la figura 1. El resguardo del cliente toma sus dos parámetros y los coloca en un mensaje de la forma que se indica. También coloca en el mensaje el nombre o número del procedimiento por llamar, puesto que el servidor podría soportar varias llamadas y se le tiene que indicar cual de ellas se necesita.



Cálculo remoto de sum (4 , 7).

Figura 1. Calculo remto de sum (4,7).

Cuando el mensaje llega al servidor, el resguardo examina éste para ver cuál procedimiento necesita y entonces lleva a cabo la llamada apropiada.

Cuando el servidor termina su labor, su resguardo recupera el control. Toma el resultado proporcionado por el servidor y lo empaca en un mensaje. Este mensaje se envía de regreso al resguardo de cliente, que lo desempaca y regresa el valor al procedimiento cliente.

Si las máquinas cliente y servidor son idénticas y todos los parámetros y resultado son de tipo escalar, como enteros, caracteres o booleanos, este método funciona bien. Sin embargo, en un sistema distribuido de gran tamaño, es común tener distintos tipos de máquinas. Cada una tiene a menudo su propia representación de los números, caracteres y otros elementos. Por ejemplo, las mainframes de IBM utilizan el código

de caracteres EBCDIC, mientras que las computadoras personales de IBM utilizan ASCII. en consecuencia, no es posible pasar un parámetro carácter de una PC de IBM cliente a una mainframe IBM servidor, mediante el sencillo esquema de la figura 1; el servidor interpretará lo interpretará de manera incorrecta.

Aparecen otros problemas similares con la representación de enteros (complemento a 1 o complemento a 2) y de manera particular, en los números de punto flotante. Además existe un problema mucho más irritante, puesto que en ciertas máquinas, como la Intel 486, numeran sus bytes de derecha a izquierda, mientras que otras, como la Sun SPARC, los numeran en el orden contrario. El formato de Intel se llama **little endian** (partidarios del extremo menor) y el de SPARC **big endian** (partidarios del extremo mayor).

1.2.3.3 Protocolos rpc.

En teoría, cualquier protocolo antiguo puede funcionar si obtiene los bits del núcleo del cliente y los lleva al núcleo del servidor; pero en la práctica hay que tomar decisiones importantes en este punto, decisiones que pueden tener un fuerte impacto en el desempeño. La primera decisión está entre un protocolo orientado a la conexión o un protocolo sin conexión. En el primer caso, al momento en que el cliente se conecta con el servidor, se establece una conexión entre ellos. Todo el tráfico, en ambas direcciones, utiliza esta conexión.

La ventaja de contar con una conexión es que la comunicación es más fácil. Cuando un núcleo envía un mensaje, no tiene que preocuparse por su pérdida, ni tampoco por los reconocimientos. Todo ello se maneja a nivel inferior, mediante que el software

que soporta la conexión. Cuando se opera una red amplia, esta ventaja es con frecuencia irresistible.

La desventaja en una LAN, es la pérdida de desempeño. Todo ese software adicional estorba en el camino. Además, la ventaja principal (no perder los paquetes) difícilmente se necesita en una LAN, puesto que las LAN son confiables en este sentido. En consecuencia, la mayoría de los sistemas distribuidos que pretenden utilizarle en un edificio o campus utilizan los protocolos sin conexión.

La segunda opción principal está en utilizar un protocolo de propósito general o alguno diseñado de forma específica para RPC. Puesto que no existen estándares en esta área, el uso de un protocolo RPC adaptado quiere decir por lo general que cada quien diseñe el suyo.

Algunos sistemas distribuidos utilizan IP (o UDP, integrado a IP) como el protocolo básico.

1.2.3.4 ***Las grandes lineas del protocolo***. El protocolo debe permitir:

- La identificación del procedimiento
- La autentificación de la petición

La identificación del procedimiento

El principio es el de agrupar los diferentes procedimientos en un programa. Los diferentes procedimientos que constituyen un mismo programa contribuyen a la realización de un servicio específico. Por ejemplo, el protocolo NFS constituye un programa de protocolo RPC y contiene diferentes procedimientos cuyo punto común es que todos permitan manipular archivos a distancia.

Un programa se identificará por un número entero y cada procedimiento de un programa será igualmente identificado por un entero. A título de ejemplo el programa NFS lleva el número 100003 y los procedimientos de lectura y escritura llevan los números 6 y 8. Finalmente para permitir la evolución de los programas, cada uno posee un número de versión.

La autenficación

La definición del protocolo prevee la posibilidad de que un cliente se identifique a un servidor, lo que permite asegurar la seguridad de los accesos a los objetos del sistema distante. Los mensaje intercambiados en el curso de llamadas a procedimientos remotos llevan información relativa a esta identificación. Como el protocolo es independiente del sistema subyacente, están previstos diferentes estilos de autentificación (por ejemplo, la ausencia de autentificación o una autentificación UNIX), dejando la posibilidad de definir otros nuevos.

1.2.3.5 Los diferentes niveles de utilización. Existen tres

niveles de utilización del mecanismo de RPC tal como ha sido implantado en UNIX. Cada uno de estos niveles ofrece transparencia al usuario, es decir, demanda un conocimiento más o menos fino (hasta nulo) del protocolo y de los mecanismos de más bajo nivel, tales como, por ejemplo, los socket.

El nivel alto

Este es el que oculta el máximo de detalles al usuario. Este solo debe realizar una llamada de función de una biblioteca, especificando el nombre de la máquina objetivo y los diferentes parámetros de la llamada. Evidentemente, a este nivel no es posible desarrollar nuevos servicios RPC (si no es por composición de los servicios existentes).

El nivel intermedio

Es incontestablemente el más interesante para el desarrollador. Supone un conocimiento mínimo de los protocolos XDR y RPC y es suficiente para desarrollar la mayoría de las aplicaciones. Descarga totalmente al usuario de la manipulación de los sockets.

El interfaz disponible a este nivel se apoya sobre el protocolo UDP. Esto significa que el tamaño de los mensajes intercambiados (y. por tanto, el tamaño cié los

parámetros v de los resultados de las funciones llamadas) es limitado. Cuando esta limitación es molesta, es necesario utilizar el interfaz propuesto por el nivel bajo. La sucesión de operaciones a realizar para definir un servicio de este tipo consiste en:

- Escribir las diferentes funciones sobre el servidor;
- Después de haber escogido los números de programa y de versión, solicitar la anotación de las diferentes funciones por el demonio «portmap» por medio de la función **regiserrpc.**

La llamada a una función desde una posición remota se realiza por medio de la función **callrpc.**

El nivel bajo

Su utilización es mucho más compleja y supone un buen conocimiento de los mecanismos concernientes a los sockets. Se muestra necesaria para las aplicaciones donde las opciones elegidas en el nivel intermedio.

1.2.4 Comunicación en grupo

1.2.4.1 Conceptos generales

1.2.4.1.1 Breve orientación teórica. Un grupo es una colección de procesos que actúan juntos en cierto sistema o alguna forma determinada por el usuario. La propiedad fundamental de todos los grupos es que cuando un mensaje se envía al propio grupo, todos los miembros de éste lo reciben. Es una forma de comunicación **uno-muchos** (un emisor, muchos receptores) y contrasta con la **comunicación** puntual de la figura 2.

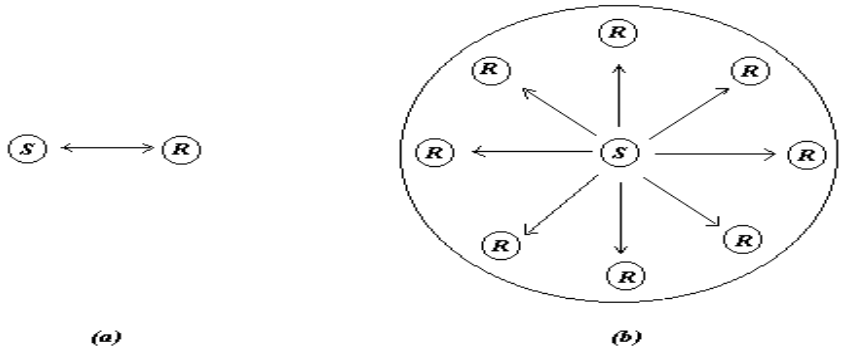


Figura 2 . La comunicación puntual y La comunicación uno-muchos .

Los grupos son dinámicos. Se pueden crear nuevos grupos y destruir grupos anteriores. Un proceso se puede unir a un grupo o dejar otro. Un proceso puede ser miembro de varios grupos a la vez. En consecuencia, se necesitan mecanismos para el manejo de grupos y la membresia de los mismos.

Los grupos son algo parecido a las organizaciones sociales. Una persona puede ser miembro de un club de lectores, un club de tenis y una organización ambientalista. En un día particular, él podría recibir correo (mensajes) que le avisen de un nuevo libro

para hornear pasteles de cumpleaños, el torneo anual del día de las madres y el inicio de una campaña para salvar a las marmotas del sur. En cualquier momento, él es libre de dejar todos o alguno de estos grupos y unirse a otros.

La finalidad de presentar los grupos es permitir a los procesos que trabajen con colecciones de procesos como una abstracción. Así, un proceso puede enviar un mensaje a un grupo de servidores sin tener que conocer su número o su localización, que puede cambiar de una llamada a la siguiente.

En ciertas redes, es posible crear una dirección especial de red (por ejemplo, indicada al hacer que uno de los bits de orden superior tome el valor 1) a la que pueden escuchar varias máquinas. Cuando se envía un mensaje a una de estas direcciones, se entrega de manera automática a todas las máquinas que escuchan a esa dirección. Esta técnica se llama **multitransmisión.**

Las redes que no tienen multitransmisión a menudo siguen teniendo **transmisión simple,** lo que significa que los paquetes que contienen cierta dirección (por ejemplo, 0) se entregan a todas las máquinas.

Por último, si no se dispone de la multitransmisión o la transmisión simple, se puede implantar la comunicación en grupo mediante la transmisión por parte del emisor de paquetes individuales a cada uno de los miembros del grupo. El envío de un mensaje

de un emisor a un receptor se llama a veces **unitransmisión,** para distinguirla de los otros tipos de transmisión.

1.2.4.2 ***Aspectos del diseño***. La comunicación en grupo tiene posibilidades de diseño similares a la transferencia regular de mensajes**,** como el almacenamiento en buffers vs. el no almacenamiento, bloqueo vs**.** no bloqueo, etc. Sin embargo, también existen un gran número de opciones adicionales por realizar, ya que el envío a un grupo es distinto de manera inherente del envío a un proceso**.** Además, los grupos se pueden organizar internamente de varias formas. También se pueden direccionar de formas novedosas que no son importantes en la comunicación puntual.

Grupos cerrados *vs.* grupos abiertos

Los sistemas que soportan la comunicación en grupo se pueden dividir en dos categorías, según quién pueda enviar a quién. Algunos sistemas soportan los grupos cerrados, donde sólo los miembros del grupo pueden enviar hacia el grupo. Los extraños no pueden enviar mensajes al grupo como un todo, aunque pueden enviar mensajes a miembros del grupo en lo Individual. En contraste, otros sistemas soportan los **grupos** abiertos, que no tienen esta propiedad. Si se utilizan los grupos abiertos, cualquier proceso del sistema puede enviar a cualquier grupo. La diferencia entre los grupos cerrados y abiertos se muestra en la figura 3.

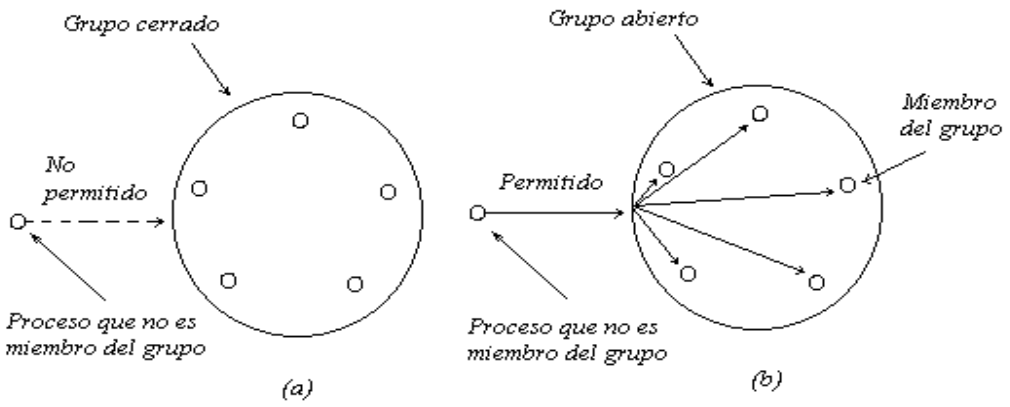


Figura 3. Comparación entre grupos abiertos y cerrados.

Grupos de compañeros *vs.* grupos jerárquicos

La distinción entre los grupos cerrados y abiertos se relaciona con la pregunta de quién se puede comunicar con el grupo. Otra distinción importante tiene que ver con la estructura interna del grupo. En algunos grupos, todos los procesos son iguales. Nadie es el jefe y todas las decisiones se toman en forma colectiva. En otros grupos, existe cierto tipo de jerarquía. Estos patrones de comunicación se muestran en la figura 4.

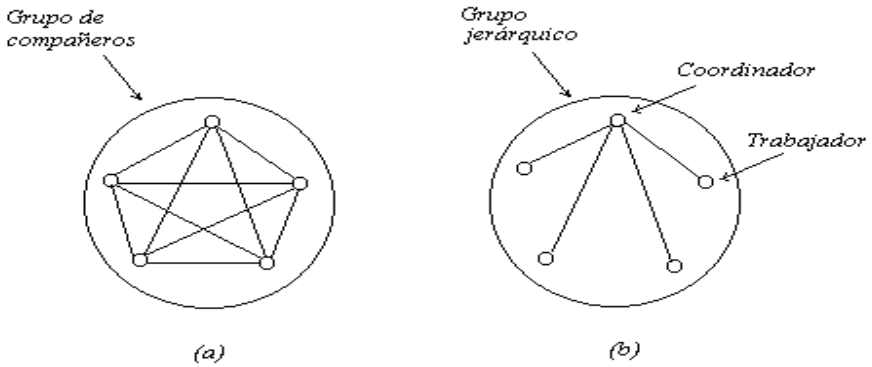


Figura 4. La comunicación en un grupo de compañeros y en un simple Grupo jerarquico.

Cada una de estas organizaciones tiene sus propias ventajas y desventajas. El grupo de compañeros es simétrico y no tiene punto de falla. Si uno de los procesos falla, el grupo sólo se vuelve más pequeño, pero puede continuar.

El grupo jerárquico tiene las propiedades opuestas. La pérdida del coordinador lleva a todo el grupo a un agobiante alto, pero mientras se mantenga en ejecución, puede tomar decisiones sin molestar a los demás.

Membresía del grupo

Si se utiliza la comunicación en grupo, se requiere cierto método para la creación y eliminación de grupos, así como para permitir a los procesos que se unan o dejen grupos. Un posible método es tener un **servidor de grupos** al cual se pueden enviar todas las solicitudes. El servidor de grupos puede mantener entonces una base de datos de todos los grupos y sus membresías exactas. Este método es directo, eficiente

y fácil de implantar. Por desgracia, comparte una desventaja fundamental con todas las técnicas centralizadas: un punto de falla. Si el servidor de grupos falla, deja de existir el manejo de los mismos. Es probable que la mayoría o todos los grupos deban reconstruirse a partir de cero, terminando con todo el trabajo realizado hasta entonces.

El método opuesto es manejar la membresía de grupo en forma distribuida. En un grupo abierto, un extraño puede enviar un mensaje a todos los miembros del grupo para anunciar su presencia. En un grupo cerrado se necesita algo similar (de hecho, incluso los grupos cerrados deben estar abiertos a la opción de admitir otro miembro). Para salir de un grupo, basta que el miembro envíe un mensaje de despedida a todos.

1.2.4.3 Direccionamiento. Para enviar un mensaje a un grupo, un proceso debe tener una forma de especificar dicho grupo. En otras palabras, los grupos deben poder direccionarse, al igual que los procesos. Una forma es darle a cada grupo una dirección, parecida a una dirección de proceso. Si la red soporta la multitransmisión, la dirección del grupo se puede asociar con una dirección de multitransmisión, de forma que cada mensaje enviado a la dirección del grupo se pueda multitransmitir.

Si el hardware no soporta la multitransmisión pero sí la transmisión simple, el mensaje se puede transmitir. Cada núcleo lo recibirá y extraerá de él la dirección del grupo. Si ninguno de los procesos en la máquina es un miembro del grupo, entonces se descarta la transmisión. En caso contrario, se transfiere a todos los miembros del grupo.

Por último, si no se soporta la multitransmisión o la transmisión simple, el núcleo de la máquina emisora debe contar con una lista de las máquinas que tienen procesos pertenecientes al grupo, para entonces enviar a cada una un mensaje puntual. Estos tres métodos de implantación se muestran en la figura 5.

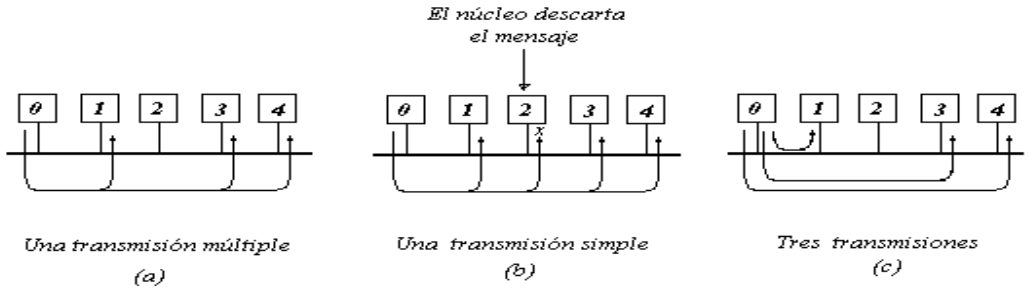


Figura 5. El proceso 0 enviado a un grupo que consta de los procesos 1, 3 y 4**.**

Un segundo método de direccionamiento de grupo consiste en pedir al emisor una lista explícita de todos los destinos (por ejemplo, direcciones IP). Si se utiliza este método, el parámetro de la llamada *send* que especifica el destino es un apuntador a una lista de direcciones.

La comunicación en grupo también permite un tercer método, un tanto novedoso, de direccionamiento, que llamaremos **dircccionamiento de predicados.** Con este sistema, se envía cada mensaje a todos los miembros del grupo (o tal vez a todo el sistema) mediante uno de los métodos ya descritos, pero con nuevo giro. Cada mensaje contiene un predicado (expresión booleana) para ser evaluado. El predicado puede utilizar el número de máquina del receptor, sus variables locales u otros factores. Si el valor del predicado es verdadero, se acepta el mensaje. SÍ es falso, el mensaje se descarta.

Primitivas send y receive

En forma ideal, la comunicación puntual y la comunicación en grupo deberían combinarse en un conjunto de primitivas. Sin embargo, si RPC es el mecanismo usual de comunicación del usuario, en vez de los simples *send y receive,* entonces es difícil combinar RPC y la comunicación en grupo. El envío de un mensaje a un grupo no se puede modelar como llamada a un procedimiento. La principal dificultad es que, con RPC, el cliente envía un mensaje al servidor y obtiene de regreso una respuesta. Con la comunicación en grupo, existen en potencia *n* respuestas diferentes. ¿Cómo podría trabajar una llamada de procedimiento con *n* respuestas? En consecuencia, un método común es abandonar el modelo solicitud/respuesta (en los dos sentidos) subyacente en RPC y regresar a las llamadas explícitas para el envío y recepción (modelo de un sentido).

Atomicidad

Una característica de la comunicación en grupo a la que hemos aludido varias veces es la propiedad del todo o nada. La mayoría de los sistemas de comunicación en grupo están diseñados de forma que, cuando se envíe un mensaje a un grupo, éste llegue de manera correcta a todos los miembros del grupo o a ninguno de ellos. No se permiten situaciones en las que ciertos miembros reciben un mensaje y otros no. La propiedad del todo o nada en la entrega se llama **atomicidad o transmisión atómica.**

La atomicidad es deseable, puesto que facilita la programación de los sistemas distribuidos. Si un proceso envía un mensaje al grupo, no tiene que preocuparse por qué hacer si alguno de ellos no lo obtiene.

Ordenamiento de mensajes

Para que la comunicación en grupo sea fácil de comprender y utilizar, se necesitan dos propiedades. La primera es la transmisión atómica, ya analizada. Ésta garantiza que un mensaje enviado al grupo llegue a todos los miembros o a ninguno. La segunda propiedad se refiere al ordenamiento de mensajes.

La mejor garantía es la entrega inmediata de todos los mensajes, en el orden en que fueron enviados. Si el proceso O envía el mensaje *A* y un poco después el proceso 4 envía el mensaje *B,* el sistema debe entregar en primer lugar *A* a todos los miembros del grupo y después entregar *B* a todos los miembros del grupo. De esta forma, todos los receptores obtiene todos los mensajes en el mismo orden. Este patrón de entrega

es algo comprensible para los programadores y en el cual basan su software. Lo llamaremos **ordenamiento con respecto al tiempo global,** puesto que entrega todos los mensajes en el orden preciso con el que fueron enviados .

Grupos traslapados

Como hemos mencionado, un proceso puede ser miembro de varios grupos a la vez. Este hecho puede provocar un nuevo tipo de inconsistencia. Para ver el problema, observemos la figura 6, la cual muestra dos grupos, 1 y 2. Los procesos *A, B y C* son miembros del grupo 1 y los procesos *B, C* y *D* son miembros del grupo 2.

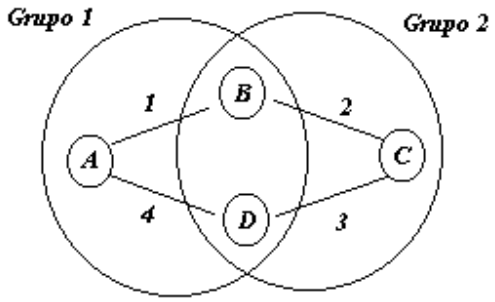


Figura 6. Cuatro procesos A, B, C y D y cuatro mensajes..

El problema aquí es que, aunque existe un ordenamiento con respecto al tiempo global dentro de cada grupo, no es necesario que exista coordinación entre varios grupos. Algunos sistemas soportan un ordenamiento bien definido entre los grupos traslapados y otros no.

Escalabilidad

Nuestro aspecto final del diseño es la escalabilidad. Muchos algoritmos funcionan bien mientras todos los grupos tengan unos cuantos miembros, pero ¿qué ocurre cuando existen decenas, centenas o incluso miles de miembros por grupo? ¿O miles de grupos? Además, ¿qué ocurre si el sistema es tan grande que no cabe en una LAN, de modo que se necesiten varias LAN y compuertas? ¿Qué ocurre si los grupos están diseminados en varios continentes?

La presencia de compuertas puede afectar muchas propiedades de la implantación. Para comenzar, la multitransmisión es más complicada. Consideremos, por ejemplo, una red que consta de cuatro LAN y cuatro compuertas, con el fin de protegerse contra la falla de cualquier compuerta.

Imaginemos que una de las máquinas de la LAN 2 ejecuta una multitransmisión. Cuando el paquete de multitransmisión llega a las compuertas *Gl* y *G3,* ¿qué deben hacer éstas? Si lo descartan, la mayoría de las máquinas nunca lo verán, lo cual destruye su valor como multitransmisión. Sin embargo, si el algoritmo sólo tiene compuertas hacia todas las mul-titransmisiones, entonces el paquete se copiará a la LAN 1 y la LAN 4 y poco después a la LAN 3 dos veces. Peor aún, la compuerta *G2* verá la multitransmisión de *G4,* la copiará a la LAN 2 y viceversa Es claro que se necesita un algoritmo más complejo, que mantenga un registro de los paquetes

anteriores, con el fin de evitar un crecimiento exponencial en el número de multitransmisiones de paquetes.

1.3 PROGRAMACION PARALELA

1.3.1 Generalidades de la programación paralela.

Una instrucción puede especificar, además de varias operaciones aritméticas, la dirección de un dato que puede ser leído o escrito en memoria y/o la dirección de la próxima instrucción a ser ejecutada. Un computador es susceptible de ser programado en términos de éste modelo básico usando el lenguaje de maquina, aunque para la mayoría de los casos resulta complejo, pues se tiene que guardar el "track" de millones de posiciones de memoria y organizar la ejecución de miles de instrucciones de máquinas. Técnicas de diseño modular son aplicadas en la construcción de complejos programas partiendo de simples componentes y tales componentes están estructurados en términos de altos niveles de abstracción tales como estructuras de datos, ciclos interactivos y procedimientos. Abstracciones tales como los procedimientos hacen más fácil el seguimiento de la modularidad observando los objetos que van a ser utilizados, sin tener en cuenta su estructura interna. Entonces hacer lenguajes de alto nivel tales como Fortran, Pascal, C y Ada los cuales guían el diseño expresado en termino de esas abstracciones para ser traducidas automáticamente al código ejecutable.

La programación paralela introduce fuentes adicionales de la complejidad: si programáramos en el nivel más bajo, no sólo aumentaría el número de las instrucciones ejecutadas, sino que también necesitaríamos manejar explícitamente la ejecución de millares de procesadores y coordinar millones de interacciones del interprocessor. Por lo tanto, la abstracción y la modularidad son por lo menos tan importantes como en la programación secuencial. En hecho, acentuaremos la modularidad como cuarto requisito fundamental para el software paralelo, además de la ejecución simultánea, la escalabilidad y la localización.

Algunos conceptos importantes:

Un cómputo: consiste en un conjunto de tareas

Un canal: es una cola de mensaje desde la cual un remitente puede poner mensajes y de cuál un receptor puede quitar un mensaje, y los bloquea si los mensajes no están disponibles. Una colección de mensajes “conectan” a las tareas entre si.

Una tarea: encapsula un programa y una memoria local y define un conjunto de los puertos que definen la interfaz a su ambiente.

El cómputo paralelo consiste en unas o más tareas. Las tareas se ejecutan en paralelo. El número de tareas puede variar durante la ejecución de programa.

1.3.2 El sistema pvm.

PVM es un conjunto integrado de herramientas y librerías de software, que emulan un marco de trabajo de computación concurrente de propósito general, flexible y heterogéneo. Todo esto sobre un conjunto de computadoras de distintas arquitecturas interconectadas entre sí. El principal objetivo del sistema PVM, es permitir que tal conjunto de máquinas, sean usadas en forma cooperativa para hacer computación concurrente o paralela.

Principios

Resumidamente, los principios sobre los cuales se basa PVM incluyen:

- El conjunto de hosts que conforman la máquina virtual puede ser configurado por el usuario (pool de hosts).
- Acceso translucido al hardware.
- Computación basada en procesos.
- Modelo de pasaje de mensajes explícito.
- Soporte heterogéneo (en cuanto a: arquitecturas de hardware, redes y aplicaciones).
- Soporte para Multiprocesadores.

Partes del sistema

El sistema PVM esta compuesto por dos partes. La primera es un demonio, llamado pvmd3 que reside en todas las máquinas que conforman la máquina virtual.

La segunda parte del sistema es una librería de interfaces de rutinas de PVM, que contiene un repertorio de primitivas necesarias para la cooperación entre las tareas de una aplicación. Tales rutinas pueden ser llamadas por el usuario y permiten realizar pasaje de mensajes, expansión de procesos, coordinación de tareas y modificación de la máquina virtual. Las interfaces están provistas para los lenguajes Fortran y C, siendo implementadas como subrutinas en el primer caso y como funciones en el último.

Todas las tareas en PVM son identificadas por un entero llamado TID (del inglés task identifier), estos enteros son suministrados por el demonio local de PVM.

PVM incluye el concepto de grupos de tareas, para ello provee un conjunto de funciones, que permiten a una tarea - entre otras funciones - unirse o dejar un grupo determinado de tareas.

2 PASOS PARA EL DISEÑO DE LAS PRACTICAS

Para la elaboración del esquema que debíamos usar en el desarrollo de nuestra tesis nos basamos en la experiencia asimilada de la participación en laboratorios vinculados a otras asignaturas a lo largo de nuestra vida universitaria. A lo anterior se suma la experiencia de nuestro asesor de la tesis y producto de múltiples conversaciones resultó el esquema que consideramos era el que más nos convenía para la consecución del objetivo general de este trabajo de grado: Que el estudiante tenga un campo de acción donde pueda llevar a la practica toda una serie de conocimientos que se imparten en la asignatura para la cual desarrollamos este trabajo (Sistemas Distribuidos).

Inicialmente nuestro tema de trabajo se ajusto al desarrollo de practicas dentro de 2 áreas de los sistemas distribuidos:

- Comunicación entre procesos
- Programación en paralelo

Para manejar el proceso de enseñanza/aprendizaje, en nuestro trabajo hicimos caso de aquel dicho que reza “dividir y conquistar”, por tanto, dividimos cada uno de los subtemas (Cliente/Servidor, RPC, Comunicación en grupo, PVM) en temas aun mas

pequeños, 7 o 5 subdivisiones, inclusive menos, dependiendo de lo extenso de la teoría necesaria para llevar a cabo las practicas. Para cada uno de estos pequeños temas aplicamos el modelo que habíamos elegido de común acuerdo con el asesor:

- Breve orientación teórica
- Preguntas de concepto
- Preguntas de análisis
- Practica

Cabe anotar que en el anteproyecto sugerimos que con un numero de 5 prácticas por subtema (Cliente/Servidor, RPC, Comunicación en grupo, PVM) el docente podía hacer una asignación de tareas a lo largo del semestre y además le brindaba la posibilidad de asignar para otros periodos académicos practicas que no coincidieran con las practicas del periodo inmediatamente anterior. Por tanto, en aquellos casos donde dividimos un subtema en mas de 5 partes existirán secciones donde no habrá practica que desarrollar pero que si vendrá acompañada de los restantes ítems del esquema que seleccionamos, para tratar el proceso de enseñanza/aprendizaje en el laboratorio.

- Breve orientación teórica
- Preguntas de concepto
- Preguntas de análisis

Volviendo unas líneas atrás, es preciso aclarar que durante el proceso de diseño del esquema que debíamos utilizar, hicimos la confrontación de distintos esquemas.

Dichos esquemas no corresponden a la opinión de un experto en el área de la investigación científica, sino, que corresponden a la formación de sendos esquemas en nuestras mentes producto de la experiencia. Como resultado de la confrontación y análisis de dichos esquemas obtuvimos como resultado el esquema del cual hemos hablado.

3 DESCRIPCION DEL MODELO

Area de los Sistemas Distribuidos

Corresponde a una de las dos grandes divisiones que se presentan en el campo de los Sistemas Distribuidos para un mejor estudio. Comunicación entre procesos, por ejemplo.

Subtemas

Son 4 en total, 3 para el área de comunicación entre procesos (Cliente/Servidor, RPC, Comunicación en grupo) y uno para el área de programación paralela (PVM). Esta organización se debe a que estos subtemas son de gran interés en los Sistemas Distribuidos y sobres los cuales se deben adquirir mas conocimientos, sobre todo para aquellos que se inician en la exploración de temas relacionados con los Sistemas Distribuidos como es el caso de los subsecuentes grupos de estudiantes que harán uso de este material.

Pequeños temas

Corresponden a divisiones temáticas de acuerdo con el volumen de información obtenida en el proceso de recolección y análisis de la documentación pertinente. Estas

divisiones llevan un orden que ayuda al estudiante a introducirse paulatinamente en el desarrollo de cada subtema.

Esquema

Esta sección se encuentra conformada por

- Breve orientación teórica
- Preguntas de concepto
- Preguntas de análisis
- Practicas

Puntos que en su orden consideramos permiten que el estudiante profundice y afiance sus conocimientos en un subtema en particular.

Breve orientación teórica

Como su nombre lo indica es teoría relacionada con un subtema particular. Con esto buscamos que el estudiante se oriente acerca de los conocimientos teóricos (de los Sistemas Distribuidos) y prácticos (programación) que se hacen indispensables para la elaboración de las practicas sugeridas. En caso de que el estudiante quiera profundizar esta teoría en cada sección le sugerimos una fuente de consulta.

Preguntas de concepto

Con esto pretendemos que el estudiante afiance sus conocimientos teóricos.

Preguntas de análisis

Con esto pretendemos que el estudiante utilice la teoría asimilada hasta el momento para resolver situaciones que reten su intelecto.

Por ejemplo: Durante la explicación teórica se puede decir lo que hace una instrucción para el manejo de sockets y luego cuestionarlo acerca del comportamiento de dicha instrucción para un valor en particular pasado como parámetro. De este modo el estudiante deberá sustentarse en la teoría y para mayor seguridad el estudiante puede hacer (programar) lo sugerido y observar su comportamiento.

Practicas

Aquí se encuentra una actividad de programación a través de la cual podrán comprobar el comportamiento real de toda una previa información teórica.

4 DESCRIPCION DE LAS PRACTICAS

4.1 CLIENTE/SERVIDOR

Modelo de programación que busca la cooperación entre procesos. Una aplicación bajo esta filosofía se compone de dos partes (Cliente y Servidor). Este ultimo ofrece servicios a aplicaciones usuarios llamadas clientes.

4.1.1 Practica I

4.1.1.1 ***Enunciado.*** Desarrollar una aplicación que realice las cuatro operaciones básicas (suma, resta, multiplicación y división),usando socket del tipo AF_INET y protocolo TCP/IP.

Laaplicación debe estar compuesta por un cliente, responsable de tomar la expresión a evaluar y enviarla al servidor y de un servidor, quien hará el cálculo y enviará la respuesta.

4.1.1.2 ***Descripcion.*** Con esta práctica se pretende que el estudiante maneje una de las formas de trabajo de la filosofía cliente/serrvidor.

Es una de las formas más sencillas. No se necesita que las expresiones a entrar en el cliente sean muy elaboradas. Pueden ser tan sencillas como 3+5 ó 7*8.

La idea es que se establezca una eficaz comunicación entre los sockets en ambos extremos.

4.1.2 Practica II

4.1.2.1 ***Enunciado.*** Desarrollar una aplicación cliente/servidor donde el cliente capture y envíe una cadena "Hola mundo" al servidor y este a su vez la reciba y visualice. Familia AF_INET. Orientado a la conexión. TCP.

4.1.2.2 ***Descripcion.*** En esta practica se hará necesario la codificación y manejo de los dos extremos en este tipo de comunicación (aplicación cliente y servidor).

Como no existe una sola forma de manejar este tipo de comunicación a través de sockets, en esta oportunidad sugerimos que el socket sea de la familia AF_INET y la comunicación con el protocolo TCP (orientado a la conexión). Con esta restricción el estudiante podrá manejar todos los pasos indispensables en la comunicación en ambos extremos como:

Apertura de un punto terminal en un canal (sockets)

Nombre de un socket (Bind)

Disponibilidad para recibir peticiones de servicio (Listen)

Petición de conexión (Connect)

Aceptación de una conexión (Accept)

Lectura o recepción de menajes de un socket

Escritura o envío de mensajes a un socket

Cierre de un canal (Close)

Conocimientos que pueden servir para esta practica están en el libro de UNIX: programación avanzada.

4.1.3 Practica III

4.1.3.1 ***Enunciado.*** Aplicación cliente/servidor. Familia de protocolos AF_UNIX. Orientado a la conexión. TCP. Donde el servidor luego de establecer la conexión envíe una cadena "Hola CUTB" al socket cliente .

4.1.3.2 Descripcion. De acuerdo con la forma de esta practica observamos que es similar a la practica anterior, sin embargo, en el fondo difieren grandemente***.*** Aquí le decimos por qué :

Primero, el tipo de socket se debe declarar como AF_UNIX lo cual cambia el manejo que se debe dar a la comunicación. Los sockets del tipo AF_UNIX son para comunicación entre procesos en el interior de una misma maquina. Aquí no se hace necesario utilizar una estructura de red (direcciones de red, puertos, etc) situacion que si se debia tener en cuenta en la anterior practica porque el tipo de socket era AF_INET.

El manejo del proceso de comunicación tambien difiere del anterior, es un poco mas simple, lleva menos pasos (instrucciones) establecer una correcta comunicación entre dos puntos terminales (sockets).

Segundo, el protocolo a utilizar cambia, ahora es UDP pues sugerimos que se trabajara en esta oportunidad con SOCK_DGRAM para una comunicación entre sockets orientada a la no conexión.

El trabajar con estos dos tipos de sockets y tipo de conexión le permite al estudiante abrir sus harizontes en cuanto al ,anejo de la filosofia Cliente/Servidor. Estara luego, la decision, en manos del programador, dependiendo de su gusto y necesidades, la forma de trabajar.

Conocimientos que pueden ser utiles para esta practica estan en el libro de Unix: programacion avanzada.

4.1.4 Practica IV

4.1.4.1 ***Enunciado.*** Dado un programa cliente/servidor con servidor del tipo interactivo, haga su equivalente en un tipo de servidor concurrente.

4.1.4.2 Descripcion. Existen dos tipos de servidores.

Interativos

Concurrentes

En esta practica se provee al estudiante de los codigos del servidor y del cliente de una aplicación. El servidor corresponde al tipo Interactivo. Lo que aquí se pide es que el estudiante transforme el codigo del servidor a uno del tipo concurrente, es decir, para que atienda mas solicitudes de servicio al mismo tiempo.

Para una mejor claridad de lo que aquí se plantea consultar la documentacion de los libros Sistemas distribuidos de Tanenbaum y Unix: programacion avanzada.

El estudiante debe manejar los conceptos de “hilos” pues cada servicio que se atienda de manera concurrente generara un nuevo proceso, el cual dependera de aquel proceso que lo llamo (conceptos fundamentales de sistemas operativos).

¿Que hace esta aplicación? Lo que ella hace es recibir un texto plano del lado del cliente, cifrarlo y enviar el ciphertexto al servidor, quien debera descifrarlo. Previamente el servidor y el cliente debieron acordar la clave con la cual trabajar.

El algoritmo de cifrado o encripcion corresponde al metodo de “Julio Cesar”. Para quien quiera saber mas al respecto puede consultar documentacion sobre ciptografia y seguridad en redes, aunque no es necesario pues el algoritmo de cifrado ya esta codificado en la aplicación que se le entrego al estudiante.

4.1.5 Practica v

4.1.5.1 Enunciado. Analizar, compilar y ejecutar la aplicación "hora", disponible en el servidor Linux, que como su nombre lo indica solicita la hora a una máquina remota. Anote sus experiencias y resultados.

4.1.5.2 Descripcion. En esta practica el estudiante debe hace algo similar a lo que utilizo en la practica II, sin embargo, en esta practica no es necesario que la aplicación pida la direccion IP de la maquina, sino, que el programa mismo debe interactuar con el sistema operativo y solicitar dicha informacion**.**

Lo anterior es una forma diferente de manejar las estructuras de direcciones en la programacion de este tipo de aplicaciones.

No esta demas decir que si estamos hablando de estructura de direcciones el socket debe ser del tipo AF_INET, lo cual afianza el conocimiento del estudiante en el manejo de este tipo de sockets.

Conocimientos que pueden servir para esta practica estan en el libro de Unix: programacion avanzada.

4.2 PRACTICAS RPC

4.2.1 Practica I

4.2.1.1 ***Enunciado***. Realizar una aplicación RPC que diga el número de usuarios conectados en una máquina remota, usando la función rnusers o en su defecto su equivalente a través de una llamada con callrpc.

4.2.1.2 Descripcion. Esta practica pertenece al primer nivel de programación de RPC, el nivel alto, donde no es necesario que el estudiante conozca a profundidad todo el funcionamiento de RPC, solo necesita conocer una instrucción rpc, sus parámetros y colocar esta instrucción en medio de un código de lenguaje C para que pueda operar.

El caso que nos ocupa ahora es el de la instrucción **rnusers**, que sirve para preguntar a la maquina por el numero de usuarios conectados al sistema.

Para poder funcionar rnusers necesita que est habilitado el servicio.

Como la plataforma de desarrollo de nuestro trabajo es LINUX, al inicio de la creación de estas practicas trabajamos con la Linux Red Hat 5.0, la cual no poseía ese servicio activo y no tenia forma de activarlo fácilmente. Por tanto, debimos buscar en Internet un conjunto de librerías, seleccionar la que necesitábamos, compilarlas y modificar un archivo de configuración de Linux, como es el caso de /etc/inetd (para mas detalle consultar el libro de comunicaciones en Unix). Sin embargo, la ultima

versión de Linux con la que trabajamos en la universidad fue la Red hat 6.0 que ya permite configurar automáticamente el servicio RPC.

Existe otra forma de conseguir el mismo resultado que se obtiene con rnusers y es usar la función callrpc con los parámetros adecuados.(Consultar libro de Unix).

4.2.2 Practica II

4.2.2.1 Enunciado. Realizar una aplicación RPC, del primer nivel de programación, que permita saber el puerto asociado a un servicio RPC activo en una maquina remota.

4.2.2.2 Descripcion. Esta practica tiene por objetivo introducir al estudiante en la programación en RPC, mas exactamente el estudiante debe utilizar la función rpcport, pero también debe hacer uso de otras herramientas que ofrece el sistema, como es el caso de rpcinfo**.**

La idea de esta practica es que se pueda conocer el puerto asociado a un servicio RPC activo en una maquina, pero para saber cuales servicios están activos se hace necesario usar la función rpcinfo. Entonces cuando ejecute la aplicación debe solicitar por teclado el numero asociado al servicio (esta información nos la brinda rpcinfo) y la aplicación debe estar en capacidad de decirnos el puerto con el que se atiende el servicio seleccionado.

Con rpcinfo se puede solicitar información de los servicios RPC de la maquina local a una maquina remota.(para mayor información consultar el libro de comunicación en Unix).

4.2.3 Practica III

4.2.3.1 ***Enunciado.*** Realizar una aplicación RPC, que posea:

- Un servidor que registre una función RPC (registerrpc) y ponga a disposición dicha función (svc_run)
- Un cliente que haga el llamado a la función (callrpc) y le transfiera el correspondiente parámetro.

Para hacer más práctico y uniforme el ejemplo se sugiere que la función halle el factorial de un número.

4.2.3.2 ***Descripcion.*** Esta practica pretende introducir al estudiante en un nivel más avanzado de la programación en RPC (nivel intermedio).

Aquí el estudiante ya debe manejar los conceptos de registro, activación y utilización de los servicios disponibles. En las practicas anteriores el estudiante solo se preocupaba de la utilización de los servicios disponibles en una maquina.

El objetivo de esta practica se logra básicamente con tres instrucciones (dos en el servidor y una en el cliente), estas son :

En el servidor :

Registrerrpc : Esta instrucción registra un servicio, el cual lo identifica a través de el envío de parámetros a esta función. Por ejemplo el numero con el cual se identificara el servicio.

Suc_run: Una vez registrado el servicio hay que ponerlo a disposición de los usuarios. Esto se logra con esta instrucción.

En el cliente :

Callrpc: Con esta función y al pasarle los parámetros que identifican al servicio que acabamos de registrar, podemos utilizar ese servicio.

Para un mejor criterio de evaluación se sugiere que todos los estudiantes trabajen sobre el mismo servicio. Realizar un servidor que reciba un entero y devuelva al cliente el factorial de dicho numero.

4.2.4 Practica IV

4.2.4.1 Enunciado. Realizar una aplicación RPC para calcular una la solución de una ecuación de segundo grado. El cliente deberá tomar los coeficientes de la ecuación y transmitirlos como parámetros al servidor, donde se llevaran acabo los cálculos**.**

4.2.4.2 Descripcion. Esta practica también pertenece también al nivel intermedio de programación en RPC.

El estudiante debe manejar los conceptos de registro, activación y utilización de los servicios disponibles en un sistema.

La idea de esta practica es que el estudiante registre y active completamente un servicio RPC, a la vez que el mismo establece como hacer la transferencia de parámetros entre las aplicaciones cliente y servidor.

Para lograr el objetivo de esta practica el estudiante debe :

- Definir la forma de paso de parámetros entre sus aplicaciones.
- Definir los módulos o servicios que quiera brindar el programador, que en este caso solo va ser uno. Un servicio para hallar la solución de una ecuación de segundo grado, basándose en los coeficientes de la ecuación que provienen del cliente.

- Crear un cliente que capture los coeficientes y los transfiera al servidor.
- Crear un servidor que realice el calculo.
- Lógicamente también se debe registrar y activar el servicio.

Para tener uniformidad e igualdad de condiciones a la hora de evaluar los resultados, se sugiere que todos los estudiantes trabajen en el caso ya expuesto.

Para profundizar y aclarar dudas de cómo realizar esto, se recomienda consultar el libro de comunicaciones en Unix.

4.2.5 Practica v

4.2.5.1 Enunciado. Analizar, compilar y ejecutar la aplicación "hora", disponible en el servidor Linux, que como su nombre lo indica solicita la hora a una máquina remota. Anote sus experiencias y resultados.

4.2.5.2 Descripcion. Esta practica pretende que el estudiante observe un ejemplo de programación en RPC del nivel mas bajo. Aquí el programador debe poseer amplios conocimientos con relación a registro de servicios, manejo de sockets entre las aplicaciones, etc.

En este nivel de programación, en ultimas, se realiza lo mismo que en otros niveles, pero descendiendo a lo elemental, aquí ya no hay instrucciones como rnusers que lo hagan todo por el programador, sin que al programador le toca hacer todo.

En esta practica el estudiante podrá mirar una forma diferente de generar código automático par su aplicación, a través de la utilización de un compilador de RPC (RPCGEN). De este modo el estudiante podrá seguir todo el proceso y analizar los resultados.

Sin embargo si el estudiante así lo quiere puede generar el código y hacer realizar todo el proceso que nosotros hicimos para llegar a la generación de esta practica o realizar la lectura de archivo **léeme** que acompaña a la practica y correr la aplicación final y anotar sus conclusiones.

4.3 PRACTICA DE COMUNICACIÓN EN GRUPO

4.3.1 Practica

4.3.1.1 ***Enunciado.*** Analice, contraste con la teoría de comunicación en grupo y elabore conclusiones al respecto del código del programa matriz.c, el cual se muestra en las Guías correspondientes.

4.3.1.2 Objetivos. La presente practica le servirá al estudiante para conocer y comprender la forma como se trabaja con varios procesos a la vez, lo que implica la comunicación y sincronización entre sí mediante herramientas del sistema LINUX.

Al finalizar este trabajo, los estudiantes habrán adquirido experiencia en programación de aplicaciones en entorno LINUX, utilizando estas herramientas :

- Funciones para ejecutar y sincronizar procesos. (FORK, WAIT Y EXIT).
- Funciones de ejecución de programas concurrentes (Familia de funciones EXEC..)
- Servicios IPC de Linux (Semáforos, memoria compartida).

4.3.1.3 ***Justificacion.*** La importancia de esta practica, no es la codificación del proceso de multiplicación de matrices, sino la manera como trabajan los procesos que se generan en la consecución del cálculo matricial.

Este programa sirve también como un ejemplo de paralelismo y es un buen ejercicio de sincronismo y comunicación en grupo.

4.3.1.4 Descripcion general. El programa matriz.c realiza la multiplicación de matrices en paralelo.

Este programa ejemplifica la comunicación en grupo, a través de un simple cálculo matricial.

La comunicación en grupos queda de manifiesto al observar la manera como interactuan los distintos procesos que se generan para lograr el objetivo primordial.

4.3.1.5 ***Descripcion del funcionamiento.*** El programa principal se encarga de leer dos matrices y comprueba si estas pueden multiplicarse entre sí. Posteriormente se generan tantos procesos como se le haya indicado en la línea de ordenes para multiplicar las matrices. Cada proceso creado, se va a encargar de generar una fila de las matriz producto, mientras queden filas por generar.

Las matrices que intervienen en la operación deben estar en memoria compartida.

Vamos a utilizar un semáforo para controlar la fila que debe generar cada proceso. Este semáforo que se inicializa con el total de las filas de la matriz producto y se va decrementando por cada fila generada.

El proceso principal se queda esperando a que todos los demás terminen para presentar el resultado.

4.3.1.6 Descripcion detallada.

Programa : Matriz.c

Descripción : Programa que multiplica matrices en paralelo.

include <stdio.h>

include <stdlib.h>

include <sys/type.h>

include <stdio.h>.....

- Definición de matriz
    - Identificar zona de memoria compartida, donde va a estar la matriz.
    - Definir coeficientes de la matriz
- Función : crear_matriz

  Descripción : Función que crea la memoria compartida.
    - Petición de memoria compartida.
    - Inicialización de la matriz
        - Dar formato a la memoria, para poder direccionar los coeficientes .
- Función : leer_matriz

  Descripción : lee una matriz del fichero estandar de entrada.
- Función : Multiplicar_matrices

  Descripción : Multiplica dos matrices y crea una nueva para el resultado.

  El trabajo se distribuye entre el total de procesos que indique **numproc.**

- Creación de dos semáforos. Uno de ellos se inicializa con el total de filas de la matriz producto.
- Creación de tantos procesos como indique **numproc** .
    - Código para los procesos hijos (fork).
- Realizar operaciones sobre los semáforos
- Calcular la fila i-ésima de la matriz producto.
- Esperar a que terminen los procesos.
- Borrar semáforo.

- Función : destruir_matriz

  Descripción : Función encargada de liberar una zona de la memoria compartida.

- Función : imprimir_matriz

  Descripción : Esta función presenta una matriz en el fichero estándar de salida.

- Función Principal
    - Definir variables locales.
    - Leer matrices.
    - Procesamiento de las matrices.

4.3.1.7 Plan de trabajo.

- Comprenda claramente en qué consiste la practica.
- Domine el funcionamiento de los IPC de Linux (Memoria compartida, Semáforos).
- Entienda el problema de sincronización de procesos que se presenta.

- Conozca y domine los mecanismos para lanzar procesos en Linux (Unix).
- Ejecute el programa, realice varias pruebas, analice, confróntelo con sus conocimientos teóricos y saque sus conclusiones al respecto.

4.3.1.8 Compilacion. cc -o mmatriz matriz.c

4.3.1.9 Ejecucion. / mmatriz

.

4.4 GRUPO DE PRACTICAS PVM

4.4.1 Practica I

4.4.1.1 Enunciado. Desarrollar una aplicación con PVM conformada por dos pequeños programas, entre los cuales se establezca una transferencia de mensajes.

Mensaje sugerido : “Hola mundo”.

El mensaje debe ir acompañado por :

- El nombre de la máquina que envía el mensaje.
- TID (task identificator) o identificador de tarea.

4.4.1.2 ***Objetivos.*** La anterior practica introducir al estudiante en lo referente a la programación paralela con PVM. Por medio de esta el estudiante podrá analizar y comprender la forma cómo se realiza en PVM, la creación de una tarea y el paso de mensajes entre estas.

Al finalizar el desarrollo de esta practica el estudiante habrá logrado iniciarse en la programación en PVM, afianzando y fortaleciendo sus conocimientos teóricos al respecto.

4.4.1.3 ***Justificación.*** Es necesario que el estudiante inicie su experimentación en la programación con PVM desde lo más sencillo, y no por ello menos importante, como lo es la creación de tareas y el paso de mensajes.

4.4.1.4 Descripcion general. Hola y Hola_otro**, son dos programas que cooperan entre sí.**

Ellos establecen una transfeencia de mensajes, en este caso una cadena de caracteres ("Hola Mundo"). Este mensaje debe poseer además de la cadena de caracteres ya mencionada, el nombre de la máquina que envía el mensaje y el identificador de tarea.

Al ejecutar este programa se aprecia la forms como se crea una nueva tarea y su respectivo paso de mensajes.

Ambos programas se compilan al tiempo, pero al momento de ejecutar, se ejecuta **Hola** inicialmente, este intenta establecer una comunicación con **hola_otro** y si no ocurre ningún error, se ejecuta dicho programa, comenzando así la transferencia de mensajes.

4.4.1.5 Plan de trabajo. Para poder tener éxito en el desarrollo de la practica es necesario :

- Entienda claramente lo que propone la practica.
- Comprenda el concepto general de la programación paralela.
- Domine los conceptos de Task (Tareas), TIDs , canal y paso de mensajes en PVM.

- Estudie y analice la manera como se crea en PVM una tarea y el tratamiento que esta le da a la memoria local y al conjunto de puertos que definen la interfaz a su ambiente.
- Diseñe los algoritmos respectivos, analícelos, codifíquelos, realice pruebas, estudiar resultados.

CONSEJOS TECNICOS

- Archivos fuente :

  hola.c hola_otro.c

- Compilación :

  %cc –o hola hola.c

  %cc –o hola_otro hola_otro.c

- Ejecución desde Shell :

  % hola

- Ejecución desde la consola de PVM :

  PVM > SPAWN -> hola

Ejemplo de la salida

  Tarea t40002

    Desde t40003 : hola.mundo .....

4.4.2 Practica II

4.4.2.1 ***Enunciado.*** Crear una aplicación del tipo Maestro / Esclavo, donde el proceso maestro cree y dirija algún número de procesos esclavos que cooperen para hacer un trabajo.

Trabajo sugerido : Creación de tres tareas que intercambien mensajes entre sí y su resultado sea visualizado por pantalla.

4.4.2.2 Objetivo. A través de este trabajo, el estudiante conocerá el concepto de Maestro / Esclavo y se entrenará en la creación de dichos procesos.

El estudiante dominará la manera como los procesos maestros crean y dirigen procesos esclavos, su forma de interactuar, etc.

4.4.2.3 Justificación. En la programación paralela con el fin de crear un sistema distribuido, conocer la forma de coordinar varios procesos esclavos a través de un proceso maestro es de vital importancia.

4.4.2.4 Descripcion general. Master1 y slave1**, son dos procesos, en donde el primero es el proceso maestro y el segundo el proceso esclavo.**

El proceso maestro crea o genera varios procesos esclavos, los cuales interactuan entre sí para la consecución del trabajo propuesto.

Estos procesos esclavos intercambian mensajes entre sí y su resultado es visualizado por pantalla, todo lo anterior con dirección y coordinación del maestro.

4.4.2.5 ***Plan de trabajo.*** Con el fin de obtener buenos resultados, recomendamos :

- Leer detenidamente el enunciado, analice y comprenda lo que se desea realizar.
- Estudiar, analizar y dominar los siguientes conceptos :
    - ¿Que es un maestro ?
    - Características.
    - Forma de operar e interactuar.
- Entender el problema de coordinación de procesos que se presenta.
- Diseñar los Algoritmos y rutinas correspondientes.
- Efectuar pruebas.
- Analizar resultados.

4.4.2.6 Consejos tecnicos.

- Archivos fuente :

    Master1.c Slave1.c

- Compilación :

    % cc -o master1 master1.c

    % cc –o slave1 slave1.c

- Ejecución desde Shell :

    % master1

- Ejecución desde consola de PVM :

    PVM > SPAWN - > master1

- Ejemplo de salida

  Intercambiando 3 tareas ... Exitoso

  recibido 100.000000 desde 1; (esperando 100.000000)

  recibido 200.000000 desde 0; (esperando 200.000000)

  recibido 300.000000 desde 2; (esperando 300.000000)

4.4.3 Practica III

4.4.3.1 Enunciado. Realizar un ejemplo de SPMD (Single program Multiple data) que utlice pvm_siblings, par determinar el número de tareas y los TIDs que resultaron.

La aplicación debe utilizar un simple token ring y paso de mensajes.

4.4.3.2 ***Objetivo.*** Lograr en el estudiante el aprendizaje y posterior dominio del modelo SPMD.

- Que el estudiante conozca la forma de operar de pvm_siblings.
- Entrenar un poco más al estudiante en la forma de trabajar con la filosofía Token Ring y el paso de mensajes en PVM.

4.4.3.3 ***Justificación.*** Es necesario que el estudiante maneje de forma práctica el modelo SPMD utilizado por grandes computadoras en cómputos donde se hace necesario la repetición de cálculos en varios conjuntos de datos.

PVM se convierte en una opción económica y viable para realizar la simulación de dichos sistemas. Además la forma de trabajar con token ring y paso de mensajes es muy utilizado en la sincronización de los sistemas distribuidos.

4.4.3.4 Descripcion.

Esta practica esta basada en el modelo SPMD y hace uso de PVM_SIBLINGS que es una librería que posee PVM para determinar el número de tareas que se generan.

Esta practica utiliza Token Ring para el paso de mensajes entre las tareas.

La aplicación muestra por pantalla el paso del token a través de las tareas utilizando los respectivos TIDs (identificadores de tareas).

4.4.3.5 Plan de trabajo

- Conocer, estudiar y analizar el concepto de Modelo SPMD.
- Dominar el concepto y empleo de PVM_siblings.
- Entrenarse en la forma de omplementar estos conceptos y herramientas mencionados.
- Conocer y dominar el concepto de token ring y paso de mensajes, así como su implementación en PVM.
- Diseñar el algoritmo.
- Realizar pruebas
- Analizar resultados.

4.4.3.6 Consejos tecnicos.

Este programa sólo puede ser ejecutado desde la consola de PVM.

- Archivo fuente :

  spmd.c

- Compilación :

% cc –o spmd spmd.c

- Ejecución :

PVM > SPAWN -4 - > spmd

- Ejemplo de la salida

[4:t4000f] Paso un token a traves de los 4 id de tareas:

[4:t4000f] 262159 -> 262160 -> 262161 -> 262162 -> 262159

[4:t4000f] token ring hecho

4.4.4 Practica IV

4.4.4.1 ***Enunciado.*** Desarrolle una aplicación que ilustre como medir el ancho de banda de una red, a través del conjunto de librerías que ofrece PVM.

4.4.4.2 ***Objetivo.*** Al realizar esta practica el estudiante conocerá y se adiestrará en el desarrollo de aplicaciones de red con PVM.

Lograr que el estudiante domine la utilización de las librerías que ofrece PVM y por lo tanto amplíe su conocimiento en el desarrollo de aplicaciones de esta índole.

4.4.4.3 Justificación. Resulta interesante el uso de PVM en la construcción de aplicaciones de red y aún cuando esta se usan para obtener información de el funcionamiento de la misma***.*** Además es una forma de adiestrarse en la programación con pvm.

4.4.4.4 ***Descripcion.*** Esta practica esta conformada por dos rutinas, timing y timing_slave, el primero es el maestro y el siguiente el esclavo.

El maestro (timing) es quien dirige la operación y realiza llamados al esclavo para realizar tareas necesarias para conseguir el objetivo.

Estas rutinas, para calcular el ancho de banda de la red, hacen uso de las librerías que ofrece PVM para la consecución de dicha información.

4.4.4.5 Plan de trabajo

- Conozca y domine el concepto de ancho de banda de una red y como puede ser calculado.
- Determine las librerías de PVM que se utilizan en la obtención de dicha información.
- Entrenarse en el uso de estas librerías.
- Diseñar el algoritmo.
- Realizar pruebas
- Codificar y analizar resultados.

4.4.4.6 Consejos tecnicos

Archivos fuentes :

timing.c timing_slave.c

Compilación :

% cc -o timing timing.c

% cc –o timing_slave timing_slave.c

Ejecución :

Desde shell

% timing

Desde Consola PVM

PVM> SPAWN - > timing.

5 GUÍA DEL ESTUDIANTE.

5.1 COMUNICACIÓN ENTRE PROCESOS

5.1.1 Cliente/servidor

5.1.1.1 Conceptos y definiciones del modelo c/s: protocolos y conexiones.

5.1.1.1.1 Breve orientación teórica. La comunicación mediante sockets es una interfaz con la capa de transporte (nivel 4) de la jerarquía OSI.

Sin embargo, la interfaz de acceso a la capa de transporte del sistema UNIX no está totalmente aislada de las capas inferiores, por lo que a la hora de trabajar con sockets es necesario conocer algunos detalles sobre esas capas. En concreto, a la hora de establecer una conexión mediante sockets, es necesario conocer la familia o dominio de la conexión y el tipo de conexión.

- Una familia agrupa todos aquellos sockets que comparten características comunes, tales como protocolos, convenios para formar direcciones de red, convenios para formar nombre, etc.
- El tipo de conexión indica el tipo de circuito que se va a establecer entre los dos procesos que se están comunicando. El circuito puede ser virtual (orientado a la conexión) o datagrama (no orientado a conexión). Para establecer un circuito virtual, se realiza una búsqueda de enlaces libres que unan los computadores a conectar. Una vez establecida la conexión, se puede proceder al envío secuencial de los datos, ya que la conexión es permanente. Por el contrario, los datagramas no trabajan con conexiones permanentes. La transmisión por los datagramas es a nivel de paquetes, donde cada paquete puede seguir una ruta distinta y no se garantiza una recepción secuencial de la información.

5.1.1.1.2 Preguntas de concepto.

a. ¿Qué es un programa servidor?
b. ¿Qué es un programa cliente?
c. ¿Qué es una familia de sockets?. Mencione algunas de las más conocidas.
d. ¿A qué nos referimos al hablar de "tipo de conexión" a la hora de establecer una comunicación entre 2 procesos (sockets)?

5.1.1.1.3 Preguntas de análisis.

a. Los servicios y protocolos de red se relacionan directamente con el modelo de comunicación, con base en capas, entre computadores. ¿Para su uso (servicios y protocolos de red) se hace necesario conocer los detalles de funcionamiento de cada una de las capas del modelo de comunicación?. Explique.

5.1.1.2 Generalidades del manejo de sockets y esquema de funcionamiento de la filosofía cliente/servidor

5.1.1.2.1 Breve orientación teórica. A la hora de referirse a un nodo de la red, cada protocolo implementa un mecanismo de direccionamiento. La dirección distingue de forma inequívoca a cada nodo o computador y es utilizada para encaminar los datos desde el nodo origen al nodo destino. La forma de construir direcciones depende de los protocolos que se empleen en la capa de transporte y de red. Hay muchas llamadas al sistema UNIX que necesitan un puntero a una estructura de dirección de socket para trabajar. Esta estructura se define el fichero de cabecera <sys/socket.h> y su forma es la siguiente:

```
struct sockaddr {
  u_short sa_family;  /*Familia de sockets. Se emplean las constantes de la forma
                        AF_***  */
  Char sa_data[14];    /*14 bytes que contiene la dirección. Su significado depende de
                        la familia de sockets que se esté empleando */
};
```

Si usamos una familia que emplea protocolos Internet, la forma de las direcciones de red será la definida en el fichero de cabecera <netined/in.h>:

```
struct in_addr {
```

```
  u_long s_addr;  /*32 bits que contienen la identificación de la red y del host. */
};

struct sockaddr_in {
  short sin_family;   /*AF_INET */
  u_short sin_port;   /*16 bits con el número de puerto */
  struct in_addr sin_addr;  /*32 bits con la identificación de la red y del host. */
  char sin_zero[8];  /*8 bytes no usados.*/
};
```

La familia de sockets conocida como UNIX domain emplea direcciones con la forma definida en <sys/un.h>:

```
struct sokaddr_un {
  short sun_family;  /* AF_UNIX */
  char sun_path[108];  /* Path name. */
};
```

Estas direcciones se corresponden en realidad con path names de ficheros y su longitud (110 bytes) es superior a los 16 bytes que de forma estándar tienen las direcciones del resto de familias. Esto es posible debido a que esta familia de sockets se utiliza para comunicar procesos que se están ejecutando bajo el control de una misma máquina, por lo que no necesitan hacer accesos a la red.

Por otra parte, el modelo Cliente/Servidor es el modelo estándar de ejecución de aplicaciones en una red.

Un servidor es un proceso que se está ejecutando en un nodo de la red y que gestiona el acceso a un determinado recurso. Un cliente es un proceso que se ejecuta en el mismo o en diferente nodo y que realiza peticiones de servicio al servidor. Las peticiones están originadas por la necesidad de acceder al recurso que gestiona el servidor.

Las acciones que debe llevar a cabo el programa servidor son las siguientes:

5. Abrir el canal de comunicaciones e informar a la red tanto de la dirección a la que responderá como de su disposición para aceptar peticiones de servicio.
6. Esperar a que un cliente le pida servicio en la dirección que él tiene declarada.
7. Cuando recibe una petición de servicio, si es un servidor interactivo atenderá al cliente. Los servidores interactivos se suelen implementar cuando la respuesta que necesita el cliente es sencilla e implica poco tiempo de proceso. Si el servidor es concurrente creará un proceso mediante fork para que le dé servicio al cliente.
8. Volver al punto 2 para esperar nuevas peticiones de servicio.

El programa cliente, por su parte, llevará a cabo las siguientes acciones:

4. Abrir el canal de comunicaciones y conectarse a la dirección de red atendida por el servidor.
5. Enviar al servidor un mensaje de petición de servicio y esperar hasta recibir la respuesta.
6. Cerrar el canal de comunicaciones y terminar la ejecución.

5.1.1.2.2 Preguntas de concepto

a. ¿Para qué sirve la dirección de red de un host?
b. ¿Existe una sola forma para las direcciones de red?. ¿De qué depende la forma de una dirección de red?
c. ¿Cuál es la estructura de una dirección de socket para una familia que emplea protocolos Internet?
d. Muestre de manera gráfica el esquema genérico de flujo de control para los procesos servidores y clientes. Explique

5.1.1.2.3 Preguntas de análisis.

a. De un ejemplo práctico de cómo se haría el llenado de una estructura de dirección de un socket para la familia AF_INET y orientado a conexión.
b. En un socket no orientado a la conexión ¿se hace necesaria la llamada a la instrucción *connect* del lado del proceso cliente?. Explique su respuesta.

5.1.1.3 Métodos de direccionamiento

5.1.1.3.1 Breve orientación teórica. Para que un cliente pueda enviar un mensaje a un servidor, debe conocer la dirección de éste.

Si sólo existe un proceso en ejecución en la máquina destino, el núcleo sabrá que hacer con el mensaje recibido (dárselo al único proceso en ejecución). Sin embargo, ¿qué ocurre si existen varios procesos en ejecución en la máquina destino?

Otro tipo de sistema de direccionamiento envía mensajes a los procesos en vez de a las máquinas. Un esquema común consiste en utilizar nombres con dos partes, para especificar tanto la máquina como el proceso.

Una ligera variación de este esquema de direccionamiento utiliza machine. Local–id en vez de machine.process. El campo local–id es por lo general, un entero aleatorio de 16 o 32 bits (o el siguiente de una serie).

Existe otro método más para la asignación de identificadores a los procesos, el cual consiste en dejar que cada proceso elija el propio identificador de un gran espacio de direcciones dispersas, como el espacio de enteros binarios de 64 bits. Sin embargo, aquí también existe un problema: ¿Cómo sabe el núcleo emisor a cuál máquina enviar el mensaje? En una LAN que soporte transmisiones, el emisor puede transmitir un **paquete especial de localización** con la dirección del proceso destino.

Aunque este esquema es transparente, incluso con ocultamiento, la transmisión provoca una carga adicional en el sistema. Esta carga se evita mediante una máquina adicional para la asociación a alto nivel (es decir, en ASCII) de los nombres de

servicios con las direcciones de las máquinas. Lo anterior se conoce a menudo como **servidor de nombres.**

Un método por completo distinto utiliza un hardware especial. Se deja que los procesos elijan su dirección en forma aleatoria.

5.1.1.3.2 Preguntas de concepto.

a. ¿Cuál es el método más simple de direccionamiento que permite el envío de solicitudes en forma inequívoca a un proceso X dentro de una máquina Y?
b. ¿ En qué consiste el método de direccionamiento de procesos con transmisión?
c. Existe un método de direccionamiento que se basa en un hardware especial. Explique.

5.1.1.3.3 Preguntas de análisis.

a. ¿El método de direccionamiento machine.process es ideal para la construcción de un sistema distribuido? Explique.
b. ¿Cuál es la principal ventaja del método de búsqueda de direccionamiento por medio de un servidor de nombres respecto al método de direccionamiento de procesos con transmisión?

5.1.1.3.4 Práctica.

5.1.1.3.4.1 *Práctica I.* Desarrollar una aplicación que realice las cuatro operaciones básicas (suma, resta, multiplicación y división),usando socket del tipo AF_INET y protocolo TCP/IP.

Laaplicación debe estar compuesta por un cliente, responsable de tomar la expresión a evaluar y enviarla al servidor y de un servidor, quien hará el cálculo y enviará la respuesta.

5.1.1.3.4.1.1 Justificación. El estudiante debe ver la importancia de conocer y saber manejar los sockets del tipo AF_INET y el protocolo TCP/IP, los cuales son empleados en la implementaciones de muchas aplicaciones .

5.1.1.3.4.1.2 Objetivos

- Con esta práctica se pretende que el estudiante maneje una de las formas de trabajo de la filosofía cliente/serrvidor.
- Establecer una eficaz comunicación entre los sockets en ambos extremos.
- Lograr que el estudiante se adiestre en el desarrollo de aplicaciones con sockets del tipo AF_INET y observe sus características.

5.1.1.4 LLamadas para el manejo de sockets.

5.1.1.4.1 Breve orientación teórica.

Apertura de un punto terminal en un canal. Socket:

La llamada para abrir un canal bidireccional de comunicaciones es ***socket,*** y se usa de la siguiente manera:

socket (af, types, protocol);

Socket crea un punto terminal para conectarse a un canal y devuelve un descriptor. El descriptor de socket devuelto se usará en llamadas posteriores a funciones de la interfaz.

Af (address family) especifica la familia de sockets o familia de direcciones que se desea emplear. Las distintas familias están definidas en el fichero de cabecera <sys/socket.h>. Las dos familias siguientes suelen estar presentes en todos los sistemas:

AF_UNIX Protocolos internos UNIX. Es la familia de sockets empleada para comunicar procesos que se ejecutan en una misma máquina. Esta familia no requiere que esté presente un hardware especial de red

AF_INET Protocolos Internet. Es la familia de sockets que se comunican mediante protocolos, tales como TCP o UDP.

El argumento types indica la semántica de la comunicación para el socket. Puede ser:

SOCK_STREAM socket con un protocolo orientado a conexión.

SOCK_DGRAM socket con un protocolo no orientado a conexión o datagrama.

Protocol especifica el protocolo particular que se va a usar en el socket.

Nombre de un socket. Bind

La llamada bind se utiliza para unir un socket a una dirección de red determinada. Se usa de la siguiente manera:

bind (sfd, addr, addrlen);

Cuando se crea un socket con la llamada socket, se le asigna una familia de direcciones, pero no una dirección particular. Bind hace que el socket de descriptor sfd se una a la dirección de socket específica en la estructura apuntada por addr. Addrlen indica el tamaño de la dirección.

Disponibilidad para recibir peticiones de servicio. Listen

Cuando se abre un socket orientado a conexión, el programa servidor indica que está disponible para recibir peticiones de conexión mediante la llamada a listen. La cual se emplea de la siguiente forma:

listen (sfd, backlog)

Listen habilita una cola asociada al socket descrito por sfd. Esta cola se va a encargar de alojar peticiones de conexión procedentes de los procesos clientes. La longitud de

esta cola es la especificada en el argumento backlog. Para que la llamada a listen tenga sentido, el socket debe ser del tipo SOCK_STREAM (socket orientado a conexión).

Petición de conexión. Connect.

Para que un proceso cliente inicie una conexión con un servidor a través de un socket, es necesario que haga una llamada a connect. Esta función se declara de la forma siguiente:

connect (sfd, addr, addrlen);

sfd es el descriptor del socket que da acceso al canal y addr es un puntero a una estructura que contiene la dirección del socket remoto al que queremos conectarnos. Addrlen es el tamaño en bytes. La estructura de la dirección dependerá de la familia de sockets con la que estemos trabajando.

Aceptación de una conexión. Accept

Los procesos servidores van a leer peticiones de servicios mediante la llamada accept. Llamada:

accept (sfd, addr, addrlen)

Esta llamada se usa con sockets orientados a conexión como el tipo SOCK_STREAM. El argumento sfd es un descriptor de socket creado por una

llamada previa a socket y unido a una dirección mediante bind. Accept extrae la primera petición de conexión que hay en la cola de peticiones pendientes creada con una llamada previa a listen. Una vez extraída la petición de conexión, accept crea un nuevo socket con las mismas propiedades que sfd y reserva un nuevo descriptor de fichero (nsfd) para él.

El socket original (sfd) permanece abierto y puede aceptar nuevas conexiones; sin embargo, el socket recién creado (nsfd) no puede usarse para aceptar más conexiones.

El argumento addr debe apuntar a una estructura local de dirección de socket. La llamada accept rellenará esa estructura con la dirección del socket remoto que pide la conexión. El argumento addrlen debe ser un puntero a int. Inicialmente, debe contener el tamaño en bytes de la estructura de dirección. La función sobreescribirá en addrlen el tamaño real de la dirección leída de la cola de direcciones.

Lectura o recepción de mensajes de un socket.

Una vez que el canal de comunicación entre los procesos servidor y cliente está correctamente inicializado y ambos procesos disponen de un conector (socket) con el canal, contamos con cinco llamadas al sistema para leer datos/mensajes de un socket y otras cinco llamadas para escribir datos/mensajes en él.

Las llamadas para leer datos de un socket son: read, readv, recv, recvfrom, recvmsg.

Escritura o envío de mensaje a un socket.

Las llamadas para escribir datos en un socket son: write, writev, send, sento, sendmsg.

Cierre del canal. Close.

Una vez que un proceso no necesita realizar más accesos a un socket, puede desconectarse del mismo. Para ello, y aprovechando que un socket es tratado sintácticamente como si fuera un fichero, podemos usar la llamada close. Esta llamada va a cerrar el sockets en sus dos sentido (servidor-cliente y cliente-servidor).

5.1.1.4.2 Preguntas de concepto

a. En la primitiva socket(af, type, protocol), ¿qué sucede al hacer cero (0) el parámetro protocol?
b. Si se une un socket AF_INET a una dirección y el campo sin_port se inicializó en cero y se olvidó cambiar este valor, por un valor de puerto diferente, ¿qué sucede?.
c. Si el socket es del tipo SOCK_DGRAM, ¿*connect* especifica la dirección del socket remoto al que se le van a enviar los mensajes?. Explique.
d. Close( ) cierra un socket en sus dos sentidos (C/S y S/C) consulte una primitiva o instrucción que permita un control más fino a la hora de cerrar y deshabilitar uno de 2 sentidos del socket.

5.1.1.4.3 Preguntas de análisis

a. ¿Se vería afectada la normal ejecución de un servidor interactivo si a él llegan 2 o más solicitudes de servicio a la vez?

5.1.1.4.4 Práctica

5.1.1.4.4.1 Práctica II. Desarrollar una aplicación cliente/servidor donde el cliente capture y envíe una cadena “Hola mundo” al servidor y este a su vez la reciba y visualice. Familia AF_INET. Orientado a la conexión. TCP.

5.1.1.4.4.1.1 Justificación. Es necesario que el estudiante experimente el paso de mensajes a través de aplicaciones bajo la filosofía cliente/servidor, a la vez que se introduce en el manejo de primitivas básicas para la manipulación de sockets.

5.1.1.4.4.1.2 Objetivos. Que el estudiante:

- Cree un par de sockets (un cliente y un servidor).
- Aprenda a llenar las estructuras que hacen parte de cada uno de los sockets.(dirección introducida por teclado).
- Establezca una conexión entre los sockets.
- Utilice primitivas para envío y recepción de mensajes.
- Trabaje con la familia de protocolos AF_INET y con sockets orientados a la conexión.

5.1.1.4.4.2 *Práctica III.* Aplicación cliente/servidor. Familia de protocolos AF_UNIX. Orientado a la conexión. TCP. Donde el servidor luego de establecer la conexión envíe una cadena “Hola CUTB” al socket cliente .

5.1.1.4.4.2.1 Justificación. Es necesario que el estudiante experimente el paso de mensajes a través de aplicaciones bajo la filosofía cliente/servidor, a la vez que se introduce en el manejo de primitivas básicas para la manipulación de sockets.

5.1.1.4.4.2.2 Objetivos

- Que el estudiante cree un par de sockets (un cliente y un servidor).
- Que el estudiante aprenda a llenar las estructuras que hacen parte de cada uno de los sockets.(dirección introducida por teclado).
- Que el estudiante establezca una conexión entre los sockets.
- Que el estudiante utilice primitivas para envío y recepción de mensajes.
- Que el estudiante trabaje con la familia de protocolos AF_UNIX y con sockets orientados a la conexión.

5.1.1.5 Tipos de servidores

5.1.1.5.1 Breve orientación teórica. El modelo Cliente/Servidor es el modelo estándar de ejecución de aplicaciones en una red.

Un servidor es un proceso que se está ejecutando en un nodo de la red y que gestiona el acceso a un determinado recurso. Un cliente es un proceso que se ejecuta en el mismo o en diferente nodo y que realiza peticiones de servicio al servidor. Las peticiones están originadas por la necesidad de acceder al recurso que gestiona el servidor.

El servidor está continuamente esperando peticiones de servicio. Cuando se produce una petición, el servidor despierta y atiende al cliente. Cuando el servicio concluye, el servidor vuelve al estado de espera. De acuerdo con la forma de prestar el servicio, podemos considerar dos tipos de servidores:

- Servidores interactivos: El servidor no solo recoge la petición de servicio, sino que él mismo se encarga de atenderla.

- Servidores concurrentes: El servidor recoge cada una de las peticiones de servicio y crea otros procesos para que se encarguen de atenderlas. Este tipo de servidores solo es aplicables en sistemas multiprocesos, como es UNIX. La ventaja que tiene este tipo de servicio es que el servidor puede recoger peticiones a muy alta velocidad, porque está descargado de la tarea de atención del cliente.

5.1.1.5.2 Preguntas de concepto

a. ¿Cuáles son los tipos de servidores?. Explique sus diferencias.

5.1.1.5.3 Preguntas de análisis

a. ¿Cuándo se recomienda el uso de servidores interactivos?
b. Se requiere implantar una aplicación C/S en un sistema donde el tiempo de atención al cliente es variable. Se necesita de su opinión: ¿Qué tipo de servidor recomienda?

5.1.1.5.4 Práctica

5.1.1.5.5 Práctica IV. Dado un programa cliente/servidor con servidor del tipo interactivo, haga su equivalente en un tipo de servidor concurrente.

5.1.1.5.5.1.1 Justificación. Es necesario que el estudiante aprecie un ejemplo de cada uno de los tipos de servidores, los compare y confronte su análisis con lo aportado por la teoría. A la vez que el estudiante aprenderá aspectos básicos de los tipos de servidores de la filosofía cliente/servidor, este pondrá en practica una parte de lo referente al manejo de procesos (tema de vital importancia dentro de los sistemas operativos en general).

5.1.1.5.5.1.2 Objetivos

- Que el estudiante con base en la practica observe las principales ventajas y desventajas de cada uno de los tipos de servidores.
- Que aprenda la manera de construir y manipular procesos hijos a través de la instrucción *fork*, la creación de una cola de procesos, eliminación de procesos, etc.

Código de la aplicación

Código del servidor (servidor.c)

```
/*Servidor: Cifrador de Julio Cesar */

#include "i_addr.h"
#define BUFF_SIZE 200

struct sockaddr_in ser_addr, cli_addr;

void close_sockets();
void resetear(char *);
void init_buffer(char[]);

char *mensaje,*ciphertext;
char buffer[BUFF_SIZE];
char alfabeto[28]="abcdefghijklmn¤opqrstuvwxyz ",
     cipher[200],soda[1];
int i,puente;

/*Vbles para el manejo de sockets*/

int ok_bind,ok_conect,ok_read,ok_write,ok_list,ok_close,
    ok_sock,id_sock,nid_sock,dir_puerto,clave;
int ser_addr_len,cli_addr_len,msg_size;
char msg_buffer[BUFF_SIZE], dir_nodo[BUFF_SIZE];
```

```
int main (void){

 /*abrir socket*/
 id_sock=socket(AF_INET,SOCK_STREAM,0);
 if (id_sock==-1) {
   printf("\nError abriendo socket\n");
   exit(-1);
 }

 /*inicializar estructura de ser_addr*/
 init_buffer(buffer);
 system("clear");
 printf("Entre la direccion IP del servidor: ");
 gets(dir_nodo);
 printf("Entre el numero del puerto: ");
 gets(buffer);
 dir_puerto=atoi(buffer);

 printf("\nAplicacion servidora de Alg de cifrado de Julio Cesar");
 printf("\nEntre la clave para descifrar mensaje: "); gets(buffer);
 clave=atoi(buffer);

 ser_addr.sin_family=AF_INET;
 ser_addr.sin_addr.s_addr=inet_addr(dir_nodo);
 ser_addr.sin_port=htons(dir_puerto);
 ser_addr_len=sizeof(ser_addr);

 /*publiacion del socket*/
```

```
ok_bind=bind(id_sock,&ser_addr,ser_addr_len);
if (ok_bind==-1) {
  printf("\nError publicando el socket\n");
  exit(-1);
}

printf("\nSocket listo. Esperando solicitud del cliente");

/*Definicion del buzon para recibir conexiones*/
ok_list=listen(id_sock,0);
if(ok_list==-1) {
  printf("\nError haciendo listen\n");
  exit(-1);
}

/*aceptando la conexion con el cliente*/
cli_addr_len=sizeof(cli_addr);
nid_sock=accept(id_sock,&cli_addr,&cli_addr_len);
if (nid_sock==-1) {
  printf("\nError aceptando conexion\n");
  exit(-1);
}

mensaje=(char *)calloc(BUFF_SIZE,sizeof(char));
ciphertext=(char *)calloc(BUFF_SIZE,sizeof(char));

while(1) {
  /*inicializa buffer de entrada*/
```

```
init_buffer(msg_buffer);

/*lea solicitud del cliente*/
ok_read=read(nid_sock,msg_buffer,sizeof(msg_buffer));
if (ok_read==-1) {
  printf("\nError al leer el buffer en el servidor\n");
  exit(-1);
}
if (msg_buffer[0]!=NULL) {
    for (i=0;i<=200;i++)
      cipher[i]=NULL;
    for (i=0;i<strlen(msg_buffer);i++) {
      ciphertext[i]=cipher[i]=msg_buffer[i];
    }
    ciphertext[strlen(msg_buffer)]=cipher[strlen(msg_buffer)]=NULL;
}
else
  resetear(ciphertext);

if (msg_buffer[0]==NULL || msg_buffer[0]=='$') {
  /*finalizar servicio por solicitud del cliente*/
  printf("\nfin del servicio por solicitud del cliente\n");
  close_sockets();
  return 0;
}
else {
  init_buffer(mensaje);
  resetear(mensaje);
```

```
printf("\nCiphertext recibido desde el cliente: %s",msg_buffer);
for (i=0;ciphertext[i];i++) {
  soda[0]=cipher[i];
  puente=strcspn(alfabeto,&soda[0]);
  if (puente-clave < 0) {
    mensaje[i]=alfabeto[(strcspn(alfabeto,&soda[0])-clave+28)%28];
  }
  else {
    mensaje[i]=alfabeto[(strcspn(alfabeto,&soda[0])-clave)%28];
  }
}
printf("\nTexto plano: %s",mensaje);
printf("\nSolicitud del cliente atendida\n");

/*prepara el buffer*/
init_buffer(msg_buffer);
strcpy(msg_buffer,"Todo listo aca en el servidor");
 /*envia respuesta al cliente*/
ok_write=write(nid_sock,msg_buffer,sizeof(msg_buffer));
if (ok_write==-1) {
  printf("\nError al hacer write desde el servidor\n");
  exit(-1);
}

printf("Desea cambiar la clave (s/n) ??: ");
gets(buffer);
if (buffer[0]=='s') {
  printf("Entre la nueva clave : ");
```

```
        gets(buffer);
        clave=atoi(buffer);
      }

      printf("Esperando nueva solicitud...\n\n");
    }
  } /*fin del mq*/
} /*fin del main*/

void resetear(char *cadena) {
  int i;
  i=0;
  cadena=(char *)calloc(BUFF_SIZE,sizeof(char));
  for (i=0;i<BUFF_SIZE;i++) cadena[i]=NULL;
}

void init_buffer (char cadena[]) {
  int i;
  i=0;
  for (i=0;i<BUFF_SIZE;i++) cadena[i]=NULL;
}

void close_sockets() {
  ok_close=close(id_sock);
  if (ok_close==-1) {
    printf("\nError al cerrar el socket1 \n");
    exit(-1);
  }
```

```
  ok_close=close(nid_sock);
  if (ok_close==-1) {
    printf("\nError al cerrar el socket2 \n");
    exit(-1);
  }
}
```

Código del cliente (cliente.c)

```
/*Cliente para cifrador de Julio Cesar*/

#include "i_addr.h"
#define BUFF_SIZE 200



struct sockaddr_in ser_addr, cli_addr;
char *cap_mensaje(), *mensaje, *ciphertext;
char alfabeto[28]="abcdefghijklmn¤opqrstuvwxyz ",
     mens[200],soda[1];
char msg_buffer[BUFF_SIZE], buffer[BUFF_SIZE],dir_nodo[BUFF_SIZE],espera;

void resetear();
void init_buffer(char[]);

/*vbles para manejo de sockets*/
int dir_puerto,ok_bind,ok_conect,ok_read,ok_write,ok_close,
    id_sock,nid_sock,ser_addr_len,cli_addr_len,msg_size,opc,i,clave;


int main(void) {
  /*abrir socket*/
  id_sock=socket(AF_INET,SOCK_STREAM,0);
  if (id_sock==-1) {
    printf("\nError abriendo socket\n");
    exit(-1);
  }
```

```
/*captura de la info de la maquina*/
init_buffer(buffer);
system("clear");
printf("Entre direccion IP del servidor: ");  gets(dir_nodo);
printf("Entre el numero del puerto: ");gets(buffer);
dir_puerto=atoi(buffer);

/*inicializar estructura ser_addr*/
ser_addr.sin_family=AF_INET;
ser_addr.sin_addr.s_addr=inet_addr(dir_nodo);
ser_addr.sin_port=htons(dir_puerto);

/*peticion de conxion al servidor*/
ok_conect=connect(id_sock,&ser_addr,sizeof(ser_addr));
if (ok_conect==-1) {
  printf("\nError al conectarse con el servidor\n");
  exit(-1);
}

/*cuerpo del cliente*/
opc=1;
mensaje=(char *)calloc(BUFF_SIZE,sizeof(char));
ciphertext=(char *)calloc(BUFF_SIZE,sizeof(char));
while (opc!=3) {
  system("clear");
  printf("Tercer tabajo de programacion en Criptografia\n");
  printf("          Cifrador de Julio Cesar\n");
  printf("          Modelo Cliente-Servidor\n\n\n");
```

```
printf("Implementado por:\n");
printf("              Antonio Buitrago.\n");
printf("              Dollceys Mestre R.\n");
printf("              Gabriel Oliveros V.\n\n\n\n");
printf("1. Captura de mensaje y clave\n");
printf("2. Cifrar y enviar al servidor\n");
printf("3. Terminar cliente y servidor\n\n");
printf("Escoga opcion==> ");
gets(buffer);
opc=atoi(buffer);
if (opc==1) {
  resetear(mensaje);
  mensaje=cap_mensaje();
  for (i=0;i<=200;i++) mens[i]=NULL;
  for (i=0;i<=200;i++) mens[i]=mensaje[i];
  printf("Digite la clave: ");  gets(buffer);
  clave=atoi(buffer);
}
if (opc==2) {
  init_buffer(ciphertext);
  resetear(ciphertext);
  for (i=0;mensaje[i];i++) {
    soda[0]=mens[i];
    ciphertext[i]=alfabeto[(strcspn(alfabeto,&soda[0])+clave)%28];
  }

  /*enviar ciphertext al servidor*/
  msg_size=sizeof(ciphertext);
```

```
  strcpy(msg_buffer,ciphertext);
  ok_write=write(id_sock,msg_buffer,sizeof(msg_buffer));
  if (ok_write==-1) {
    printf("\nError al enviar ciphertexto\n");
    exit(-1);
  }

  /*esperar respuesta del servidor*/
  ok_read=read(id_sock,msg_buffer,sizeof(msg_buffer));
  if (ok_read==-1) {
    printf("\nError al hacer read\n");
    exit(-1);
  }

  /*resultados en el cliente*/
  printf("\nTexto plano  : %s",mensaje);
  printf("\nValor de la clave: %d",clave);
  printf("\nTexto cifrado: %s\n",ciphertext);
  printf("%s\n",msg_buffer);
  printf("Presione cualquier tecla para continuar...");
  espera=getchar();
 } /*fin de opc2*/
 if (opc==3) {
  ok_close=close(id_sock);
  if (ok_close==-1) {
    printf("\nError al hacer close\n");
    exit(-1);
  }
```

```
      printf("\nFin del cliente\n");
      exit(0);
    }
  } /*fin del Mq*/
} /*fin del main*/

void resetear (char *cadena) {
  int i;
  i=0;
  cadena=(char *)calloc(BUFF_SIZE,sizeof(char));
  for (i=0;i<BUFF_SIZE;i++) cadena[i]=NULL;
}

void init_buffer (char cadena[]) {
  int i;
  i=0;
  for (i=0;i<BUFF_SIZE;i++) cadena[i]=NULL;
}

char *cap_mensaje() {
  char *buffer;
  buffer=(char *)calloc(BUFF_SIZE,sizeof(char));
  resetear(buffer);
  printf("\nEntre mensaje: "); gets(buffer);
  return buffer;
}
```

Código del archivo de cabecera (i_addr.h)

```
#ifndef _INT_ADDR_
#define _INT_ADDR_
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/utsname.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <fcntl.h>

#endif
```

5.1.1.6 Primitivas de transferencia y recepción de mensajes

5.1.1.6.1 Breve orientación teórica. En las **primitivas de bloqueo** (a veces llamadas **primitivas síncronas**) cuando un proceso llama a **send,** especifica un destino y buffer dónde enviar ese destino. Mientras se envía el mensaje, el proceso emisor se bloque (es decir, se suspende). La instrucción que sigue a la llamada a send no se ejecuta sino hasta que el mensaje se envía en su totalidad.

Una alternativa a las primitivas con bloqueo son las **primitivas sin bloqueo** (a veces llamadas **primitivas asíncronas**). Si send no tiene bloqueo, regresa de inmediato el control a quien hizo la llamada, antes de enviar el mensaje.

Así como los diseñadores de sistemas pueden elegir entre las primitivas con o sin bloqueo, también pueden elegir entre las primitivas almacenadas en buffer o no almacenadas.

Los mensajes se pueden perder, lo cual afecta la semántica del modelo de transferencia de mensajes. Supongamos que se utilizan las primitivas por bloqueo. Cuando un cliente envía un mensaje, se le suspende hasta que el mensaje ha sido enviado. Sin embargo, cuando vuelve a iniciar, no existe garantía alguna acerca de la entrega del mensaje.

Existen tres enfoques de este problema. El primero consiste en volver a definir la semántica de **send** para hacerla no confiable. El sistema no da garantía alguna acerca

de la entrega de los mensajes. La implantación de una comunicación confiable se deja por completo en manos de los usuarios.

El segundo método exige que el núcleo de la maquina receptora envíe un reconocimiento a la maquina emisora. Sólo cuando se reciba este reconocimiento, el núcleo emisor liberará el proceso usuario (cliente). El reconocimiento va de un núcleo al otro ; ni el cliente ni el servidor ven alguna vez un reconocimiento. De la misma forma que la solicitud de un cliente a un servidor es reconocida por el núcleo del servidor, la respuesta del servidor es reconocida por el núcleo del cliente. Así una solicitud de respuesta consta de cuatro mensajes.

El tercer método aprovecha el hecho de que la comunicación cliente/servidor se estructura como solicitud del cliente al servidor, seguida de una respuesta del servidor al cliente.

5.1.1.6.2 Preguntas de concepto

a. Explique la diferencia principal entre las primitivas con bloqueo y las primitivas sin bloqueo.
b. Enuncie las ventajas y desventajas de las primitivas bloqueantes y no bloqueantes.
c. Existen dos formas para corregir el principal problema que presentan las primitivas sin bloque. Explíquelas, enuncie sus ventajas y desventajas.
d. Describa el funcionamiento de las primitivas almacenadas en buffers y las no almacenadas.
e. ¿Cuál es el principal problema que presentan las primitivas no almacenadas? ¿Cómo se puede solucionar este problema?. Explique.
f. Cuando un cliente envía un mensaje, el servidor lo recibirá. Los mensajes se pueden perder, lo cual afecta la semántica del modelo de transferencia de mensajes. ¿Cuántos métodos existen para solucionar este problema?. Explique.

5.1.1.6.3 Preguntas de análisis

a. En muchos sistemas de comunicación, las llamadas a *send* inician un cronómetro para protegerse contra la suspención indefinida del cliente si el servidor falla. Suponga que se implanta un sistema tolerante a fallas mediante varios procesadores para todos los cliente y los servidores, de forma que la probabilidad de falla es nula. ¿Cree usted que sería seguro eliminar los tiempos de espera en este sistema?

b. Si se utiliza la comunicación con almacenamiento en buffers, se dispone por lo general de una primitiva para que los usuarios creen buzones. ¿Por qué no se especifica que esta primitiva exija el tamaño del buzón?
c. Utilizando las primitivas sin bloqueo. ¿ Qué ocurriría si el emisor no puede modificar su buffers de mensajes sino hasta que el mensaje haya sido enviado?
d. En las primitivas no almacenadas en buffers el esquema funciona bien mientras el servidor llame a *receive* antes de que el cliente llame a *send.* ¿ Qué pasaría si el cliente llama a *send* antes de que el servidor llame a *receive*?

5.1.1.7 Miscelánea de llamadas y funciones

5.1.1.7.1 Breve orientación teórica.

Nombres de un socket. Getsockname, getpeername.

Para determinar las direcciones de los procesos conectados a un socket, se utilizan las llamadas getsockname y getpeername. Estas llamadas tienen aplicación en sockets orientados a conexión y se usan de la siguiente forma:

```
getsockname (sfd, addr, addrlen);
getpeername (sfd, addr, addrlen);
```

Nombre del nodo actual. Gethostname.

Para determinar el nombre oficial que tiene un nodo dentro de la red, podemos usar la llamada gethostname, que se usa como sigue:

```
gethostname (hostname, size);
```

5.1.1.7.2 Preguntas de concepto.

a. Explique el funcionamiento de las primitivas getsockname y getpeername.
b. ¿A través de que primitivas se puede conseguir el nombre del nodo actual?. Explique.

5.1.1.7.3 Práctica

5.1.1.7.3.1 *Práctica V.* Aplicación cliente/servidor con sockets del tipo SOCK_STREAM, familia de protocolos AF_INET. Donde, luego de comunicarse los dos sockets(cliente y servidor), este último envíe un mensaje al cliente y el cliente as vez devuelva el mismo mensaje al servidor. El servidor deberá luego comparar que la cadena enviada y recibida sean iguales ("eco").

Las direcciones IP no van a ser provistas por la persona que ejecutará el programa, sino que la aplicación misma deberá obtenerla del sistema haciendo uso de la instrucción gethostbyname.

Sugerencia:

El servidor puede enviarse a ejecutar de la siguiente forma:

$ servidor puerto

El cliente puede enviarse a ejecutar de la siguiente forma:

$ cliente nombre-host puerto

5.1.1.7.3.1.1 Justificación. El estudiante debe conocer formas alternas para el

desarrollo de aplicaciones cliente/servidor que mejoren su ejecución y los acerquen más al manejo real que se les da a este tipo de aplicaciones en un ambiente de red.

5.1.1.7.3.1.2 Objetivos

- Que el estudiante conozca la existencia del archivo /etc/hosts en los ambientes UNIX y de manera practica conozca una de sus utilidades.
- Que maneje la primitiva gethostname y la domine en el proceso de desarrollo de aplicaciones cliente servidor.

5.1.2 LLamada a procedimiento remoto (rpc)

5.1.2.1 Conceptos generales de las rpc.

5.1.2.1.1 Breve orientación teórica . Aunque el modelo cliente/servidor es una forma conveniente de estructurar un sistema operativo distribuido, adolece de una enfermedad incurable; el paradigma básico en torno al cual se construye la comunicación es la entrada/salida. Los procedimientos *send* y *receive* están reservados para la realización de E/S no es uno de los conceptos fundamentales de los sistemas centralizados, el hecho de que sean la base del cómputo distribuido es visto por las personas que laboran en este campo como un grave error. Su objetivo es lograr que el cómputo distribuido se vea como el cómputo centralizado. La construcción de todo en torno de la E/S no es la forma de lograrlo.

La mayoría de los sistemas de ordenadores están conectados en red, soportando comunicación de datos entre ellos. Como resultado de esto, se han desarrollado muchas técnicas para soportar el desarrollo de aplicaciones que requieren procesos en diferentes sistemas, para comunicar y coordinar sus actividades. Una de estas técnicas son las RPC "Remote Procedure Call", (llamadas a procedimientos remotos)

El concepto de RPC es una sencilla técnica para desarrollar aplicaciones donde se requiere la comunicación entre procesadores que cooperan en un sistema distribuido. RPC es una técnica consistente, como evidencia la existencia de muchas especificaciones e implementaciones.

El mecanismo RPC proporciona un servicio para el programador de aplicaciones que le permite el uso transparente de un servidor para proporcionar alguna actividad por parte de la aplicación. Esto efectivamente puede ser utilizado para interactuar con un servidor computacional o con una Base de Datos, y ha sido usado por algunos sistemas para proporcionar acceso a servicios del sistema operativo. El último uso conocido es en sistemas basados en microordenadores, que da un mejor resultado que en los sistemas tradicionales encontrados en la mayoría de sistemas UNIX, y es especialmente utilizado en sistemas operativos distribuidos.

Las RPC son expresadas como procedimientos ordinarios. Estas llamadas no requieren un compilador especial para el código fuente del programa. Las ventajas de esto incluyen:

- Transparencia: La capa RPC puede ser reemplazada con llamadas a funciones directas si llegan a estar disponibles.
- Familiaridad de la interface: La mayoría de programadores están acostumbrados a una forma de llamadas a procedimientos. Esto permite una fácil adaptación al mecanismo RPC en sistemas ya implantados.

5.1.2.1.2 Preguntas de concepto

a. ¿A qué nos referimos cuando hablamos de RPC de manera general?

b. ¿Cómo son expresadas las RPC?

c. ¿RPC requiere de un compilador especial para el código fuente del programa? ¿Es esto ventajoso o no?

5.1.2.1.3 Preguntas de análisis

a. ¿Qué le proporciona RPC al programador?

b. ¿En donde puede ser usado RPC?

5.1.2.2 Transferencia de parámetros

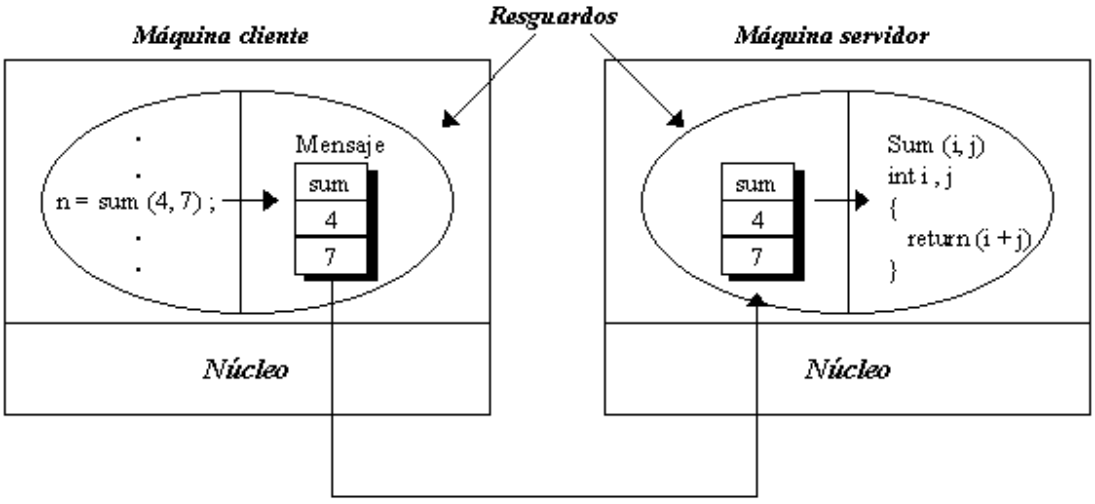
5.1.2.2.1 Breve orientación teórica . Las RPC son realmente una manera de ocultar el protocolo de pase de mensajes.

Esto proporciona una manera adecuada de permitir una interpretación a alto nivel, abstrayendo el mecanismo de comunicación a bajo nivel, es decir haciendo transparente al programador el protocolo de red. Las RPC tienen la intención de ser parecidas a una llamada a procedimiento ordinario, pero atravesando la red transparentemente. Un proceso realiza una RPC poniendo sus parámetros y una dirección de retorno en la pila, y salta al comienzo del procedimiento. El procedimiento es responsable de acceder y usar la red. Cuando la ejecución remota finaliza, el procedimiento salta hacia atrás a la dirección de retorno. El proceso de llamada (cliente) entonces continua.

La función del resguardo (figura 1) del cliente es tomar sus parámetros, empacarlos en un mensaje y enviarlos al resguardo del servidor. Aunque esto parece directo, no es tan sencillo como aparenta. El empacamiento de parámetros en un mensaje se denomina **ordenamiento de parámetros.**

El ejemplo más sencillo es considerar un procedimiento remoto, *sum*(i,j), que toma dos parámetros enteros y regresa su suma aritmética.

La llamada a *sum*, con los parámetros 4 y 7, aparece en la parte izquierda del proceso cliente de la figura 1. El resguardo del cliente toma sus dos parámetros y los coloca en un mensaje de la forma que se indica. También coloca en el mensaje el nombre o número del procedimiento por llamar, puesto que el servidor podría soportar varias llamadas y se le tiene que indicar cual de ellas se necesita.



Cálculo remoto de sum (4 , 7).

Figura 1. Calculo remoto de sum (4,7).

Cuando el mensaje llega al servidor, el resguardo examina éste para ver cuál procedimiento necesita y entonces lleva a cabo la llamada apropiada.

Cuando el servidor termina su labor, su resguardo recupera el control. Toma el resultado proporcionado por el servidor y lo empaca en un mensaje. Este mensaje se envía de regreso al resguardo de cliente, que lo desempaca y regresa el valor al procedimiento cliente.

Si las máquinas cliente y servidor son idénticas y todos los parámetros y resultado son de tipo escalar, como enteros, caracteres o booleanos, este método funciona bien. Sin embargo, en un sistema distribuido de gran tamaño, es común tener distintos tipos de máquinas. Cada una tiene a menudo su propia representación de los números, caracteres y otros elementos. Por ejemplo, las mainframes de IBM utilizan el código de caracteres EBCDIC, mientras que las computadoras personales de IBM utilizan ASCII. en consecuencia, no es posible pasar un parámetro carácter de una PC de IBM cliente a una mainframe IBM servidor, mediante el sencillo esquema de la figura 1 ; el servidor interpretará lo interpretará de manera incorrecta.

Aparecen otros problemas similares con la representación de enteros (complemento a 1 o complemento a 2) y de manera particular, en los números de punto flotante. Además existe un problema mucho más irritante, puesto que en ciertas máquinas, como la Intel 486, numeran sus bytes de derecha a izquierda, mientras que otras, como la Sun SPARC, los numeran en el orden contrario. El formato de Intel se llama **little endian** (partidarios del extremo menor) y el de SPARC **big endian** (partidarios del extremo mayor).

5.1.2.2.2 Preguntas de concepto.

a. ¿ Cómo se debe representar la información de los mensajes?

b. ¿ Cuál es el origen de los procedimientos de resguardo o stubs?

5.1.2.2.3 Preguntas de análisis

a. ¿Qué hace diferente a las RPC Vs. la filosofía Cliente/Servidor?

b. Explique de manera general el funcionamiento de RPC.

5.1.2.3 Protocolos rpc

5.1.2.3.1 Breve orientación teórica. En teoría, cualquier protocolo antiguo puede funcionar si obtiene los bits del núcleo del cliente y los lleva al núcleo del servidor; pero en la práctica hay que tomar decisiones importantes en este punto, decisiones que pueden tener un fuerte impacto en el desempeño.

La primera decisión está entre un protocolo orientado a la conexión o un protocolo sin conexión. En el primer caso, al momento en que el cliente se conecta con el servidor, se establece una conexión entre ellos. Todo el tráfico, en ambas direcciones, utiliza esta conexión.

La ventaja de contar con una conexión es que la comunicación es más fácil. Cuando un núcleo envía un mensaje, no tiene que preocuparse por su pérdida, ni tampoco por los reconocimientos. Todo ello se maneja a nivel inferior, mediante que el software que soporta la conexión. Cuando se opera una red amplia, esta ventaja es con frecuencia irresistible.

La desventaja en una LAN, es la pérdida de desempeño. Todo ese software adicional estorba en el camino. Además, la ventaja principal (no perder los paquetes) difícilmente se necesita en una LAN, puesto que las LAN son confiables en este sentido. En consecuencia, la mayoría de los sistemas distribuidos que pretenden utilizarle en un edificio o campus utilizan los protocolos sin conexión.

La segunda opción principal está en utilizar un protocolo de propósito general o alguno diseñado de forma específica para RPC. Puesto que no existen estándares en esta área, el uso de un protocolo RPC adaptado quiere decir por lo general que cada quien diseñe el suyo.

Algunos sistemas distribuidos utilizan IP (o UDP, integrado a IP) como el protocolo básico.

5.1.2.3.2 Preguntas de análisis

a. ¿En RPC se debe utilizar protocolos orientados a la conexión o sin conexión?

b. ¿Qué ventajas y desventajas traería el uso de IP (o UDP, integrado a IP) como el protocolo básico?

5.1.2.4 Las grandes lineas del protocolo

5.1.2.4.1 Breve orientación teórica. El protocolo debe permitir:

- La identificación del procedimiento
- La autentificación de la petición

La identificación del procedimiento

El principio es el de agrupar los diferentes procedimientos en un programa. Los diferentes procedimientos que constituyen un mismo programa contribuyen a la realización de un servicio específico. Por ejemplo, el protocolo NFS constituye un programa de protocolo RPC y contiene diferentes procedimientos cuyo punto común es que todos permitan manipular archivos a distancia.

Un programa se identificará por un número entero y cada procedimiento de un programa será igualmente identificado por un entero. A título de ejemplo el programa NFS lleva el número 100003 y los procedimientos de lectura y escritura llevan los números 6 y 8. Finalmente para permitir la evolución de los programas, cada uno posee un número de versión.

La autenficación

La definición del protocolo prevee la posibilidad de que un cliente se identifique a un servidor, lo que permite asegurar la seguridad de los accesos a los objetos del sistema distante. Los mensaje intercambiados en el curso de llamadas a procedimientos remotos llevan información relativa a esta identificación. Como el protocolo es independiente del sistema subyacente, están previstos diferentes estilos de autentificación (por ejemplo, la ausencia de autentificación o una autentificación UNIX), dejando la posibilidad de definir otros nuevos.

5.1.2.4.2 Preguntas de concepto

a. Ordene de lo particular a lo general

   Procedimientos, servicios, programas

   Servicios, programas, procedimientos.

   Procedimientos, programas, servicios.

   Programas, servicios, procedimientos.

b. ¿A través de qué, se establece la identificación de los procedimientos y programas de un servicio RPC?

c. Todo procedimiento tiene un procedimiento de número cero que no permite hace ningún cálculo. ¿Entonces, cual es su funcionalidad?

5.1.2.4.3 Preguntas de análisis

a. RPC permite el llamado a procedimientos que se encuentran en máquinas remotas. ¿A través de que mecanismo se puede implementar la seguridad en dichos accesos?

b. ¿Qué hace la orden *rpcinfo*?

5.1.2.5 Los diferentes niveles de utilización

5.1.2.5.1 Breve orientación teórica. Hay tres niveles de utilización del mecanismo de RPC tal como ha sido implantado en UNIX.

Cada uno de estos niveles ofrece transparencia al usuario, es decir, demanda un conocimiento más o menos fino (hasta nulo) del protocolo y de los mecanismos de más bajo nivel, tales como, por ejemplo, los socket.

El nivel alto

Este es el que oculta el máximo de detalles al usuario. Este solo debe realizar una llamada de función de una biblioteca, especificando el nombre de la máquina objetivo y los diferentes parámetros de la llamada. Evidentemente, a este nivel no es posible desarrollar nuevos servicios RPC (si no es por composición de los servicios existentes).

El nivel intermedio

Es incontestablemente el más interesante para el desarrollador. Supone un conocimiento mínimo de los protocolos XDR y RPC y es suficiente para desarrollar la mayoría de las aplicaciones. Descarga totalmente al usuario de la manipulación de los sockets.

El interfaz disponible a este nivel se apoya sobre el protocolo UDP. Esto significa que el tamaño de los mensajes intercambiados (y. por tanto, el tamaño cié los

parámetros v de los resultados de las funciones llamadas) es limitado. Cuando esta limitación es molesta, es necesario utilizar el interfaz propuesto por el nivel bajo. La sucesión de operaciones a realizar para definir un servicio de este tipo consiste en:

- Escribir las diferentes funciones sobre el servidor;
- Después de haber escogido los números de programa y de versión, solicitar la anotación de las diferentes funciones por el demonio «portmap» por medio de la función **regiserrpc.**

La llamada a una función desde una posición remota se realiza por medio de la función **callrpc.**

El nivel bajo

Su utilización es mucho más compleja y supone un buen conocimiento de los mecanismos concernientes a los sockets. Se muestra necesaria para las aplicaciones donde las opciones elegidas en el nivel intermedio

5.1.2.5.2 Preguntas de concepto

a. Mencione una instrucción del nivel alto de programación en RPC
b. A través de que instrucción se coloca a disposición un servicio RPC en una máquina

5.1.2.5.3 Preguntas de análisis

a. ¿Es posible que un desarrollador implemente un nuevo servicio RPC usando las llamadas a las funciones de la biblioteca del primer nivel de programación?
b. ¿Basta con hacer un nuevo servicio RPC y registrarlo en el sistema?

5.1.2.6 Grupo general de prácticas rpc

5.1.2.6.1 Práctica I

5.1.2.6.1.1 Enunciado. Realizar una aplicación RPC que diga el número de usuarios conectados en una maquina remota, usando la función rnusers o en su defecto su equivalente a través de una llamada con callrpc.

5.1.2.6.1.2 *Justificación.* El estudiante de una forma amena y real puede introducirse en el uso de funciones RPC, de primer nivel, al tiempo que observa un ejemplo de la aplicación de este tipo de llamadas con miras a la formación experimental del concepto de un sistema distribuido.

5.1.2.6.1.3 Objetivos

- Que el estudiante comprenda la manera de utilizar las funciones **rnusers** y **callrpc.**
- Lograr que el estudiante descubra las diferencias y semejanzas que se presentan al hacer uso de estas funciones.
- Fortalecer los conceptos teóricos que el estudiante posee sobre el funcionamiento de dichas funciones.

5.1.2.6.2 Práctica II

5.1.2.6.2.1 *Enunciado*. Realizar una aplicación RPC, del primer nivel de programación, que permita saber el puerto asociado a un servicio RPC activo en una maquina remota.

5.1.2.6.2.2 *Justificación.* Es una de las formas más elementales de iniciarse en el uso de las funciones RPC. Sugiere el uso de las funciones getrpcbynumber y getrpcport. Además induce al estudiante a la manipulación de herramientas complementarias del servicio RPC como es el caso de rpcinfo para mirar primero la información referente a los servicios activos en la maquina par luego introducir un valor significativo durante la ejecución de su aplicación.

5.1.2.6.2.3 Objetivos

- Inducir al estudiante a manipular las funciones a utilizar en este caso (getrpcbynumber y getrpcport) y a comprender su forma de operar.
- Que el estudiante se entrene en la utilización de las funciones RPC, del primer nivel de programación mencionadas anteriormente.

5.1.2.6.3 Práctica III

5.1.2.6.3.1 Enunciado. Realizar una aplicación RPC, que posea:

- Un servidor que registre una función RPC (registerrpc) y ponga a disposición dicha función (svc_run)
- Un cliente que haga el llamado a la función (callrpc) y le transfiera el correspondiente parámetro.

Para hacer más práctico y uniforme el ejemplo se sugiere que la función halle el factorial de un número.

5.1.2.6.3.2 *Justificación.* Esta práctica se convierte en una buena introducción para lo que será la programación de nivel intermedio en RPC, a la vez que el estudiante con base en la abstracción que le ofrece el utilizar las sentencias sugeridas en el enunciado se le facilitará la asimilación de conceptos básicos de RPC que le ayudará en una concepción más específica de lo que es un sistema distribuido.

5.1.2.6.3.3 Objetivos

- Introducir al estudiante en la programación de nivel intermedio de RPC.
- Entrenar al estudiante en lo relacionado con la utilización de las funciones sugeridas en el enunciado, así como la transferencia de parámetros en RPC.

5.1.2.6.4 Práctica IV

5.1.2.6.4.1 *Enunciado.* Realizar una aplicación RPC para calcular la solución de una ecuación de segundo grado. El cliente deberá tomar los coeficientes de la ecuación y transmitirlos como parámetros al servidor, donde se llevaran acabo los cálculos

5.1.2.6.4.2 Justificación. A través de esta aplicación el estudiante podrá:

- Manipular las estructuras empleadas para la transmisión de parámetros.
- Registrar completamente su propio programa dentro de los servicios RPC que ofrece la máquina.
- Adiestrarse en la programación de nivel intermedio de RPC que es para muchos el nivel más interesante para un desarrollador.

5.1.2.6.4.3 Objetivos

- Que el estudiante conozca y se entrene en la forma de manipular las estructuras empleadas para la transmisión de parámetros.
- Afianzar al estudiante en la programación de nivel intermedio de RPC.
- Lograr que el estudiante visualice la utilización de RPC en problemas cotidianos.

5.1.2.6.5 Práctica V

5.1.2.6.5.1 Enunciado. Analizar, compilar y ejecutar la aplicación "hora", disponible en el servidor Linux, que como su nombre lo indica solicita la hora a una máquina remota. Anote sus experiencias y resultados.

5.1.2.6.5.2 *Justificación.* Esta es una forma diferente de realización de aplicaciones RPC, la cual corresponde al nivel bajo. Aunque es más compleja su realización, se hace necesaria cuando las opciones del nivel intermedio no son apropiadas.

Con esta práctica el estudiante tendrá la necesidad de utilizar herramientas complementarias al servicio RPC como es el caso de *rpcgen* (compilador de RPC), el cual toma como base un archivo de tipo *.x y genera el código del cliente y servidor de la aplicación.

La aplicación cuenta con un par de archivos de texto que señalan el camino a seguir por el estudiante para llegar hasta la ejecución final de la misma, camino que enriquecerá sus conocimiento en este ultimo (aunque difícil) nivel de programación.

5.1.2.6.5.3 Objetivos

- Ilustrar al estudiante en lo referente a la programación de nivel bajo de RPC e inducirlo al análisis y comprensión de la misma.
- Enseñar al estudiante el uso de herramientas complementarias al servicio RPC, como lo es **rpcgen**(compilador de RPC).

5.1.3 Comunicación en grupo

5.1.3.1 Conceptos generales

5.1.3.1.1 Breve orientación teórica. Un grupo es una colección de procesos que actúan juntos en cierto sistema o alguna forma determinada por el usuario.

La propiedad fundamental de todos los grupos es que cuando un mensaje se envía al propio grupo, todos los miembros de éste lo reciben. Es una forma de comunicación **uno-muchos** (un emisor, muchos receptores) y contrasta con la **comunicación puntual** de la figura 2.

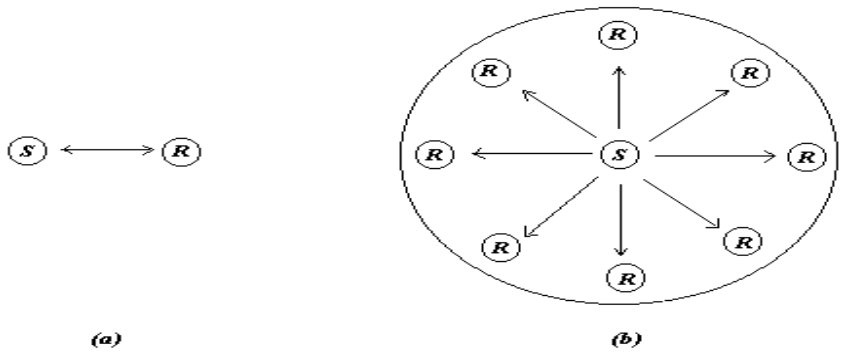


Figura 2. La comunicación puntual y la comunicación uno-muchos .

Los grupos son dinámicos. Se pueden crear nuevos grupos y destruir grupos anteriores. Un proceso se puede unir a un grupo o dejar otro. Un proceso puede ser miembro de varios grupos a la vez. En consecuencia, se necesitan mecanismos para el manejo de grupos y la membresia de los mismos.

Los grupos son algo parecido a las organizaciones sociales. Una persona puede ser miembro de un club de lectores, un club de tenis y una organización ambientalista. En un día particular, él podría recibir correo (mensajes) que le avisen de un nuevo libro para hornear pasteles de cumpleaños, el torneo anual del día de las madres y el inicio de una campaña para salvar a las marmotas del sur. En cualquier momento, él es libre de dejar todos o alguno de estos grupos y unirse a otros.

La finalidad de presentar los grupos es permitir a los procesos que trabajen con colecciones de procesos como una abstracción. Así, un proceso puede enviar un mensaje a un grupo de servidores sin tener que conocer su número o su localización, que puede cambiar de una llamada a la siguiente.

En ciertas redes, es posible crear una dirección especial de red (por ejemplo, indicada al hacer que uno de los bits de orden superior tome el valor 1) a la que pueden escuchar varias máquinas. Cuando se envía un mensaje a una de estas direcciones, se entrega de manera automática a todas las máquinas que escuchan a esa dirección. Esta técnica se llama **multitransmisión.**

Las redes que no tienen multitransmisión a menudo siguen teniendo **transmisión simple,** lo que significa que los paquetes que contienen cierta dirección (por ejemplo, 0) se entregan a todas las máquinas.

Por último, si no se dispone de la multitransmisión o la transmisión simple, se puede implantar la comunicación en grupo mediante la transmisión por parte del emisor de paquetes individuales a cada uno de los miembros del grupo. El envío de un mensaje de un emisor a un receptor se llama a veces **unitransmisión,** para distinguirla de los otros tipos de transmisión.

5.1.3.1.2 Preguntas de concepto

a. ¿Qué entiende usted por GRUPO y cual es su propiedad fundamental?
b. ¿Cuál es la diferencia primordial entre la comunicación puntual y la comunicación Uno-Muchos? Explique con un gráfico.
c. ¿Qué entiende usted por Multitransmision?
d. Además de la multitransmisión y la transmisión simple ¿ qué otro tipo de transmisión se puede implantar?

5.1.3.1.3 Preguntas de análisis

a. Realice una analogía entre Grupo y algún acontecimiento de la vida cotidiana.
b. ¿La transmisión simple puede ser utilizada para trabajar en comunicación en grupo? ¿Qué desventaja presenta?

5.1.3.2 Aspectos del diseño

5.1.3.2.1 Breve orientación teórica. La comunicación en grupo tiene posibilidades de diseño similares a la transferencia regular de mensajes, como el almacenamiento en buffers *vs.* el no almacenamiento, bloqueo *vs.* no bloqueo, etc.

Sin embargo, también existen un gran número de opciones adicionales por realizar, ya que el envío a un grupo es distinto de manera inherente del envío a un proceso. Además, los grupos se pueden organizar internamente de varias formas. También se pueden direccionar de formas novedosas que no son importantes en la comunicación puntual

Grupos cerrados *vs.* grupos abiertos

Los sistemas que soportan la comunicación en grupo se pueden dividir en dos categorías, según quién pueda enviar a quién. Algunos sistemas soportan los grupos cerrados, donde sólo los miembros del grupo pueden enviar hacia el grupo. Los extraños no pueden enviar mensajes al grupo como un todo, aunque pueden enviar mensajes a miembros del grupo en lo Individual. En contraste, otros sistemas soportan los **grupos** abiertos, que no tienen esta propiedad. Si se utilizan los grupos abiertos, cualquier proceso del sistema puede enviar a cualquier grupo. La diferencia entre los grupos cerrados y abiertos se muestra en la figura 3.

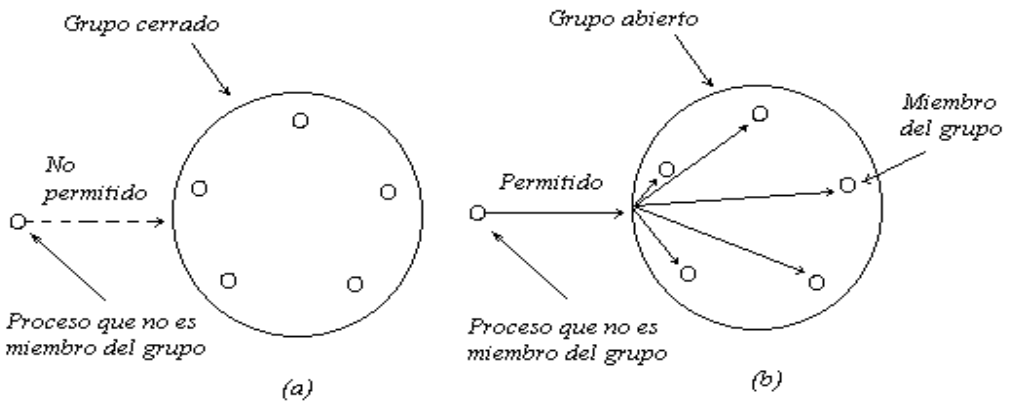


Figura 3. Comparación entre grupos abiertos y grupos cerrados.

Grupos de compañeros *vs.* grupos jerárquicos

La distinción entre los grupos cerrados y abiertos se relaciona con la pregunta de quién se puede comunicar con el grupo. Otra distinción importante tiene que ver con la estructura interna del grupo. En algunos grupos, todos los procesos son iguales. Nadie es el jefe y todas las decisiones se toman en forma colectiva. En otros grupos, existe cierto tipo de jerarquía. Estos patrones de comunicación se muestran en la figura 4.

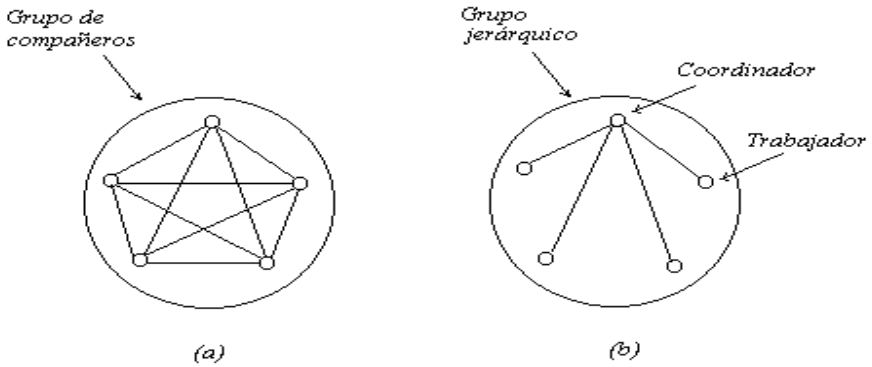


Figura 4. La comunicación en un grupo de compañeros y en un simple grupo jerárquico .

Cada una de estas organizaciones tiene sus propias ventajas y desventajas. El grupo de compañeros es simétrico y no tiene punto de falla. Si uno de los procesos falla, el grupo sólo se vuelve más pequeño, pero puede continuar.

El grupo jerárquico tiene las propiedades opuestas. La pérdida del coordinador lleva a todo el grupo a un agobiante alto, pero mientras se mantenga en ejecución, puede tomar decisiones sin molestar a los demás.

Membresía del grupo

Si se utiliza la comunicación en grupo, se requiere cierto método para la creación y eliminación de grupos, así como para permitir a los procesos que se unan o dejen grupos. Un posible método es tener un **servidor de grupos** al cual se pueden enviar todas las solicitudes. El servidor de grupos puede mantener entonces una base de datos de todos los grupos y sus membresías exactas. Este método es directo, eficiente

y fácil de implantar. Por desgracia, comparte una desventaja fundamental con todas las técnicas centralizadas: un punto de falla. Si el servidor de grupos falla, deja de existir el manejo de los mismos. Es probable que la mayoría o todos los grupos deban reconstruirse a partir de cero, terminando con todo el trabajo realizado hasta entonces.

El método opuesto es manejar la membresía de grupo en forma distribuida. En un grupo abierto, un extraño puede enviar un mensaje a todos los miembros del grupo para anunciar su presencia. En un grupo cerrado se necesita algo similar (de hecho, incluso los grupos cerrados deben estar abiertos a la opción de admitir otro miembro). Para salir de un grupo, basta que el miembro envíe un mensaje de despedida a todos.

5.1.3.2.2 Preguntas de concepto

a. ¿Cuáles son las categorias en las cuales se pueden dividr los sistemas que soportan la comunicación en grupo?
b. ¿Cuándo se debe utilizar los grupos abiertos y los grupos cerrados? Explique.
c. Explique el concepto de grupo jerarquico y grupo de compañeros.
d. ¿Qué métodos se pueden utilizar para la creación y eliminación de grupos, así como para permitir a los procesos que se unan o dejen grupos?

5.1.3.2.3 Preguntas de análisis

a. ¿Cuáles son las ventajas y desventajas que en su concepto presentan los grupos jerárquicos y los grupos de compañeros? Explique con ejemplos.

b. Explique los problemas más comunes asociados con la membresía de los grupos.

5.1.3.3 Direccionamiento

5.1.3.3.1 Breve orientacion teórica. Para enviar un mensaje a un grupo, un proceso debe tener una forma de especificar dicho grupo.

En otras palabras, los grupos deben poder direccionarse, al igual que los procesos. Una forma es darle a cada grupo una dirección, parecida a una dirección de proceso. Si la red soporta la multitransmisión, la dirección del grupo se puede asociar con una dirección de multitransmisión, de forma que cada mensaje enviado a la dirección del grupo se pueda multitransmitir

Si el hardware no soporta la multitransmisión pero sí la transmisión simple, el mensaje se puede transmitir. Cada núcleo lo recibirá y extraerá de él la dirección del grupo. Si ninguno de los procesos en la máquina es un miembro del grupo, entonces se descarta la transmisión. En caso contrario, se transfiere a todos los miembros del grupo.

Por último, si no se soporta la multitransmisión o la transmisión simple, el núcleo de la máquina emisora debe contar con una lista de las máquinas que tienen procesos pertenecientes al grupo, para entonces enviar a cada una un mensaje puntual. Estos tres métodos de implantación se muestran en la figura 5.

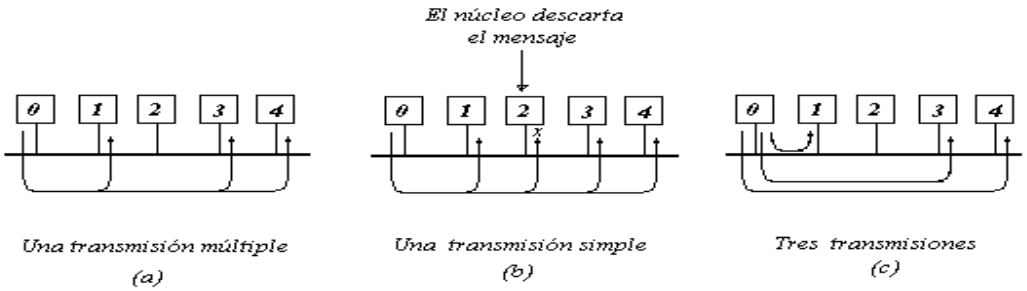


Figura 5. El proceso 0 enviado a un grupo que consta de los procesos 1, 3 y 4**.**

Un segundo método de direccionamiento de grupo consiste en pedir al emisor una lista explícita de todos los destinos (por ejemplo, direcciones IP). Si se utiliza este método, el parámetro de la llamada *send* que especifica el destino es un apuntador a una lista de direcciones.

La comunicación en grupo también permite un tercer método, un tanto novedoso, de direccionamiento, que llamaremos **dircccionamiento de predicados.** Con este sistema, se envía cada mensaje a todos los miembros del grupo (o tal vez a todo el sistema) mediante uno de los métodos ya descritos, pero con nuevo giro. Cada mensaje contiene un predicado (expresión booleana) para ser evaluado. El predicado puede utilizar el número de máquina del receptor, sus variables locales u otros factores. Si el valor del predicado es verdadero, se acepta el mensaje. SÍ es falso, el mensaje se descarta.

Primitivas send y receive

En forma ideal, la comunicación puntual y la comunicación en grupo deberían combinarse en un conjunto de primitivas. Sin embargo, si RPC es el mecanismo usual de comunicación del usuario, en vez de los simples *send y receive,* entonces es difícil combinar RPC y la comunicación en grupo. El envío de un mensaje a un grupo no se puede modelar como llamada a un procedimiento. La principal dificultad es que, con RPC, el cliente envía un mensaje al servidor y obtiene de regreso una respuesta. Con la comunicación en grupo, existen en potencia *n* respuestas diferentes. ¿Cómo podría trabajar una llamada de procedimiento con *n* respuestas? En consecuencia, un método común es abandonar el modelo solicitud/respuesta (en los dos sentidos) subyacente en RPC y regresar a las llamadas explícitas para el envío y recepción (modelo de un sentido).

Atomicidad

Una característica de la comunicación en grupo a la que hemos aludido varias veces es la propiedad del todo o nada. La mayoría de los sistemas de comunicación en grupo están diseñados de forma que, cuando se envíe un mensaje a un grupo, éste llegue de manera correcta a todos los miembros del grupo o a ninguno de ellos. No se permiten situaciones en las que ciertos miembros reciben un mensaje y otros no. La propiedad del todo o nada en la entrega se llama **atomicidad o transmisión atómica.**

La atomicidad es deseable, puesto que facilita la programación de los sistemas distribuidos. Si un proceso envía un mensaje al grupo, no tiene que preocuparse por qué hacer si alguno de ellos no lo obtiene.

Ordenamiento de mensajes

Para que la comunicación en grupo sea fácil de comprender y utilizar, se necesitan dos propiedades. La primera es la transmisión atómica, ya analizada. Ésta garantiza que un mensaje enviado al grupo llegue a todos los miembros o a ninguno. La segunda propiedad se refiere al ordenamiento de mensajes.

La mejor garantía es la entrega inmediata de todos los mensajes, en el orden en que fueron enviados. Si el proceso O envía el mensaje *A* y un poco después el proceso 4 envía el mensaje *B,* el sistema debe entregar en primer lugar *A* a todos los miembros del grupo y después entregar *B* a todos los miembros del grupo. De esta forma, todos los receptores obtiene todos los mensajes en el mismo orden. Este patrón de entrega es algo comprensible para los programadores y en el cual basan su software. Lo llamaremos **ordenamiento con respecto al tiempo global,** puesto que entrega todos los mensajes en el orden preciso con el que fueron enviados .

Grupos traslapados

Como hemos mencionado, un proceso puede ser miembro de varios grupos a la vez. Este hecho puede provocar un nuevo tipo de inconsistencia. Para ver el problema,

observemos la figura 6, la cual muestra dos grupos, 1 y 2. Los procesos *A, B y C* son miembros del grupo 1 y los procesos *B, C* y *D* son miembros del grupo 2.

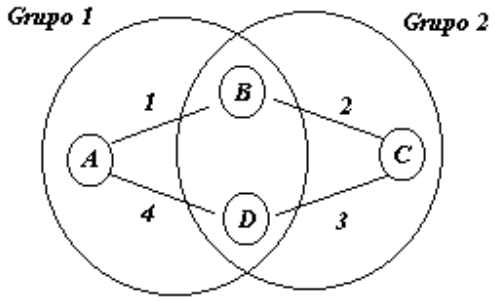


Figura. 6. Cuatro procesos A, B, C y D y cuatro mensajes..

El problema aquí es que, aunque existe un ordenamiento con respecto al tiempo global dentro de cada grupo, no es necesario que exista coordinación entre varios grupos. Algunos sistemas soportan un ordenamiento bien definido entre los grupos traslapados y otros no.

Escalabilidad

Nuestro aspecto final del diseño es la escalabilidad. Muchos algoritmos funcionan bien mientras todos los grupos tengan unos cuantos miembros, pero ¿qué ocurre cuando existen decenas, centenas o incluso miles de miembros por grupo? ¿O miles de grupos? Además, ¿qué ocurre si el sistema es tan grande que no cabe en una LAN, de modo que se necesiten varias LAN y compuertas? ¿Qué ocurre si los grupos están diseminados en varios continentes?

La presencia de compuertas puede afectar muchas propiedades de la implantación. Para comenzar, la multitransmisión es más complicada. Consideremos, por ejemplo, la red que se muestra en la figura 6. Consta de cuatro LAN y cuatro compuertas, con el fin de protegerse contra la falla de cualquier compuerta.

Imaginemos que una de las máquinas de la LAN 2 ejecuta una multitransmisión. Cuando el paquete de multitransmisión llega a las compuertas *Gl* y *G3,* ¿qué deben hacer éstas? Si lo descartan, la mayoría de las máquinas nunca lo verán, lo cual destruye su valor como multitransmisión. Sin embargo, si el algoritmo sólo tiene compuertas hacia todas las mul-titransmisiones, entonces el paquete se copiará a la LAN 1 y la LAN 4 y poco después a la LAN 3 dos veces. Peor aún, la compuerta *G2* verá la multitransmisión de *G4,* la copiará a la LAN 2 y viceversa Es claro que se necesita un algoritmo más complejo, que mantenga un registro de los paquetes anteriores, con el fin de evitar un crecimiento exponencial en el número de multitransmisiones de paquetes.

5.1.3.3.2 Preguntas de concepto

a. ¿Cuáles son los métodos más utlizados en el direccionamiento de los grupos. Explique su funcionamiento.
b. ¿Cuál es la principal dificultad que se presenta al trabajar comunicación en grupo con RPC?
c. Explique el funcionamiento de SEND y RECEIVE en la comunicación en grupo.
d. ¿Qué se entiende por atomicidad?
e. Explique el método de ordenamiento con respecto al tiempo global.

5.1.3.3.3 Preguntas de análisis

a. ¿Cómo podría trabajar una llamada de procedimiento remoto con *n* respuestas?

b. Teniendo en cuenta la Atomicidad ¿los metodos descritos anteriormente (Transmision simple, multiple y unitransmision) fallan? ¿Cómo podría solucionarse?

c. Describa un algoritmo sencillo en el cual exista la posibilidad de transmisión atómica.

5.1.3.4 Practica de comunicación en grupos

5.1.3.4.1 Enunciado. Analice, contraste con la teoría de comunicación en grupos y elabores conclusiones al respecto del código dispuesto más adelante.

Descripción de su funcionamiento:

El programa principal se va a encargar de leer dos matrices y comprobar si se pueden multiplicar. Acto seguido van a arrancar tantos procesos como le hayamos indicado en la línea de órdenes para multiplicar las matrices. Cada proceso se va a encargar de generar una fila de la matriz producto mientras queden filas por generar. Naturalmente, las matrices que intervienen en la operación deben estar en memoria compartida. Para controlar la fila que debe generar cada proceso, vamos a utilizar un semáforo que se inicializa con en el total de filas de la matriz producto y se va decrementando por cada fila generada. El proceso principal se queda esperando a que todos los demás terminen para presentar el resultado.

5.1.3.4.2 Justificación. El anterior sirve también como un ejemplo de paralelismo, sin embargo en sistemas monoprocesadores como los que disponemos este método no resulta eficiente, pero es un buen ejercicio de sincronismo y comunicación en grupo.

5.1.3.4.3 Objetivos

- Fortalecer los conocimientos teóricos que el estudiante posee acerca del funcionamiento y en general todo lo concerniente a la comunicación en grupo.
- Inducir al estudiante a analizar, comparar y elaborar conclusiones con relación a la teoría de comunicación en grupo y lo que se aprecia en el código presentado.
- Lograr en el estudiante un entendimiento considerable de la forma de operar de la comunicación en grupo y las características de esta.

5.2 PROGRAMACION PARALELA

5.2.1 Generalidades de la programación paralela

5.2.1.1 ***Breve Orientación Teórica***. Una instrucción puede especificar, además de varias operaciones aritméticas, la dirección de un dato que puede ser leído o escrito en memoria y/o la dirección de la próxima instrucción a ser ejecutada.
Un computador es susceptible de ser programado en términos de éste modelo básico usando el lenguaje de maquina, aunque para la mayoría de los casos resulta complejo, pues se tiene que guardar el "track" de millones de posiciones de memoria y organizar la ejecución de miles de instrucciones de máquinas. Técnicas de diseño modular son aplicadas en la construcción de complejos programas partiendo de simples componentes y tales componentes están estructurados en términos de altos niveles de abstracción tales como estructuras de datos, ciclos interactivos y procedimientos. Abstracciones tales como los procedimientos hacen más fácil el seguimiento de la modularidad observando los objetos que van a ser utilizados, sin tener en cuenta su estructura interna. Entonces hacer lenguajes de alto nivel tales como Fortran, Pascal, C y Ada los cuales guían el diseño expresado en termino de esas abstracciones para ser traducidas automáticamente al código ejecutable.

La programación paralela introduce fuentes adicionales de la complejidad: si programáramos en el nivel más bajo, no sólo aumentaría el número de las instrucciones ejecutadas, sino que también necesitaríamos manejar explícitamente la ejecución de millares de procesadores y coordinar millones de interacciones del

interprocessor. Por lo tanto, la abstracción y la modularidad son por lo menos tan importantes como en la programación secuencial. En hecho, acentuaremos la modularidad como cuarto requisito fundamental para el software paralelo, además de la ejecución simultánea, la escalabilidad y la localización.

Algunos conceptos importantes

Un cómputo: consiste en un conjunto de tareas

Un canal: es una cola de mensaje desde la cual un remitente puede poner mensajes y de cuál un receptor puede quitar un mensaje, y los bloquea si los mensajes no están disponibles. Una colección de mensajes "conectan" a las tareas entre si.

Una tarea: encapsula un programa y una memoria local y define un conjunto de los puertos que definen la interfaz a su ambiente.

El cómputo paralelo consiste en unas o más tareas. Las tareas se ejecutan en paralelo. El número de tareas puede variar durante la ejecución de programa.

5.2.1.2 Preguntas de concepto

a. ¿En qué consiste el computo en paralelo?

b. Consulte el modelo de programación en paralelo, tarea/canal, y anote sus principales características.

c. Consulte otros modelos de programación paralela y diga sus principales características.

5.2.2 El sistema pvm

5.2.2.1 Breve orientación teórica

Introducción

PVM es un conjunto integrado de herramientas y librerías de software, que emulan un marco de trabajo de computación concurrente de propósito general, flexible y heterogéneo. Todo esto sobre un conjunto de computadoras de distintas arquitecturas interconectadas entre sí. El principal objetivo del sistema PVM, es permitir que tal conjunto de máquinas, sean usadas en forma cooperativa para hacer computación concurrente o paralela.

Principios

Resumidamente, los principios sobre los cuales se basa PVM incluyen:

- El conjunto de hosts que conforman la máquina virtual puede ser configurado por el usuario (pool de hosts).
- Acceso translucido al hardware.
- Computación basada en procesos.
- Modelo de pasaje de mensajes explícito.
- Soporte heterogéneo (en cuanto a: arquitecturas de hardware, redes y aplicaciones).
- Soporte para Multiprocesadores.

Partes del sistema

El sistema PVM esta compuesto por dos partes. La primera es un demonio, llamado pvmd3 que reside en todas las máquinas que conforman la máquina virtual.

La segunda parte del sistema es una librería de interfaces de rutinas de PVM, que contiene un repertorio de primitivas necesarias para la cooperación entre las tareas de una aplicación. Tales rutinas pueden ser llamadas por el usuario y permiten realizar pasaje de mensajes, expansión de procesos, coordinación de tareas y modificación de la máquina virtual. Las interfaces están provistas para los lenguajes Fortran y C, siendo implementadas como subrutinas en el primer caso y como funciones en el último.

Todas las tareas en PVM son identificadas por un entero llamado TID (del inglés task identifier), estos enteros son suministrados por el demonio local de PVM.

PVM incluye el concepto de grupos de tareas, para ello provee un conjunto de funciones, que permiten a una tarea - entre otras funciones - unirse o dejar un grupo determinado de tareas

5.2.3 Grupo general de prácticas pvm

5.2.3.1 Práctica I

5.2.3.1.1 Enunciado. Desarrollar una aplicación con PVM formada por dos pequeños programas, entre los cuales se establezca una transferencia de mensajes. Mensaje sugerido: “Hola mundo”, el mensaje debe ir acompañado por:

- El nombre de la maquina que envía el mensaje
- TID (Task Identificator) o identificador de tarea.

5.2.3.1.2 Justificación. Es necesario que el estudiante inicie su experimentación en la programación con PVM desde lo más sencillo, y no por ello menos importante, como lo es la creación de una tarea y el paso de mensajes.

5.2.3.1.3 Objetivos

- Introducir al estudiante en la programación en PVM.
- Entrenar al estudiante en todo lo relacionado con la creación de tareas y de sus respectivos envíos de mensajes.

5.2.3.2 Práctica II

5.2.3.2.1 Enunciado. Crear una aplicación del tipo “Maestro/Esclavo”, donde el proceso maestro cree y dirija algún número de procesos esclavos que cooperen para hacer un trabajo.

Trabajo sugerido: Creación de tres tareas que intercambien mensajes entre si y su resultado se visualizado por pantalla.

5.2.3.2.2 Justificación. En la programación paralela con el fin de crear un sistema distribuido, conocer la forma de coordinar varios procesos esclavos a través de un proceso maestro es de vital importancia.

5.2.3.2.3 Objetivos

- Que el estudiante comprenda y domine la forma de coordinar varios procesos esclavos mediante un proceso maestro.
- Lograr en el estudiante el dominio del concepto Maestro/Esclavo.

5.2.3.3 Práctica III

5.2.3.3.1 Enunciado. Realizar un ejemplo de SPMD (Single Program Multiple Data) que use pvm_siblings para determinar el número de tareas y sus TIDs .

La aplicación debe usar un simple "Token Ring" y paso de mensajes.

5.2.3.3.2 Justificación. Es necesario que el estudiante maneje de forma práctica el modelo SPMD, utilizado por grandes computadoras en cómputos donde se hace necesario la repetición de cálculos en varios conjuntos de datos.

PVM se convierte en una opción económica y viable para hacer la simulación de dichos sistemas. Además la forma de trabajar con "Token Ring" y paso de mensajes es muy utilizado en la sincronización en los sistemas distribuidos.

5.2.3.3.3 Objetivos

- Introducir al estudiante al manejo de forma practica del modelo SPMD, el cual es de vital importancia.
- Lograr que el estudiante comprenda el uso y funcionamiento del mencionado modelo.
- Ilustrar al estudiante de manera practica el concepto de Token Ring y paso de mensajes.

5.2.3.4 Práctica IV

5.2.3.5 Enunciado. Desarrolle una aplicación que ilustre como medir el ancho de banda en una red a través de la utilización del conjunto de librerías que ofrece PVM.

5.2.3.5.1 Justificación. Resulta interesante el uso de PVM en la construcción de aplicaciones de red y más aún cuando estas se usan para obtener información del funcionamiento de la misma red. Además, es una forma de adiestrarse en la programación con PVM.

5.2.3.5.2 Objetivos

- Adiestrar al estudiante en la construcción de aplicaciones de RED, haciendo uso de un nuevo paradigma de programación (paralela).
- Que el estudiante compare el resultado de esta programación paralela con el resultado que el estudiante pudiera obtener con la programación de esta misma practica con un método secuencial.
- Afianzar en el estudiante el manejo de las librerías que ofrece PVM.

6 GUÍA DEL DOCENTE.

6.1 COMUNICACIÓN ENTRE PROCESOS

6.1.1 Cliente/servidor

6.1.1.1 Conceptos y definiciones del modelo c/s: protocolos y conexiones.

6.1.1.1.1 Breve orientación teórica. La comunicación mediante sockets es una interfaz con la capa de transporte (nivel 4) de la jerarquía OSI.

Sin embargo, la interfaz de acceso a la capa de transporte del sistema UNIX no está totalmente aislada de las capas inferiores, por lo que a la hora de trabajar con sockets es necesario conocer algunos detalles sobre esas capas. En concreto, a la hora de establecer una conexión mediante sockets, es necesario conocer la familia o dominio de la conexión y el tipo de conexión.

- Una familia agrupa todos aquellos sockets que comparten características comunes, tales como protocolos, convenios para formar direcciones de red, convenios para formar nombre, etc.
- El tipo de conexión indica el tipo de circuito que se va a establecer entre los dos procesos que se están comunicando. El circuito puede ser virtual (orientado a la conexión) o datagrama (no orientado a conexión). Para establecer un circuito virtual, se realiza una búsqueda de enlaces libres que unan los computadores a conectar. Una vez establecida la conexión, se puede proceder al envío secuencial de los datos, ya que la conexión es permanente. Por el contrario, los datagramas no trabajan con conexiones permanentes. La transmisión por los datagramas es a nivel de paquetes, donde cada paquete puede seguir una ruta distinta y no se garantiza una recepción secuencial de la información.

6.1.1.1.2 Preguntas de concepto

e. ¿Qué es un programa servidor?

Un servidor es un proceso que se está ejecutando en un nodo de la red y que gestiona el acceso a un determinado recurso. El servidor está continuamente esperando peticiones de servicio. Cuando se produce una petición, el servidor despierta y atiende al cliente. Cuando el servicio concluye, el servidor vuelve al estado de espera.

f. ¿Qué es un programa cliente?

Un cliente es un proceso que se ejecuta en el mismo o en diferente nodo y que realiza peticiones de servicio al servidor. Las peticiones están originadas por la necesidad de acceder al recurso que gestiona el servidor.

g. ¿Qué es una familia de sockets?. Mencione algunas de las más conocidas.

Una familia de sockets es la entidad que agrupa todos aquellos sockets que comparten características comunes, tales como protocolos, convenios para formar direcciones de red, convenios para formar nombre, etc.

Las dos familias siguientes suelen estar presentes en todos los sistemas:

AF_UNIX Protocolos internos Unix. Es la familia de sockets empleada para comunicar procesos que se ejecutan en una misma máquina. Esta familia no requiere que esté presente un hardware especial de red

AF_INET Protocolos Internet. Es la familia de sockets que se comunican mediante protocolos, tales como TCP o UDP.

Otras familias menos frecuentes son:

AF_CCITT Norma X.25 del CCITT.

AF_NS Protocolos NS de Xerox.

h. ¿A qué nos referimos al hablar de "tipo de conexión" a la hora de establecer una comunicación entre 2 procesos (sockets)?

El tipo de conexión indica el tipo de circuito que se va a establecer entre los dos procesos que se están comunicando. El circuito puede ser virtual (orientado a la conexión) o datagrama (no orientado a conexión). Para establecer un circuito virtual, se realiza una búsqueda de enlaces libres que unan los computadores a conectar. Una vez establecida la conexión, se puede proceder al envío secuencial de los datos, ya que la conexión es permanente. Por el contrario, los datagramas no trabajan con conexiones permanentes. La transmisión por los datagramas es a nivel de paquetes, donde cada paquete puede seguir una ruta distinta y no se garantiza una recepción secuencial de la información.

6.1.1.1.3 Preguntas de análisis

b. Los servicios y protocolos de red se relacionan directamente con el modelo de comunicación (con base en capas) entre computadores. ¿Para su uso (servicios y protocolos de red) se hace necesario conocer los detalles de implementación de cada una de las capas del modelo de comunicación?. Explique.

No se hace necesario, porque la filosofía de la división por capas de estos sistemas es encapsular, dentro de cada una de ellas, detalles que conciernen sólo a la capa, y

presentársela al usuario como una caja negra con unas determinadas entradas y salidas, de tal forma que el usuario pueda trabajar con ella sin necesidad de conocer sus detalles de implementación.

6.1.1.2 Generalidades del manejo de sockets y esquema de funcionamiento de la filosofía cliente/servidor

6.1.1.2.1 Breve orientación teórica. A la hora de referirse a un nodo de la red, cada protocolo implementa un mecanismo de direccionamiento.

La dirección distingue de forma inequívoca a cada nodo o computador y es utilizada para encaminar los datos desde el nodo origen al nodo destino. La forma de construir direcciones depende de los protocolos que se empleen en la capa de transporte y de red. Hay muchas llamadas al sistema UNIX que necesitan un puntero a una estructura de dirección de socket para trabajar. Esta estructura se define el fichero de cabecera <sys/socket.h> y su forma es la siguiente:

```
struct sockaddr {
  u_short sa_family;  /*Familia de sockets. Se emplean las constantes de la forma
                          AF_***  */
  Char sa_data[14];    /*14 bytes que contiene la dirección. Su significado depende de
                          la familia de sockets que se esté empleando */
};
```

Si usamos una familia que emplea protocolos internet, la forma de las direcciones de red será la definida en el fichero de cabecera <netined/in.h>:

```
struct in_addr {
```

```
  u_long s_addr;  /*32 bits que contienen la identificación de la red y del host. */
};

struct sockaddr_in {
  short sin_family;   /*AF_INET */
  u_short sin_port;   /*16 bits con el número de puerto */
  struct in_addr sin_addr;  /*32 bits con la identificación de la red y del host. */
  char sin_zero[8];  /*8 bytes no usados.*/
};
```

La familia de sockets conocida como UNIX domain emplea direcciones con la forma definida en <sys/un.h>:

```
struct sokaddr_un {
  short sun_family;  /* AF_UNIX */
  char sun_path[108];  /* Path name. */
};
```

Estas direcciones se corresponden en realidad con path names de ficheros y su longitud (110 bytes) es superior a los 16 bytes que de forma estándar tienen las direcciones del resto de familias. Esto es posible debido a que esta familia de sockets se utiliza para comunicar procesos que se están ejecutando bajo el control de una misma máquina, por lo que no necesitan hacer accesos a la red.

Por otra parte, el modelo Cliente/Servidor es el modelo estándar de ejecución de aplicaciones en una red.

Un servidor es un proceso que se está ejecutando en un nodo de la red y que gestiona el acceso a un determinado recurso. Un cliente es un proceso que se ejecuta en el mismo o en diferente nodo y que realiza peticiones de servicio al servidor. Las peticiones están originadas por la necesidad de acceder al recurso que gestiona el servidor.

Las acciones que debe llevar a cabo el programa servidor son las siguientes:

9. Abrir el canal de comunicaciones e informar a la red tanto de la dirección a la que responderá como de su disposición para aceptar peticiones de servicio.
10. Esperar a que un cliente le pida servicio en la dirección que él tiene declarada.
11. Cuando recibe una petición de servicio, si es un servidor interactivo atenderá al cliente. Los servidores interactivos se suelen implementar cuando la respuesta que necesita el cliente es sencilla e implica poco tiempo de proceso. Si el servidor es concurrente creará un proceso mediante fork para que le de servicio al cliente.
12. Volver al punto 2 para esperar nuevas peticiones de servicio.

El programa cliente, por su parte, llevará a cabo las siguientes acciones:

7. Abrir el canal de comunicaciones y conectarse a la dirección de red atendida por el servidor.

8. Enviar al servidor un mensaje de petición de servicio y esperar hasta recibir la respuesta.
9. Cerrar el canal de comunicaciones y terminar la ejecución.

6.1.1.2.2 Preguntas de concepto

e. ¿Para qué sirve la dirección de red de un host?

La dirección distingue de forma inequívoca a cada nodo o computador y es utilizada también para encaminar los datos desde el nodo origen al nodo destino.

f. ¿Existe una sola forma para las direcciones de red?. ¿De qué depende la forma de una dirección de red?

No existen varias formas. Su forma depende del protocolo que se emplee en su construcción.

g. ¿Cuál es la estructura de una dirección de socket para una familia que emplea protocolos Internet?

Si usamos una familia que emplea protocolos internet, la forma de las direcciones de red será la definida en el fichero de cabecera <netined/in.h>:

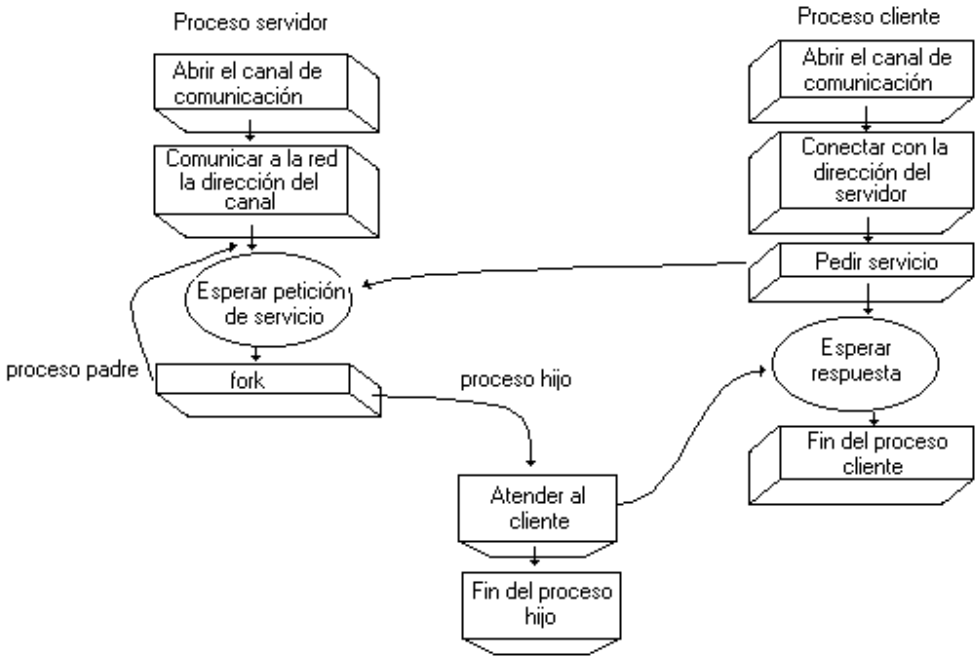
```
struct in_addr {
 u_long s_addr;  /*32 bits que contienen la identificación de la red y del host. */
};

struct sockaddr_in {
 short sin_family;   /*AF_INET */
 u_short sin_port;   /*16 bits con el número de puerto */
 struct in_addr sin_addr;  /*32 bits con la identificación de la red y del host. */
 char sin_zero[8];  /*8 bytes no usados.*/
};
```

h. Muestre de manera gráfica el esquema genérico de flujo de control para los procesos servidores y clientes. Explique.

Las acciones que debe llevar a cabo el programa servidor son las siguientes:

1. Abrir el canal de comunicaciones e informar a la red tanto de la dirección a la que responderá como de su disposición para aceptar peticiones de servicio.
2. Esperar a que un cliente le pida servicio en la dirección que él tiene declarada.
3. Cuando recibe una petición de servicio, si es un servidor interactivo atenderá al cliente. Los servidores interactivos se suelen implementar cuando la respuesta que necesita el cliente es sencilla e implica poco tiempo de proceso. Si el servidor es concurrente creará un proceso mediante fork para que le de servicio al cliente.
4. Volver al punto 2 para esperar nuevas peticiones de servicio.



El programa cliente, por su parte, llevará a cabo las siguientes acciones:

1. Abrir el canal de comunicaciones y conectarse a la dirección de red atendida por el servidor.
2. Enviar al servidor un mensaje de petición de servicio y esperar hasta recibir la respuesta.
3. Cerrar el canal de comunicaciones y terminar la ejecución.

6.1.1.2.3 Preguntas de análisis

c. De un ejemplo práctico de cómo se haría el llenado de una estructura de dirección de un socket para la familia AF_INET y orientado a conexión.

```
/*inicializar estructura de ser_addr*/

printf("Entre la direccion IP del servidor: ");
gets(dir_nodo);
printf("Entre el numero del puerto: ");
gets(buffer);
dir_puerto=atoi(buffer);
ser_addr.sin_family=AF_INET;
ser_addr.sin_addr.s_addr=inet_addr(dir_nodo);
ser_addr.sin_port=htons(dir_puerto);
ser_addr_len=sizeof(ser_addr);
```

d. En un socket no orientado a la conexión ¿se hace necesaria la llamada a la instrucción *connect* del lado del proceso cliente?. Explique su respuesta.

No es necesario, porque *connect* es la respuesta para que *accept* (en el lado del servidor) establezca la conexión entre los procesos y la naturaleza misma de las conexiones del tipo SOCK_DGRAM hace esto innecesario.

6.1.1.3 Métodos de direccionamiento

6.1.1.3.1 Breve orientación teórica. Para que un cliente pueda enviar un mensaje a un servidor, debe conocer la dirección de éste.

Si sólo existe un proceso en ejecución en la máquina destino, el núcleo sabrá que hacer con el mensaje recibido (dárselo al único proceso en ejecución). Sin embargo, ¿qué ocurre si existen varios procesos en ejecución en la máquina destino?

Otro tipo de sistema de direccionamiento envía mensajes a los procesos en vez de a las máquinas. Un esquema común consiste en utilizar nombres con dos partes, para especificar tanto la máquina como el proceso.

Una ligera variación de este esquema de direccionamiento utiliza machine. Local–id en vez de machine.process. El campo local–id es por lo general, un entero aleatorio de 16 o 32 bits (o el siguiente de una serie).

Existe otro método más para la asignación de identificadores a los procesos, el cual consiste en dejar que cada proceso elija el propio identificador de un gran espacio de direcciones dispersas, como el espacio de enteros binarios de 64 bits. Sin embargo, aquí también existe un problema: ¿Cómo sabe el núcleo emisor a cuál máquina enviar el mensaje? En una LAN que soporte transmisiones, el emisor puede transmitir un **paquete especial de localización** con la dirección del proceso destino.

Aunque este esquema es transparente, incluso con ocultamiento, la transmisión provoca una carga adicional en el sistema. Esta carga se evita mediante una máquina adicional para la asociación a alto nivel (es decir, en ASCII) de los nombres de

servicios con las direcciones de las máquinas. Lo anterior se conoce a menudo como **servidor de nombres.**

Un método por completo distinto utiliza un hardware especial. Se deja que los procesos elijan su dirección en forma aleatoria.

6.1.1.3.2 Preguntas de concepto

d. ¿Cuál es el método más simple de direccionamiento que permite el envío de solicitudes en forma inequívoca a un proceso X dentro de una máquina Y?

   Machine.process

e. ¿ En qué consiste el método de direccionamiento de procesos con transmisión?

   En este método el emisor puede transmitir un **paquete especial de localización** con la dirección del proceso destino. Puesto que es un paquete de transmisión, será recibido por todas las máquinas de la red. Todos los núcleos verifican si la dirección es la suya y, en caso que lo sea, regresa un mensaje **aquí estoy** con su dirección en la red (número de máquina). El núcleo emisor utiliza entonces esa dirección y la captura, para evitar el envío de otra transmisión la próxima vez que necesite al servidor.

f. Existe un método de direccionamiento que se basa en un hardware especial. Explique.

   Un método por completo distinto utiliza un hardware especial. Se deja que los procesos elijan su dirección en forma aleatoria. Sin embargo, en vez de localizarlos mediante transmisiones a toda la red, los circuitos de interfaz de la red se diseñan de modo que permitan a los procesos guardar direcciones de procesos

en ellos. Entonces, los marcos usarían direcciones de procesos en vez de direcciones de máquinas. Al recibir cada marco, el circuito de interfaz de la red sólo tendría que examinar el marco para ver si el proceso destino se encuentra en su máquina. En caso afirmativo, aceptaría el marco; en caso negativo no se aceptaría.

6.1.1.3.3 Preguntas de análisis

c. ¿El método de direccionamiento machine.process es ideal para la construcción de un sistema distribuido? Explique.

No es ideal, porque no es transparente, el usuario debe conocer la posición (dirección) del servidor, y la transparencia es uno de los principales objetivos de la construcción de un sistema distribuido.

d. ¿Cuál es la principal ventaja del método de búsqueda de direccionamiento por medio de un servidor de nombres respecto al método de direccionamiento de procesos con transmisión?

El método de direccionamiento de procesos con transmisión, provoca en el sistema una sobrecarga, la cual es eliminada con el método de servidor de nombres, ya que no tiene que enviar ningún paquete a toda la red, sino que busca en el servidor de nombres la dirección de la máquina donde se encuentra dicho proceso y lo envía directamente.

6.1.1.3.4 Práctica

6.1.1.3.4.1 *Practica I.* Desarrollar una aplicación que realice las cuatro operaciones básicas (suma, resta, multiplicación y división),usando socket del tipo AF_INET y protocolo TCP/IP.

Laaplicación debe estar compuesta por un cliente, responsable de tomar la expresión a evaluar y enviarla al servidor y de un servidor, quien hará el cálculo y enviará la respuesta.

6.1.1.3.4.1.1 Justificación. El estudiante debe ver la importancia de conocer y saber manejar los sockets del tipo AF_INET y el protocolo TCP/IP, los cuales son empleados en la implementaciones de muchas aplicaciones .

6.1.1.3.4.1.2 Objetivos

- Con esta práctica se pretende que el estudiante maneje una de las formas de trabajo de la filosofía cliente/serrvidor.
- Establecer una eficaz comunicación entre los sockets en ambos extremos.
- Lograr que el estudiante se adiestre en el desarrollo de aplicaciones con sockets del tipo AF_INET y observe sus características.

6.1.1.4 LLamadas para el manejo de sockets.

6.1.1.4.1 Breve orientación teórica.

Apertura de un punto terminal en un canal. Socket:

La llamada para abrir un canal bidireccional de comunicaciones es ***socket,*** y se usa de la siguiente manera:

socket (af, types, protocol);

Socket crea un punto terminal para conectarse a un canal y devuelve un descriptor. El descriptor de socket devuelto se usará en llamadas posteriores a funciones de la interfaz.

Af (address family) especifica la familia de sockets o familia de direcciones que se desea emplear. Las distintas familias están definidas en el fichero de cabecera <sys/socket.h>. Las dos familias siguientes suelen estar presentes en todos los sistemas:

AF_UNIX Protocolos internos UNIX. Es la familia de sockets empleada para comunicar procesos que se ejecutan en una misma máquina. Esta familia no requiere que esté presente un hardware especial de red

AF_INET Protocolos Internet. Es la familia de sockets que se comunican mediante protocolos, tales como TCP o UDP.

El argumento types indica la semántica de la comunicación para el socket. Puede ser:

SOCK_STREAM socket con un protocolo orientado a conexión.

SOCK_DGRAM socket con un protocolo no orientado a conexión o datagrama.

Protocol especifica el protocolo particular que se va a usar en el socket.

Nombre de un socket. Bind

La llamada bind se utiliza para unir un socket a una dirección de red determinada. Se usa de la siguiente manera:

bind (sfd, addr, addrlen);

Cuando se crea un socket con la llamada socket, se le asigna una familia de direcciones, pero no una dirección particular. Bind hace que el socket de descriptor sfd se una a la dirección de socket específica en la estructura apuntada por addr. Addrlen indica el tamaño de la dirección.

Disponibilidad para recibir peticiones de servicio. Listen

Cuando se abre un socket orientado a conexión, el programa servidor indica que está disponible para recibir peticiones de conexión mediante la llamada a listen. La cual se emplea de la siguiente forma:

listen (sfd, backlog)

Listen habilita una cola asociada al socket descrito por sfd. Esta cola se va a encargar de alojar peticiones de conexión procedentes de los procesos clientes. La longitud de esta cola es la especificada en el argumento backlog. Para que la llamada a listen

tenga sentido, el socket debe ser del tipo SOCK_STREAM (socket orientado a conexión).

Petición de conexión. Connect.

Para que un proceso cliente inicie una conexión con un servidor a través de un socket, es necesario que haga una llamada a connect. Esta función se declara de la forma siguiente:

connect (sfd, addr, addrlen);

sfd es el descriptor del socket que da acceso al canal y addr es un puntero a una estructura que contiene la dirección del socket remoto al que queremos conectarnos. Addrlen es el tamaño en bytes. La estructura de la dirección dependerá de la familia de sockets con la que estemos trabajando.

Aceptación de una conexión. Accept

Los procesos servidores van a leer peticiones de servicios mediante la llamada accept. Llamada:

accept (sfd, addr, addrlen)

Esta llamada se usa con sockets orientados a conexión como el tipo SOCK_STREAM. El argumento sfd es un descriptor de socket creado por una llamada previa a socket y unido a una dirección mediante bind. Accept extrae la

primera petición de conexión que hay en la cola de peticiones pendientes creada con una llamada previa a listen. Una vez extraída la petición de conexión, accept crea un nuevo socket con las mismas propiedades que sfd y reserva un nuevo descriptor de fichero (nsfd) para él.

El socket original (sfd) permanece abierto y puede aceptar nuevas conexiones; sin embargo, el socket recién creado (nsfd) no puede usarse para aceptar más conexiones.

El argumento addr debe apuntar a una estructura local de dirección de socket. La llamada accept rellenará esa estructura con la dirección del socket remoto que pide la conexión. El argumento addrlen debe ser un puntero a int. Inicialmente, debe contener el tamaño en bytes de la estructura de dirección. La función sobreescribirá en addrlen el tamaño real de la dirección leída de la cola de direcciones.

Lectura o recepción de mensajes de un socket.

Una vez que el canal de comunicación entre los procesos servidor y cliente está correctamente inicializado y ambos procesos disponen de un conector (socket) con el canal, contamos con cinco llamadas al sistema para leer datos/mensajes de un socket y otras cinco llamadas para escribir datos/mensajes en él.

Las llamadas para leer datos de un socket son: read, readv, recv, recvfrom, recvmsg.

Escritura o envío de mensaje a un socket.

Las llamadas para escribir datos en un socket son: write, writev, send, sento, sendmsg.

Cierre del canal. Close.

Una vez que un proceso no necesita realizar más accesos a un socket, puede desconectarse del mismo. Para ello, y aprovechando que un socket es tratado sintácticamente como si fuera un fichero, podemos usar la llamada close. Esta llamada va a cerrar el sockets en sus dos sentido (servidor-cliente y cliente-servidor).

6.1.1.4.2 Preguntas de concepto

e. En la primitiva socket(af, type, protocol), ¿qué sucede al hacer cero (0) el parámetro protocol?

La elección del protocolo se deja en manos del sistema.

f. Si se une un socket AF_INET a una dirección y el campo sin_port se inicializó en cero y se olvidó cambiar este valor, por un valor de puerto diferente, ¿qué sucede?

El sistema asigna un número de puerto libre.

g. Si el socket es del tipo SOCK_DGRAM, ¿*connect* especifica la dirección del socket remoto al que se le van a enviar los mensajes? ¿Es necesario?. Explique.

Si, connect especifica la dirección del socket remoto y es necesario en socket del tipo SOCK_DGRAM aunque no se conecte con él.

h. Close( ) cierra un socket en sus dos sentidos (C/S y S/C) consulte una primitiva o instrucción que permita un control más fino a la hora de cerrar y deshabilitar uno de 2 sentidos del socket.

Shutdown(sfd,how), donde sfd es el descriptor del socket y how indica que sentido se ha de deshabilitar:

0 deshabilita la recepción de datos del socket

1 deshabilita el envío de datos a través del socket

2 deshabilita el envío y la recepción de datos. Es equivalente a close().

6.1.1.4.3 Preguntas de análisis

b. ¿Se vería afectada la normal ejecución de un servidor interactivo si a él llegan 2 o más solicitudes de servicio a la vez?

No necesariamente, puesto que si el servidor a través de la llamada a listen especifica un valor, superior a cero, para el argumento backlog con esto estaría definiendo la longitud de una cola de conexiones que daría continuidad al proceso servidor.

6.1.1.4.4 Práctica

1.1.1.1.1 Práctica II. Desarrollar una aplicación cliente/servidor donde el cliente capture y envíe una cadena “Hola mundo” al servidor y este a su vez la reciba y visualice.

Familia AF_INET. Orientado a la conexión. TCP.

6.1.1.4.4.1.1 Justificación. Es necesario que el estudiante experimente el paso de mensajes a través de aplicaciones bajo la filosofía cliente/servidor, a la vez que se introduce en el manejo de primitivas básicas para la manipulación de sockets.

6.1.1.4.4.1.2 Objetivos. Que el estudiante:

- Cree un par de sockets (un cliente y un servidor).
- Aprenda a llenar las estructuras que hacen parte de cada uno de los sockets.(dirección introducida por teclado).
- Establezca una conexión entre los sockets.
- Utilice primitivas para envío y recepción de mensajes.
- Trabaje con la familia de protocolos AF_INET y con sockets orientados a la conexión.

6.1.1.4.4.1.3 Solución

6.1.1.4.4.1.3.1 Código del servidor

```
/*Servidor:  Hola mundo*/

#include "i_addr.h"
#define BUFF_SIZE 200

struct sockaddr_in ser_addr, cli_addr;

void close_sockets();
void resetear(char *);
void init_buffer(char[]);

char buffer[BUFF_SIZE];
int i;

/*Vbles para el manejo de sockets*/

int ok_bind,ok_conect,ok_read,ok_write,ok_list,ok_close,
    ok_sock,id_sock,nid_sock,dir_puerto,clave;
int ser_addr_len,cli_addr_len,msg_size;
char msg_buffer[BUFF_SIZE], dir_nodo[BUFF_SIZE];
```

```
int main (void){

 /*abrir socket*/
 id_sock=socket(AF_INET,SOCK_STREAM,0);
 if (id_sock==-1) {
  printf("\nError abriendo socket\n");
  exit(-1);
 }

 /*inicializar estructura de ser_addr*/
 init_buffer(buffer);
 system("clear");
 printf("Entre la direccion IP del servidor: ");
 gets(dir_nodo);
 printf("Entre el numero del puerto: ");
 gets(buffer);
 dir_puerto=atoi(buffer);

 printf("\nMi primer programa Cliente-Servidor");
 printf("\nCorriendo...\n");

 ser_addr.sin_family=AF_INET;
 ser_addr.sin_addr.s_addr=inet_addr(dir_nodo);
 ser_addr.sin_port=htons(dir_puerto);
 ser_addr_len=sizeof(ser_addr);
```

```
/*publiacion del socket*/
ok_bind=bind(id_sock,&ser_addr,ser_addr_len);
if (ok_bind==-1) {
  printf("\nError publicando el socket\n");
  exit(-1);
}

printf("\nSocket listo. Esperando solicitud del cliente");

/*Definicion del buzon para recibir conexiones*/
ok_list=listen(id_sock,0);
if(ok_list==-1) {
  printf("\nError haciendo listen\n");
  exit(-1);
}

/*aceptando la conexion con el cliente*/
cli_addr_len=sizeof(cli_addr);
nid_sock=accept(id_sock,&cli_addr,&cli_addr_len);
if (nid_sock==-1) {
  printf("\nError aceptando conexion\n");
  exit(-1);
}

while(1) {
```

```
/*inicializa buffer de entrada*/
init_buffer(msg_buffer);

/*lea solicitud del cliente*/
ok_read=read(nid_sock,msg_buffer,sizeof(msg_buffer));
if (ok_read==-1) {
  printf("\nError al leer el buffer en el servidor\n");
  exit(-1);
}

if (msg_buffer[0]==NULL || msg_buffer[0]=='$') {
  /*finalizar servicio por solicitud del cliente*/
  printf("\nfin del servicio por solicitud del cliente\n");
  close_sockets();
  return 0;
}
else {
  printf("\nMensaje recibido desde el cliente: \n %s",msg_buffer);
  printf("\nSolicitud del cliente atendida\n");

  /*prepara el buffer*/
  init_buffer(msg_buffer);
  strcpy(msg_buffer,"Mensaje bien recibido en el servidor");

  /*envia respuesta al cliente*/
```

```
      ok_write=write(nid_sock,msg_buffer,sizeof(msg_buffer));
      if (ok_write==-1) {
        printf("\nError al hacer write desde el servidor\n");
        exit(-1);
      }

      printf("Esperando nueva solicitud...\n\n");
    }
  } /*fin del mq*/
} /*fin del main*/
void resetear(char *cadena) {
  int i;
  i=0;
  cadena=(char *)calloc(BUFF_SIZE,sizeof(char));
  for (i=0;i<BUFF_SIZE;i++) cadena[i]=NULL;
}

void init_buffer (char cadena[]) {
  int i;
  i=0;
  for (i=0;i<BUFF_SIZE;i++) cadena[i]=NULL;
}

void close_sockets() {
  ok_close=close(id_sock);
```

```
  if (ok_close==-1) {
    printf("\nError al cerrar el socket1 \n");
    exit(-1);
  }
  ok_close=close(nid_sock);
  if (ok_close==-1) {
    printf("\nError al cerrar el socket2 \n");
    exit(-1);
  }
}
```

1.1.1.1.1.1.1 Código de archivo de cabecera

```
#ifndef _INT_ADDR_

#define _INT_ADDR_

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/utsname.h>

#include <sys/socket.h>

#include <sys/un.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <string.h>

#include <fcntl.h>


/*#define  PUERTO_SERVIDOR_TCP  5002

#define DIRECCION_NODO_SERVIDOR  "172.16.1.9"*/


#endif
```

6.1.1.4.4.2 Práctica III. Aplicación cliente/servidor. Familia de protocolos AF_UNIX. Orientado a la conexión. TCP. Donde el servidor luego de establecer la conexión envíe una cadena “Hola CUTB” al socket cliente

6.1.1.4.4.2.1 Justificación. Es necesario que el estudiante experimente el paso de mensajes a través de aplicaciones bajo la filosofía cliente/servidor, a la vez que se introduce en el manejo de primitivas básicas para la manipulación de sockets.

6.1.1.4.4.2.2 Objetivos

- Que el estudiante cree un par de sockets (un cliente y un servidor).
- Que el estudiante aprenda a llenar las estructuras que hacen parte de cada uno de los sockets.(dirección introducida por teclado).
- Que el estudiante establezca una conexión entre los sockets.
- Que el estudiante utilice primitivas para envío y recepción de mensajes.
- Que el estudiante trabaje con la familia de protocolos AF_UNIX y con sockets orientados a la conexión.

6.1.1.4.4.2.3 Solución

6.1.1.4.4.2.3.1 Código del servidor

```
/*COMUNICACION MEDIANTE SOCKETS TIPO UNIX ( EN LA MISMA MAQUINA )*/
/**********servidor.c**********/
/*********proceso servidor con sockets AF_UNIX**********/
/*********************************************************/
#include <stdio.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>  /*para sockets UNIX*/
#define PROTOCOLO_DEFECTO 0

main()
{
  int dfServer, dfClient, longServer, longClient;

  struct sockaddr_un dirUNIXServer;
  struct sockaddr_un dirUNIXClient;

  struct sockaddr* puntSockServer;
  struct sockaddr* puntSockClient;
  signal ( SIGCHLD, SIG_IGN ); /*para no crear zombies */

  puntSockServer = ( struct sockaddr* ) &dirUNIXServer;
  longServer = sizeof ( dirUNIXServer );
```

```
puntSockClient = ( struct sockaddr* ) &dirUNIXClient;
longClient = sizeof ( dirUNIXClient );

dfServer = socket ( AF_UNIX, SOCK_STREAM, PROTOCOLO_DEFECTO );
/* se crea un socket UNIX, bidireccional */
dirUNIXServer.sun_family = AF_UNIX;   /* tipo de dominio */
strcpy ( dirUNIXServer.sun_path, "fichero" ); /* nombre */
unlink ( "fichero" );
bind ( dfServer, puntSockServer, longServer ); /* crea el fichero */
/* o sea, nombra el socket */
printf ("\n estoy a la espera \n");
listen ( dfServer, 5 );
while (1)
  {
    dfClient = accept ( dfServer, puntSockClient, &longClient );

    /* acepta la conexion cliente */
    printf ("\n acepto la conexion \n");
    if ( fork() == 0 ) /* crea hijo y envia fichero */
      {
        escribeFichero ( dfClient );
        close ( dfClient ); /* cierra el socket */
        exit ( 0 );
      }
    else
      close ( dfClient ); /* cierra el descriptor cliente */
  }  /* en el padre */
}
```

```
/******** funcion escribeFichero( df ) ***************/
escribeFichero ( int df )
{
  static char* linea1 = "                Hola CUTB, ";
  static char* linea2 = "Programa C-S, AF_UNIX, Servidor Concurrente ";

  write ( df, linea1, strlen (linea1) + 1 );
  write ( df, linea2, strlen (linea2) + 1 );
}
```

1.1.1.1.1.1.2 Código del cliente

```
/*COMUNICACION MEDIANTE SOCKETS TIPO UNIX ( EN LA MISMA MAQUINA ) */
/************************************************************/
/**************cliente.c**********************/
/************************************************************/
/*********proceso cliente con sockets AF_UNIX **********/
/************************************************************/

#include <stdio.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>  /*para sockets UNIX*/
#define PROTOCOLO_DEFECTO 0

/************************************************/
```

```
main()
{
int dfClient, longServer, resultado;
struct sockaddr_un dirUNIXServer;
struct sockaddr* puntSockServer;

puntSockServer = ( struct sockaddr* ) &dirUNIXServer;
longServer = sizeof ( dirUNIXServer );

dfClient = socket ( AF_UNIX, SOCK_STREAM, PROTOCOLO_DEFECTO );
/* se crea un socket UNIX, bidireccional */

dirUNIXServer.sun_family = AF_UNIX; /* tipo de dominio server */
strcpy ( dirUNIXServer.sun_path, "fichero" );  /* nombre server */

do
  {
     resultado = connect ( dfClient, puntSockServer, longServer );
     if ( resultado == -1 ) sleep (1);  /* reintento */
  }
  while ( resultado == -1 );
  leeFichero ( dfClient );   /* lee el fichero */
  close ( dfClient );   /* cierra el socket */
  exit (0);   /* buen fin */
}

/************ leeFichero ( df )****************/
leeFichero ( int df )
```

```
{
  char cad[200];
  while ( leeLinea ( df, cad ) )   /* lee hasta fin de la entrada */
     printf ("%s\n", cad );   /* e imprime lo leido */
}

/************* leeLinea ( df, cad ) ******************/
leeLinea ( int df, char *cad )
{
  int n;
  do
    {
     n = read ( df, cad, 1 );  /*lectura de un caracter */
    }
  while ( n > 0 && *cad++ != NULL );  /*lee hasta NULL o fin entrada */

  return ( n > 0 );  /* devuelve falso si fin de entrada */
}
```

6.1.1.5 Tipos de servidores

6.1.1.5.1 Breve orientación teórica. El modelo Cliente/Servidor es el modelo estándar de ejecución de aplicaciones en una red.

Un servidor es un proceso que se está ejecutando en un nodo de la red y que gestiona el acceso a un determinado recurso. Un cliente es un proceso que se ejecuta en el mismo o en diferente nodo y que realiza peticiones de servicio al servidor. Las peticiones están originadas por la necesidad de acceder al recurso que gestiona el servidor.

El servidor está continuamente esperando peticiones de servicio. Cuando se produce una petición, el servidor despierta y atiende al cliente. Cuando el servicio concluye, el servidor vuelve al estado de espera. De acuerdo con la forma de prestar el servicio, podemos considerar dos tipos de servidores:

- Servidores interactivos: El servidor no solo recoge la petición de servicio, sino que él mismo se encarga de atenderla.

- Servidores concurrentes: El servidor recoge cada una de las peticiones de servicio y crea otros procesos para que se encarguen de atenderlas. Este tipo de servidores solo es aplicables en sistemas multiprocesos, como es UNIX. La ventaja que tiene este tipo de servicio es que el servidor puede recoger peticiones a muy alta velocidad, porque está descargado de la tarea de atención del cliente.

6.1.1.5.2 Preguntas de concepto

b. ¿Cuáles son los tipos de servidores?. Explique sus diferencias.

Existen dos tipos de servidores, interactivos y concurrentes.

Servidores interactivos Vs. Servidores concurrentes	
Recogen una sola petición de servicio al tiempo	Recoge varias peticiones de servicio al tiempo
Él mismo procesa y responde la petición de servicio	Crea procesos hijos para cada uno de las peticiones y serán ellos quienes las procesen y den respuesta
Lentos	Rápidos
Tiempos de espera largos	Tiempos de espera cortos

6.1.1.5.3 Preguntas de análisis

c. ¿Cuándo se recomienda el uso de servidores interactivos?

Cuando la solicitud de servicio sea poco frecuente y el tiempo de servicio sea corto.

d. Se requiere implantar una aplicación C/S en un sistema donde el tiempo de atención al cliente es variable. Se necesita de su opinión: ¿Qué tipo de servidor recomienda?

Un servidor del tipo concurrente. Porque optimiza el tiempo de respuesta de la aplicación y muestra al sistema como un sistema ágil.

6.1.1.5.4 Práctica

6.1.1.5.4.1 Práctica IV. Dado un programa cliente/servidor con servidor del tipo interactivo, haga su equivalente en un tipo de servidor concurrente.

6.1.1.5.4.1.1 Justificación. Es necesario que el estudiante aprecie un ejemplo de cada uno de los tipos de servidores, los compare y confronte su análisis con lo aportado por la teoría.

A la vez que el estudiante aprenderá aspectos básicos de los tipos de servidores de la filosofía cliente/servidor, este pondrá en practica una parte de lo referente al manejo de procesos (tema de vital importancia dentro de los sistemas operativos en general).

6.1.1.5.4.1.2 Objetivos

- Que el estudiante con base en la practica observe las principales ventajas y desventajas de cada uno de los tipos de servidores.
- Que aprenda la manera de construir y manipular procesos hijos a través de la instrucción *fork*, la creación de una cola de procesos, eliminación de procesos, etc.

6.1.1.5.4.1.3 Código de la aplicación planteada para el alumno (servidor interactivo)

6.1.1.5.4.1.3.1 Código del servidor

```
/*Servidor: Cifrador de Julio Cesar */

#include "i_addr.h"
#define BUFF_SIZE 200

struct sockaddr_in ser_addr, cli_addr;

void close_sockets();
void resetear(char *);
void init_buffer(char[]);

char *mensaje,*ciphertext;
char buffer[BUFF_SIZE];
char alfabeto[28]="abcdefghijklmn¤opqrstuvwxyz ",
     cipher[200],soda[1];
int i,puente;

/*Vbles para el manejo de sockets*/

int ok_bind,ok_conect,ok_read,ok_write,ok_list,ok_close,
    ok_sock,id_sock,nid_sock,dir_puerto,clave;
int ser_addr_len,cli_addr_len,msg_size;
char msg_buffer[BUFF_SIZE], dir_nodo[BUFF_SIZE];
int main (void){
```

```
/*abrir socket*/
id_sock=socket(AF_INET,SOCK_STREAM,0);
if (id_sock==-1) {
  printf("\nError abriendo socket\n");
  exit(-1);
}

/*inicializar estructura de ser_addr*/
init_buffer(buffer);
system("clear");
printf("Entre la direccion IP del servidor: ");
gets(dir_nodo);
printf("Entre el numero del puerto: ");
gets(buffer);
dir_puerto=atoi(buffer);

printf("\nAplicacion servidora de Alg de cifrado de Julio Cesar");
printf("\nEntre la clave para descifrar mensaje: "); gets(buffer);
clave=atoi(buffer);

ser_addr.sin_family=AF_INET;
ser_addr.sin_addr.s_addr=inet_addr(dir_nodo);
ser_addr.sin_port=htons(dir_puerto);
ser_addr_len=sizeof(ser_addr);

/*publiacion del socket*/
ok_bind=bind(id_sock,&ser_addr,ser_addr_len);
```

```
if (ok_bind==-1) {
  printf("\nError publicando el socket\n");
  exit(-1);
}

printf("\nSocket listo. Esperando solicitud del cliente");

/*Definicion del buzon para recibir conexiones*/
ok_list=listen(id_sock,0);
if(ok_list==-1) {
  printf("\nError haciendo listen\n");
  exit(-1);
}

/*aceptando la conexion con el cliente*/
cli_addr_len=sizeof(cli_addr);
nid_sock=accept(id_sock,&cli_addr,&cli_addr_len);
if (nid_sock==-1) {
  printf("\nError aceptando conexion\n");
  exit(-1);
}

mensaje=(char *)calloc(BUFF_SIZE,sizeof(char));
ciphertext=(char *)calloc(BUFF_SIZE,sizeof(char));

while(1) {
  /*inicializa buffer de entrada*/
  init_buffer(msg_buffer);
```

```
/*lea solicitud del cliente*/
ok_read=read(nid_sock,msg_buffer,sizeof(msg_buffer));
if (ok_read==-1) {
  printf("\nError al leer el buffer en el servidor\n");
  exit(-1);
}
if (msg_buffer[0]!=NULL) {
    for (i=0;i<=200;i++)
      cipher[i]=NULL;
    for (i=0;i<strlen(msg_buffer);i++) {
      ciphertext[i]=cipher[i]=msg_buffer[i];
    }
    ciphertext[strlen(msg_buffer)]=cipher[strlen(msg_buffer)]=NULL;
}
else
  resetear(ciphertext);

if (msg_buffer[0]==NULL || msg_buffer[0]=='$') {
  /*finalizar servicio por solicitud del cliente*/
  printf("\nfin del servicio por solicitud del cliente\n");
  close_sockets();
  return 0;
}
else {
  init_buffer(mensaje);
  resetear(mensaje);
  printf("\nCiphertext recibido desde el cliente: %s",msg_buffer);
```

```
      for (i=0;ciphertext[i];i++) {
        soda[0]=cipher[i];
        puente=strcspn(alfabeto,&soda[0]);
        if (puente-clave < 0) {
          mensaje[i]=alfabeto[(strcspn(alfabeto,&soda[0])-clave+28)%28];
        }
        else {
          mensaje[i]=alfabeto[(strcspn(alfabeto,&soda[0])-clave)%28];
        }
      }
      printf("\nTexto plano: %s",mensaje);
      printf("\nSolicitud del cliente atendida\n");

      /*prepara el buffer*/
      init_buffer(msg_buffer);
      strcpy(msg_buffer,"Todo listo aca en el servidor");
       /*envia respuesta al cliente*/
      ok_write=write(nid_sock,msg_buffer,sizeof(msg_buffer));
      if (ok_write==-1) {
        printf("\nError al hacer write desde el servidor\n");
        exit(-1);
      }

      printf("Esperando nueva solicitud...\n\n");
     }
   } /*fin del mq*/
} /*fin del main*/
void resetear(char *cadena) {
```

```
  int i;
  i=0;
  cadena=(char *)calloc(BUFF_SIZE,sizeof(char));
  for (i=0;i<BUFF_SIZE;i++) cadena[i]=NULL;
}

void init_buffer (char cadena[]) {
  int i;
  i=0;
  for (i=0;i<BUFF_SIZE;i++) cadena[i]=NULL;
}

void close_sockets() {
  ok_close=close(id_sock);
  if (ok_close==-1) {
    printf("\nError al cerrar el socket1 \n");
    exit(-1);
  }
  ok_close=close(nid_sock);
  if (ok_close==-1) {
    printf("\nError al cerrar el socket2 \n");
    exit(-1);
  }
}
```

6.1.1.5.4.1.3.2 Código del cliente

```
/*Cliente para cifrador de Julio Cesar*/

#include "i_addr.h"
#define BUFF_SIZE 200



struct sockaddr_in ser_addr, cli_addr;
char *cap_mensaje(), *mensaje, *ciphertext;
char alfabeto[28]="abcdefghijklmn¤opqrstuvwxyz ",
     mens[200],soda[1];
char msg_buffer[BUFF_SIZE], buffer[BUFF_SIZE],dir_nodo[BUFF_SIZE],espera;

void resetear();
void init_buffer(char[]);

/*vbles para manejo de sockets*/
int dir_puerto,ok_bind,ok_conect,ok_read,ok_write,ok_close,
    id_sock,nid_sock,ser_addr_len,cli_addr_len,msg_size,opc,i,clave;

int main(void) {
  /*abrir socket*/
  id_sock=socket(AF_INET,SOCK_STREAM,0);
  if (id_sock==-1) {
    printf("\nError abriendo socket\n");
    exit(-1);
```

```
}
/*captura de la info de la maquina*/
init_buffer(buffer);
system("clear");
printf("Entre direccion IP del servidor: ");  gets(dir_nodo);
printf("Entre el numero del puerto: ");gets(buffer);
dir_puerto=atoi(buffer);

/*inicializar estructura ser_addr*/
ser_addr.sin_family=AF_INET;
ser_addr.sin_addr.s_addr=inet_addr(dir_nodo);
ser_addr.sin_port=htons(dir_puerto);

/*peticion de conxion al servidor*/
ok_conect=connect(id_sock,&ser_addr,sizeof(ser_addr));
if (ok_conect==-1) {
  printf("\nError al conectarse con el servidor\n");
  exit(-1);
}

/*cuerpo del cliente*/
opc=1;
mensaje=(char *)calloc(BUFF_SIZE,sizeof(char));
ciphertext=(char *)calloc(BUFF_SIZE,sizeof(char));
while (opc!=3) {
  system("clear");
  printf("          Cifrador de Julio Cesar\n");
  printf("          Modelo Cliente-Servidor\n\n\n");
```

```
printf("Implementado por:\n");
printf("               Efrain Ortiz.\n");
printf("               Gabriel Oliveros V.\n\n\n\n");
printf("1. Captura de mensaje y clave\n");
printf("2. Cifrar y enviar al servidor\n");
printf("3. Terminar cliente y servidor\n\n");
printf("Escoga opcion==> ");
gets(buffer);
opc=atoi(buffer);
if (opc==1) {
  resetear(mensaje);
  mensaje=cap_mensaje();
  for (i=0;i<=200;i++) mens[i]=NULL;
  for (i=0;i<=200;i++) mens[i]=mensaje[i];
  printf("Digite la clave: ");  gets(buffer);
  clave=atoi(buffer);
}
if (opc==2) {
  init_buffer(ciphertext);
  resetear(ciphertext);
  for (i=0;mensaje[i];i++) {
    soda[0]=mens[i];
    ciphertext[i]=alfabeto[(strcspn(alfabeto,&soda[0])+clave)%28];
  }

  /*enviar ciphertext al servidor*/
  msg_size=sizeof(ciphertext);
  strcpy(msg_buffer,ciphertext);
```

```
  ok_write=write(id_sock,msg_buffer,sizeof(msg_buffer));
  if (ok_write==-1) {
    printf("\nError al enviar ciphertexto\n");
    exit(-1);
  }

  /*esperar respuesta del servidor*/
  ok_read=read(id_sock,msg_buffer,sizeof(msg_buffer));
  if (ok_read==-1) {
    printf("\nError al hacer read\n");
    exit(-1);
  }

  /*resultados en el cliente*/
  printf("\nTexto plano  : %s",mensaje);
  printf("\nValor de la clave: %d",clave);
  printf("\nTexto cifrado: %s\n",ciphertext);
  printf("%s\n",msg_buffer);
  printf("Presione cualquier tecla para continuar...");
  espera=getchar();
 } /*fin de opc2*/
 if (opc==3) {
  ok_close=close(id_sock);
  if (ok_close==-1) {
    printf("\nError al hacer close\n");
    exit(-1);
  }
  printf("\nFin del cliente\n");
```

```
      exit(0);
    }
  } /*fin del Mq*/
} /*fin del main*/

void resetear (char *cadena) {
  int i;
  i=0;
  cadena=(char *)calloc(BUFF_SIZE,sizeof(char));
  for (i=0;i<BUFF_SIZE;i++) cadena[i]=NULL;
}

void init_buffer (char cadena[]) {
  int i;
  i=0;
  for (i=0;i<BUFF_SIZE;i++) cadena[i]=NULL;
}

char *cap_mensaje() {
  char *buffer;
  buffer=(char *)calloc(BUFF_SIZE,sizeof(char));
  resetear(buffer);
  printf("\nEntre mensaje: "); gets(buffer);
  return buffer;
}
```

6.1.1.5.4.1.3.3 Código del archivo de cabecera (i_addr.h)

```
#ifndef _INT_ADDR_

#define _INT_ADDR_

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/utsname.h>

#include <sys/socket.h>

#include <sys/un.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <string.h>

#include <fcntl.h>


#endif
```

6.1.1.5.4.1.4 Solución (servidor concurrente)

Código de la aplicación servidora planteada para el docente (servidor concurrente)

Servidor concurrente: Cifrador de Julio Cesar

```
#include "i_addr.h"
#define BUFF_SIZE 200

struct sockaddr_in ser_addr, cli_addr;

void close_sockets();
void resetear(char *);
void init_buffer(char[]);

char *mensaje,*ciphertext;
char buffer[BUFF_SIZE];
char alfabeto[28]="abcdefghijklmn¤opqrstuvwxyz ",
    cipher[200],soda[1];
int i,puente,pid;

/*Vbles para el manejo de sockets*/

int ok_bind,ok_conect,ok_read,ok_write,ok_list,ok_close,
```

```
   ok_sock,id_sock,nid_sock,dir_puerto,clave;
int ser_addr_len,cli_addr_len,msg_size;
char msg_buffer[BUFF_SIZE], dir_nodo[BUFF_SIZE];

int main (void){

 /*abrir socket*/
 id_sock=socket(AF_INET,SOCK_STREAM,0);
 if (id_sock==-1) {
  printf("\nError abriendo socket\n");
  exit(-1);
 }

 /*inicializar estructura de ser_addr*/
 init_buffer(buffer);
 system("clear");
 printf("Entre la direccion IP del servidor: ");
 gets(dir_nodo);
 printf("Entre el numero del puerto: ");
 gets(buffer);
 dir_puerto=atoi(buffer);

 printf("\nAplicacion servidora de Alg de cifrado de Julio Cesar");
 printf("\nEntre la clave para descifrar mensaje: "); gets(buffer);
 clave=atoi(buffer);

 ser_addr.sin_family=AF_INET;
 ser_addr.sin_addr.s_addr=inet_addr(dir_nodo);
```

```
ser_addr.sin_port=htons(dir_puerto);
ser_addr_len=sizeof(ser_addr);

/*publiacion del socket*/
ok_bind=bind(id_sock,&ser_addr,ser_addr_len);
if (ok_bind==-1) {
  printf("\nError publicando el socket\n");
  exit(-1);
}

printf("\nSocket listo. Esperando solicitud del cliente");

/*Definicion del buzon para recibir conexiones*/
ok_list=listen(id_sock,5);          /*** cambios al listen ***/
if(ok_list==-1) {
  printf("\nError haciendo listen\n");
  exit(-1);
}

          /*** bucle para hacer lecturas de conexion ***/
for (;;) {



/*aceptando la conexion con el cliente*/
cli_addr_len=sizeof(cli_addr);
nid_sock=accept(id_sock,&cli_addr,&cli_addr_len);
if (nid_sock==-1) {
```

```
    printf("\nError aceptando conexion\n");
    exit(-1);
  }

/********************* aqui cambios  para el fork ******************/

if ((pid=fork())==-1)
  printf("\nError en el fork\n");
else if (pid==0) {

   close(id_sock);

  mensaje=(char *)calloc(BUFF_SIZE,sizeof(char));
  ciphertext=(char *)calloc(BUFF_SIZE,sizeof(char));

  while(1) {
    /*inicializa buffer de entrada*/
    init_buffer(msg_buffer);

    /*lea solicitud del cliente*/
    ok_read=read(nid_sock,msg_buffer,sizeof(msg_buffer));
    if (ok_read==-1) {
      printf("\nError al leer el buffer en el servidor\n");
      exit(-1);
    }
    if (msg_buffer[0]!=NULL) {
        for (i=0;i<=200;i++)
          cipher[i]=NULL;
```

```
    for (i=0;i<strlen(msg_buffer);i++) {
      ciphertext[i]=cipher[i]=msg_buffer[i];
    }
    ciphertext[strlen(msg_buffer)]=cipher[strlen(msg_buffer)]=NULL;
}
else
  resetear(ciphertext);

if (msg_buffer[0]==NULL || msg_buffer[0]=='$') {
  /*finalizar servicio por solicitud del cliente*/
  printf("\nfin del servicio por solicitud de un cliente\n");
  close_sockets();
  return 0;
}
else {
 init_buffer(mensaje);
  resetear(mensaje);
  printf("\nCiphertext recibido desde el cliente: %s",msg_buffer);
  for (i=0;ciphertext[i];i++) {
    soda[0]=cipher[i];
    puente=strcspn(alfabeto,&soda[0]);
    if (puente-clave < 0) {
      mensaje[i]=alfabeto[(strcspn(alfabeto,&soda[0])-clave+28)%28];
    }
    else {
      mensaje[i]=alfabeto[(strcspn(alfabeto,&soda[0])-clave)%28];
    }
  }
```

```
      printf("\nTexto plano: %s",mensaje);
      printf("\nSolicitud del cliente atendida\n");

      /*prepara el buffer*/
      init_buffer(msg_buffer);
      strcpy(msg_buffer,"Todo listo aca en el servidor");

      /*envia respuesta al cliente*/
      ok_write=write(nid_sock,msg_buffer,sizeof(msg_buffer));
      if (ok_write==-1) {
        printf("\nError al hacer write desde el servidor\n");
        exit(-1);
      }

      printf("Esperando nueva solicitud...\n\n");

    }
  } /*fin del mq*/

  /*close(nid_sock);*/
  exit(0);
} /* fin del else */
/*************** hasta aqui cambios para el fork ********************/
 } /* fin del bucle de lectura de peticiones */
} /*fin del main*/
```

```
void resetear(char *cadena) {
  int i;
  i=0;
  cadena=(char *)calloc(BUFF_SIZE,sizeof(char));
  for (i=0;i<BUFF_SIZE;i++) cadena[i]=NULL;
}

void init_buffer (char cadena[]) {
  int i;
  i=0;
  for (i=0;i<BUFF_SIZE;i++) cadena[i]=NULL;
}

void close_sockets() {
  /*ok_close=close(id_sock);
  if (ok_close==-1) {
    printf("\nError al cerrar el socket1 \n");
    exit(-1);
  }*/
  ok_close=close(nid_sock);
  if (ok_close==-1) {
    printf("\nError al cerrar el socket2 \n");
    exit(-1);
  }
}
```

6.1.1.6 Primitivas de transferencia y recepción de mensajes

6.1.1.6.1 Breve orientación teórica. En las **primitivas de bloqueo** (a veces llamadas **primitivas síncronas**) cuando un proceso llama a **send,** especifica un destino y buffer dónde enviar ese destino.

Mientras se envía el mensaje, el proceso emisor se bloque (es decir, se suspende). La instrucción que sigue a la llamada a send no se ejecuta sino hasta que el mensaje se envía en su totalidad.

Una alternativa a las primitivas con bloqueo son las **primitivas sin bloqueo** (a veces llamadas **primitivas asíncronas**). Si send no tiene bloqueo, regresa de inmediato el control a quien hizo la llamada, antes de enviar el mensaje.

Así como los diseñadores de sistemas pueden elegir entre las primitivas con o sin bloqueo, también pueden elegir entre las primitivas almacenadas en buffer o no almacenadas.

Los mensajes se pueden perder, lo cual afecta la semántica del modelo de transferencia de mensajes. Supongamos que se utilizan las primitivas por bloqueo. Cuando un cliente envía un mensaje, se le suspende hasta que el mensaje ha sido enviado. Sin embargo, cuando vuelve a iniciar, no existe garantía alguna acerca de la entrega del mensaje.

Existen tres enfoques de este problema. El primero consiste en volver a definir la semántica de **send** para hacerla no confiable. El sistema no da garantía alguna acerca

de la entrega de los mensajes. La implantación de una comunicación confiable se deja por completo en manos de los usuarios.

El segundo método exige que el núcleo de la maquina receptora envíe un reconocimiento a la maquina emisora. Sólo cuando se reciba este reconocimiento, el núcleo emisor liberará el proceso usuario (cliente). El reconocimiento va de un núcleo al otro ; ni el cliente ni el servidor ven alguna vez un reconocimiento. De la misma forma que la solicitud de un cliente a un servidor es reconocida por el núcleo del servidor, la respuesta del servidor es reconocida por el núcleo del cliente. Así una solicitud de respuesta consta de cuatro mensajes.

El tercer método aprovecha el hecho de que la comunicación cliente/servidor se estructura como solicitud del cliente al servidor, seguida de una respuesta del servidor al cliente.

6.1.1.6.2 Preguntas de concepto

g. Explique la diferencia principal entre las primitivas con bloqueo y las primitivas sin bloqueo.

La principal diferencia radica en que en las primitivas sin bloqueo el proceso emisor puede continuar su cómputo en forma paralela con la transmisión del mensaje, en vez de tener inactivo al CPU(suponiendo que ningún otro proceso sea ejecutable), lo que no ocurre con las primitivas con bloqueo.

h. Enuncie las ventajas y desventajas de las primitivas bloqueantes y no bloqueantes.

La ventaja de las primitivas bloqueantes es que el emisor sabe si el mensaje fue enviado o no y el servidor si fue recibido o no al permanecer bloqueados hasta que culmine la operación.

La principal ventaja de las primitivas sin bloqueo radica el proceso emisor puede continuar su cómputo en forma paralela con las transmisión del mensaje, en vez de tener inactivo al CPU. Sin embargo, la ventaja de desempeño que ofrecen las primitivas sin bloqueo se ve afectada por una seria desventaja: el emisor no puede modificar el buffer de mensajes sino hasta que el mensaje haya sido enviado. Las consecuencias de que el proceso escriba sobre el mensaje durante la transmisión son demasiado graves. Peor aún, el proceso emisor no tiene idea de cuando termine la transmisión, por lo que no sabe cuándo será seguro volver a utilizar el buffer.

i. Existen dos formas para corregir el principal problema que presentan las primitivas sin bloque. Explíquelas, enuncie sus ventajas y desventajas.

La primera solución es que el núcleo copie el mensaje a un buffer interno del núcleo y que entonces permita el proceso que continúe. Desde el punto de vista del emisor, este esquema es el mismo que el de una llamada con bloqueo: tan pronto como recupera el control, es libre de volver a utilizar el buffer. Por supuesto, el mensaje no ha sido enviado todavía, pero el emisor no se preocupa por este hecho. La desventaja del método es que cada mensaje de salida debe ser copiado desde el espacio del usuario al espacio del núcleo. Con muchas interfaces de red, de todas formas el mensaje deberá copiarse posteriormente a un buffer de transmisión en hardware, de modo que, en esencia, la primera copia se desperdicia. La copia adicional puede reducir el desempeño del sistema en forma considerable.

La segunda solución es interrumpir al emisor cuando se envíe el mensaje, para informarle que el buffer está de nuevo disponible. No se requiere de una copia, lo que ahorra tiempo, pero las interrupciones a nivel usuario hacen que la programación sea truculenta, difícil y sujeta a condiciones de competencia, lo que la hace irreproducible. La mayoría de los expertos coinciden en que, a pesar de que este método es muy eficiente y permite un máximo paralelismo, las desventajas tienen mayor peso sobre las ventajas: es difícil escribir en forma correcta los programas que se basan en interrupciones y es casi imposible depurarlo cuando están incorrectos.

j. Describa el funcionamiento de las primitivas almacenadas en buffers y las no almacenadas.

Las primitivas descritas hasta ahora son en esencia **primitivas no almacenadas.** Esto significa que una dirección se refiere a un proceso específico. Una llamada receive (addr, &m) indica al núcleo de la máquina donde se ejecuta ésta que el proceso que llamó escucha a la dirección addr y que está preparada para recibir un mensaje enviado a esa dirección. Se dispone de un buffer de mensajes, al que apunta m, con el fin de capturar el mensaje por llegar. Cuando el mensaje llega, el núcleo receptor lo copia al buffer y elimina el bloqueo del proceso receptor. El uso de una dirección para hacer referencia a un proceso específico.

Una forma conceptual sencilla de enfrentar este manejo de los buffers es definir una nueva estructura de datos llamada **buzón.** Un proceso interesado en recibir mensajes le indica al núcleo que cree un buzón para él y especifica una dirección en la cual busca los paquetes de la red. Así, todos los mensajes que lleguen con esa dirección se colocan en el buzón. La llamada a **receive** elimina ahora un mensaje del buzón o se bloquea (si se utilizan primitivas con bloqueo) si no hay mensajes presentes. Esta técnica se conoce a menudo como **primitiva con almacenamiento en buffers.**

k. ¿Cuál es el principal problema que presentan las primitivas no almacenadas? ¿Cómo se puede solucionar este problema?. Explique.

Este esquema funciona bien, mientras el servidor llame a receive antes de que el cliente llame a send. La llamada a receive es el mecanismo que indica al núcleo del servidor la dirección que utiliza el servidor y la posición donde colocar el mensaje que está por llegar. El problema surge cuando send se lleva a cabo antes de receive. ¿Cómo sabe el núcleo del servidor cuál de sus procesos (si existe alguno) utiliza la dirección en el mensaje recién llegado? ¿Cómo sabe dónde copiar el mensaje? La respuesta es sencilla: no lo sabe.

Una estrategia de implantación consiste sólo en descartar el mensaje, dejar que el cliente espere y confiar en que el servidor llame a receive antes de que el cliente vuelva a transmitir. Este método se puede implantar con facilidad, pero con algo de mala suerte, el cliente (o más probable, el núcleo del cliente) deberá intentar varias veces antes de tener éxito. Peor aún, si fracasa un número suficiente de intentos consecutivos, el núcleo del cliente se podría dar por vencido y concluir erróneamente que el servidor se ha descompuesto o que la dirección no es válida.

El segundo método para enfrentar este problema es que el núcleo receptor mantenga pendientes los mensajes por un instante, sólo para prevenir que un **receive** adecuado se realice en poco tiempo. Siempre que llegue un mensaje "no deseado", se inicia un cronómetro. Si el tiempo expira antes de que ocurra un **receive** apropiado, el mensaje se descarta.

Aunque este método reduce la probabilidad de que un mensaje se pierda, presenta el problema de almacenar y manejar los mensajes que van llegando en forma prematura. Se necesitan los buffers y tienen que ser asignados, liberados y en

general manejados. Una forma conceptual sencilla de enfrentar este manejo de los buffers es definir una nueva estructura de datos llamada **buzón.** Un proceso interesado en recibir mensajes le indica al núcleo que cree un buzón para él y especifica una dirección en la cual busca los paquetes de la red. Así, todos los mensajes que lleguen con esa dirección se colocan en el buzón. La llamada a **receive** elimina ahora un mensaje del buzón o se bloquea (si se utilizan primitivas con bloqueo) si no hay mensajes presentes .

l. Cuando un cliente envía un mensaje, el servidor lo recibirá. Los mensajes se pueden perder, lo cual afecta la semántica del modelo de transferencia de mensajes. ¿Cuántos métodos existen para solucionar este problema?. Explique.

Existen tres enfoques de este problema. El primero consiste en volver a definir la semántica de **send** para hacerla no confiable. El sistema no da garantía alguna acerca de la entrega de los mensajes. La implantación de una comunicación confiable se deja por completo en manos de los usuarios.

El segundo método exige que el nucleo de la maquina receptora envíe un reconocimiento a la maquina emisora. Sólo cuando se reciba este reconocimiento, el núcleo emisor liberará el proceso usuario (cliente). El reconocimiento va de un núcleo al otro ; ni el cliente ni el servidor ven alguna vez un reconocimiento. De la misma forma que la solicitud de un cliente a un servidor es reconocida por el núcleo del servidor, la respuesta del servidor es reconocida por el núcleo del cliente. Así una solicitud de respuesta consta de cuatro mensajes .El tercer método

aprovecha el hecho de que la comunicación cliente-servidor se estructura como solicitud del cliente al servidor, seguida de una respuesta del servidor al cliente. En este método, el cliente se bloquea después de enviar un mensaje. El núcleo del servidor no envía de regreso un reconocimiento sini que la misma respuesta funciona como tal. Así el emisor permanece bloqueado hasta que regresa la respuesta. Si tarda demasiado, el núcleo puede enviar de nuevo la solicitud para protegerse de la posibilidad de pérdida del mensaje.

6.1.1.6.3 Preguntas de análisis

e. En muchos sistemas de comunicación, las llamadas a *send* inician un cronómetro para protegerse contra la suspención indefinida del cliente si el servidor falla. Suponga que se implanta un sistema tolerante a fallas mediante varios procesadores para todos los cliente y los servidores, de forma que la probabilidad de falla es nula. ¿Cree usted que sería seguro eliminar los tiempos de espera en este sistema?

f. Si se utiliza la comunicación con almacenamiento en buffers, se dispone por lo general de una primitiva para que los usuarios creen buzones. ¿Por qué no se especifica que esta primitiva exija el tamaño del buzón?

   No se especifica el tamaño del buzon, ya que este es variable y depende del tamaño del mensaje.

g. Utilizando las primitivas sin bloqueo. ¿ Qué ocurriría si el emisor no puede modificar su buffers de mensajes sino hasta que el mensaje haya sido enviado?

El proceso escribiría sobre el mensaje durante la transmisión; esto es demasiado grave. Peor aún, el proceso emisor no tiene idea de cuándo termine la transmisión, por lo que no sabe cuándo será seguro volver a utilizar el buffer. Es difícil que evite utilizarlo por siempre.

h. En las primitivas no almacenadas en buffers el esquema funciona bien mientras el servidor llame a *receive* antes de que el cliente llame a *send.* ¿ Qué pasaría si el cliente llama a *send* antes de que el servidor llame a *receive*?

El núcleo del servidor no sabría cuál de sus procesos (si existe alguno) utiliza la dirección en el mensaje recién llegado ni donde copiar el mensaje.

6.1.1.7 Miscelánea de llamadas y funciones

6.1.1.7.1 Breve orientación teórica.

Nombres de un socket. Getsockname, getpeername.

Para determinar las direcciones de los procesos conectados a un socket, se utilizan las llamadas getsockname y getpeername. Estas llamadas tienen aplicación en sockets orientados a conexión y se usan de la siguiente forma:

```
getsockname (sfd, addr, addrlen);
getpeername (sfd, addr, addrlen);
```

Nombre del nodo actual. Gethostname.

Para determinar el nombre oficial que tiene un nodo dentro de la red, podemos usar la llamada gethostname, que se usa como sigue:

```
gethostname (hostname, size);
```

6.1.1.7.2 Preguntas de concepto.

c. Explique el funcionamiento de las primitivas getsockname y getpeername.

Para determinar las direcciones de los procesos conectados a un socket, se utilizan las llamadas getsockname y getpeername. Estas llamadas tienen aplicación en sockets orientados a conexión y se declaran de la siguiente forma:

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

int getsockname (sfd, addr, addrlen)
int sfd;
void *addr;
int *addrlen;

int getpeername (sfd, addr, addrlen)
int sfd;
void *addr;
int addrlen;
```

Getsockname devuelve sobre la estrutura apuntada por addr la dirección del socket cuyo descriptor coincide con sfd. Getpeername devuelve sobre la estructura apuntada por addr la dirección del socket que se encuentra conectado al socket descrito por sfd. En ambas llamadas, addrlen le indica a la función cual es el

tamaño en bytes de la estructura apuntada por addr, y al salir de la función que contiene el tamaño real de la dirección.

d. ¿A través de que primitivas se puede conseguir el nombre del nodo actual?. Explique.

Para determinar el nombre oficial que tiene un nodo dentro de la red, podemos usar la llamada gethostname, que se declara como sigue:

```
#include <unistd.h>

int gethostname (hostname, size)
char *hostname;
size_t size;
```

Gethostname devuelve en el array apuntado por hostname el nombre oficial de host del computador que hace la llamada. Size indica la longitud del array hostname. Esta llamada puede implementarse a partir de la llamada uname, que es más genérica.

6.1.1.7.3 Práctica

6.1.1.7.3.1 Práctica V. Aplicación cliente/servidor con sockets del tipo

SOCK_STREAM, familia de protocolos AF_INET.

Donde, luego de comunicarse los dos sockets(cliente y servidor), este último envíe un mensaje al cliente y el cliente as vez devuelva el mismo mensaje al servidor. El servidor deberá luego comparar que la cadena enviada y recibida sean iguales ("eco").

Las direcciones IP no van a ser provistas por la persona que ejecutará el programa, sino que la aplicación misma deberá obtenerla del sistema haciendo uso de la instrucción gethostbyname.

Sugerencia:

El servidor puede enviarse a ejecutar de la siguiente forma:

$ servidor puerto

El cliente puede enviarse a ejecutar de la siguiente forma:

$ cliente nombre-host puerto

6.1.1.7.3.1.1 Justificación. El estudiante debe conocer formas alternas para el desarrollo de aplicaciones cliente/servidor que mejoren su ejecución y los acerquen más al manejo real que se les da a este tipo de aplicaciones en un ambiente de red.

6.1.1.7.3.1.2 Objetivos

- Que el conozca la existencia del archivo /etc/hosts en los ambientes UNIX y de manera practica conozca una de sus utilidades.
- Que maneje la primitiva gethostname y la domine en el proceso de desarrollo de aplicaciones cliente servidor.

6.1.1.7.3.1.3 Solución

6.1.1.7.3.1.3.1 Código del servidor

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

#define SOCKET_ERROR        -1
#define BUFFER_SIZE         100
#define MESSAGE             "Esta es la cadena para probar ecooooo"
#define QUEUE_SIZE          5

int main(int argc, char* argv[])
{
   int hSocket,hServerSocket;  /* para manejar socket */
   struct hostent* pHostInfo;   /* guarda info. acerca de la maquina */
   struct sockaddr_in Address; /* Estructura de direccion Internet socket */
   int nAddressSize;
   char pBuffer[BUFFER_SIZE];
   int nHostPort;
```

```
if(argc < 2)
  {
        printf("\nUse: servidor y puerto del host\n");
        return 0;
  }
else
  {
        nHostPort=atoi(argv[1]);
  }

printf("\nIniciando servidor");

printf("\nHaciendo socket");
/* make a socket */
hServerSocket=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);

if(hServerSocket == SOCKET_ERROR)
{
   printf("\nError construyendo socket\n");
   return 0;
}

/* fill address struct */
Address.sin_addr.s_addr=INADDR_ANY;
Address.sin_port=nHostPort;
Address.sin_family=AF_INET;

printf("\nPublicando en el puerto %d",nHostPort);
```

```
/* bind to a port */
if(bind(hServerSocket,(struct sockaddr*)&Address,sizeof(Address))
                == SOCKET_ERROR)
{
   printf("\nError en la conexion con el host\n");
   return 0;
}

printf("\nHaciendo una lista de %d elementos",QUEUE_SIZE);
/* establish listen queue */
if(listen(hServerSocket,QUEUE_SIZE) == SOCKET_ERROR)
{
   printf("\nError haciendo listen\n");
   return 0;
}

for(;;)
{
   printf("\nEsperando por una conexion\n");
   /* Recibiendo la conexion de un socket */
   hSocket=accept(hServerSocket,(struct sockaddr*)&Address,&nAddressSize);

        printf("\nRecibida una conexion");
   strcpy(pBuffer,MESSAGE);
        printf("\nEnviando \"%s\" a cliente",pBuffer);
   /* number returned by read() and write() is the number of bytes
   ** read or written, with -1 being that an error occured
   ** write what we received back to the server */
```

```
        write(hSocket,pBuffer,strlen(pBuffer)+1);
        /* read from socket into buffer */
        read(hSocket,pBuffer,BUFFER_SIZE);

        if(strcmp(pBuffer,MESSAGE) == 0)
           printf("\nLos mensajes coinciden");
        else
           printf("\nAlgo fue cambiado en el mensaje");

            printf("\nCerrando el socket");
        /* Cerrar socket */
        if(close(hSocket) == SOCKET_ERROR)
        {
         printf("\nError cerrando socket\n");
         return 0;
        }
    }
}
```

6.1.1.7.3.1.3.2 Código del cliente

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
```

```
#include <unistd.h>
#include <stdio.h>

#define SOCKET_ERROR        -1
#define BUFFER_SIZE         100
#define HOST_NAME_SIZE      255

int  main(int argc, char* argv[])
{
   int hSocket;                 /* Para manejo del socket */
   struct hostent* pHostInfo;   /* Guarda info acerca de una machine */
   struct sockaddr_in Address;  /* Estruct de direccion de un Internet socket*/
   long nHostAddress;
   char pBuffer[BUFFER_SIZE];
   unsigned nReadAmount;
   char strHostName[HOST_NAME_SIZE];
   int nHostPort;

   if(argc < 3)
     {
          printf("\nUse: cliente nombre-host puerto\n");
          return 0;
     }
   else
     {
          strcpy(strHostName,argv[1]);
          nHostPort=atoi(argv[2]);
     }
```

```
printf("\nHaciendo un socket");
/* Hacer un socket */
hSocket=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);

if(hSocket == SOCKET_ERROR)
{
   printf("\nError haciendo socket\n");
   return 0;
}

/* obtener direccion IP del host-name */
pHostInfo=gethostbyname(strHostName);
/* copiar direccion en un long */
memcpy(&nHostAddress,pHostInfo->h_addr,pHostInfo->h_length);

/* llenar estruct de la direccion */
Address.sin_addr.s_addr=nHostAddress;
Address.sin_port=nHostPort;
Address.sin_family=AF_INET;

printf("\nConectando a  %s en el puerto %d",strHostName,nHostPort);

/* Conectando el host */
if(connect(hSocket,(struct sockaddr*)&Address,sizeof(Address))
  == SOCKET_ERROR)
{
   printf("\nError en conexion al host\n");
   return 0;
```

```
    }
    /* Lee lo que le envia el servidor */
    nReadAmount=read(hSocket,pBuffer,BUFFER_SIZE);
    printf("\nRecibido \"%s\" desde el servidor\n",pBuffer);
    /* Envia la misma cadena de regreso al servidor */
    write(hSocket,pBuffer,nReadAmount);
    printf("\nEscribiendo \"%s\" hacia el servidor",pBuffer);


    printf("\nCerrando socket\n");
    /* Cerrar socket */
    if(close(hSocket) == SOCKET_ERROR)
    {
       printf("\nError cerrando socket\n");
       return 0;
    }
}
```

6.1.2 LLamada a un procedimiento remoto (rpc)

6.1.2.1 Conceptos generales de las rpc

6.1.2.1.1 Breve orientación teórica. Aunque el modelo cliente/servidor es una forma

conveniente de estructurar un sistema operativo distribuido, adolece de una enfermedad incurable; el paradigma básico en torno al cual se construye la comunicación es la entrada/salida.

Los procedimientos *send* y *receive* están reservados para la realización de E/S no es uno de los conceptos fundamentales de los sistemas centralizados, el hecho de que sean la base del cómputo distribuido es visto por las personas que laboran en este campo como un grave error. Su objetivo es lograr que el cómputo distribuido se vea como el cómputo centralizado. La construcción de todo en torno de la E/S no es la forma de lograrlo.

La mayoría de los sistemas de ordenadores están conectados en red, soportando comunicación de datos entre ellos. Como resultado de esto, se han desarrollado muchas técnicas para soportar el desarrollo de aplicaciones que requieren procesos en diferentes sistemas, para comunicar y coordinar sus actividades. Una de estas técnicas son las RPC "Remote Procedure Call", (llamadas a procedimientos remotos)

El concepto de RPC es una sencilla técnica para desarrollar aplicaciones donde se requiere la comunicación entre procesadores que cooperan en un sistema distribuido. RPC es una técnica consistente, como evidencia la existencia de muchas especificaciones e implementaciones.

El mecanismo RPC proporciona un servicio para el programador de aplicaciones que le permite el uso transparente de un servidor para proporcionar alguna actividad por parte de la aplicación. Esto efectivamente puede ser utilizado para interactuar con un servidor computacional o con una Base de Datos, y ha sido usado por algunos sistemas para proporcionar acceso a servicios del sistema operativo. El último uso conocido es en sistemas basados en microordenadores, que da un mejor resultado que en los sistemas tradicionales encontrados en la mayoría de sistemas UNIX, y es especialmente utilizado en sistemas operativos distribuidos.

Las RPC son expresadas como procedimientos ordinarios. Estas llamadas no requieren un compilador especial para el código fuente del programa. Las ventajas de esto incluyen:

- Transparencia: La capa RPC puede ser reemplazada con llamadas a funciones directas si llegan a estar disponibles.
- Familiaridad de la interface: La mayoría de programadores están acostumbrados a una forma de llamadas a procedimientos. Esto permite una fácil adaptación al mecanismo RPC en sistemas ya implantados.

6.1.2.1.2 Preguntas de concepto

d. ¿A qué nos referimos cuando hablamos de RPC de manera general?

La mayoría de los sistemas de ordenadores están conectados en red, soportando comunicación de datos entre ellos. Como resultado de esto, se han desarrollado muchas técnicas para soportar el desarrollo de aplicaciones que requieren procesos en diferentes sistemas, para comunicar y coordinar sus actividades. Una de estas técnicas son las RPC "Remote Procedure Call", (llamadas a procedimientos remotos)

El concepto de RPC es una sencilla técnica para desarrollar aplicaciones donde se requiere la comunicación entre procesadores que cooperan en un sistema distribuido. RPC es una técnica consistente, como evidencia la existencia de muchas especificaciones e implementaciones.

e. ¿Cómo son expresadas las RPC?

Las RPC son expresadas como procedimientos ordinarios.

f. ¿RPC requiere de un compilador especial para el código fuente del programa? ¿Es esto ventajoso o no?

Estas llamadas no requieren un compilador especial para el código fuente del programa. Si es ventajoso.

Las ventajas de esto incluyen:

- Transparencia: La capa RPC puede ser reemplazada con llamadas a funciones directas si llegan a estar disponibles.

- Familiaridad de la interface: La mayoría de programadores están acostumbrados a una forma de llamadas a procedimientos. Esto permite una fácil adaptación al mecanismo RPC en sistemas ya implantados.

6.1.2.1.3 Preguntas de análisis

a. ¿Qué le proporciona RPC al programador?

El mecanismo RPC proporciona un servicio para el programador de aplicaciones que le permite el uso transparente de un servidor para proporcionar alguna actividad por parte de la aplicación.

c. ¿En donde puede ser usado RPC?

RPC puede ser utilizado para interactuar con un servidor computacional o con una Base de Datos, y ha sido usado por algunos sistemas para proporcionar acceso a servicios del sistema operativo. El ultimo uso conocido es en sistemas basados en microordenadores, que da un mejor resultado que en los sistemas tradicionales encontrados en la mayoría de sistemas UNIX, y es especialmente utilizado en sistemas operativos distribuidos.

6.1.2.2 Transferencia de parámetros

6.1.2.2.1 Breve orientación teórica. Las RPC son realmente una manera de ocultar el protocolo de pase de mensajes.

Esto proporciona una manera adecuada de permitir una interpretación a alto nivel, abstrayendo el mecanismo de comunicación a bajo nivel, es decir haciendo transparente al programador el protocolo de red. Las RPC tienen la intención de ser parecidas a una llamada a procedimiento ordinario, pero atravesando la red transparentemente. Un proceso realiza una RPC poniendo sus parámetros y una dirección de retorno en la pila, y salta al comienzo del procedimiento. El procedimiento es responsable de acceder y usar la red. Cuando la ejecución remota finaliza, el procedimiento salta hacia atrás a la dirección de retorno. El proceso de llamada (cliente) entonces continua.

La función del resguardo (figura 1.) del cliente es tomar sus parámetros, empacarlos en un mensaje y enviarlos al resguardo del servidor. Aunque esto parece directo, no es tan sencillo como aparenta. El empacamiento de parámetros en un mensaje se denomina **ordenamiento de parámetros.**

El ejemplo más sencillo es considerar un procedimiento remoto, *sum*(i,j), que toma dos parámetros enteros y regresa su suma aritmética.

La llamada a *sum*, con los parámetros 4 y 7, aparece en la parte izquierda del proceso cliente de la figura 1. El resguardo del cliente toma sus dos parámetros y los coloca en un mensaje de la forma que se indica. También coloca en el mensaje el nombre o número del procedimiento por llamar, puesto que el servidor podría soportar varias llamadas y se le tiene que indicar cual de ellas se necesita.

Cuando el mensaje llega al servidor, el resguardo examina éste para ver cuál procedimiento necesita y entonces lleva a cabo la llamada apropiada.
Cuando el servidor termina su labor, su resguardo recupera el control. Toma el resultado proporcionado por el servidor y lo empaca en un mensaje. Este mensaje se envía de regreso al resguardo de cliente, que lo desempaca y regresa el valor al procedimiento cliente.

Si las máquinas cliente y servidor son idénticas y todos los parámetros y resultado son de tipo escalar, como enteros, caracteres o booleanos, este método funciona bien. Sin embargo, en un sistema distribuido de gran tamaño, es común tener distintos tipos de máquinas. Cada una tiene a menudo su propia representación de los números, caracteres y otros elementos. Por ejemplo, las mainframes de IBM utilizan el código de caracteres EBCDIC, mientras que las computadoras personales de IBM utilizan ASCII. en consecuencia, no es posible pasar un parámetro carácter de una PC de IBM cliente a una mainframe IBM servidor, mediante el sencillo esquema de la figura 1 ; el servidor interpretará lo interpretará de manera incorrecta.

Aparecen otros problemas similares con la representación de enteros (complemento a 1 o complemento a 2) y de manera particular, en los números de punto flotante. Además existe un problema mucho más irritante, puesto que en ciertas máquinas, como la Intel 486, numeran sus bytes de derecha a izquierda, mientras que otras, como la Sun SPARC, los numeran en el orden contrario. El formato de Intel se llama **little endian** (partidarios del extremo menor) y el de SPARC **big endian** (partidarios del extremo mayor).

6.1.2.2.2 Preguntas de concepto

c. ¿ Cómo se debe representar la información de los mensajes?

Una forma es diseñar un estándar de red o **forma canónica** para los enteros, caracteres, booleanos, números de punto flotante, etc.., y pedir a todos los emisores que conviertan sus representaciones internas a esta forma durante el ordenamiento. Por ejemplo, supongamos que se decide utilizar complemento a 2 para los enteros, ASCII para los caracteres, 0 (falso) y 1 (cierto) para los booleanos, y el formato IEEE para los números de punto flotante ; todo guardado como little endian. Para cualquier lista de enteros, caracteres, booleanos y números de punto flotante, el patrón preciso necesario queda determinado por completo hasta el último bit. Como resultado el resguardo del servidor no tiene que preocuparse más por el orden que tiene el cliente, puesto que el orden de los bits en el mensaje ya está determinado, en forma independiente del hardware del cliente.

El problema con éste método es que a veces es ineficiente. Supongamos que un cliente big endian se comunica con un servidor big endian. De acuerdo con las reglas, el cliente debe convertir todo a little endian en el mensaje y el servidor debe volver a convertir todo cuando el mensaje llegue. Aunque esto no contiene ambigüedades, requiere de dos conversiones que en realidad no son necesarias. Esta observación da lugar a un segundo método : el cliente utiliza su propio formato original e indica en el primer byte del mensaje su formato. Así, un cliente little endian construye un mensaje little endian y un cliente big endian construye un mensaje con su formato

correspondiente. Tan pronto llega el mensaje, el resguardo del servidor examina el primer byte para ver quien es el cliente. Si es igual al servidor, no necesita conversión, en caso contrario, el resguardo del servidor realiza la correspondiente conversión.

Aunque sólo hemos analizado la conversión entre endians, se puede realizar de la misma manera la conversión entre complemento a 1 y complemento a 2, EBCDIC a ASCII, etc. el truco consiste en conocer la organización del mensaje y la identidad del cliente.

d. ¿ Cuál es el origen de los procedimientos de resguardo o stubs?

En muchos de los sistemas basados en RPC, éstos se generan de forma automática. Dada una especificación del procedimiento servidor y las reglas de codificación, el formato del mensaje queda determinado de forma única. Así, es posible tener un compilador que lea las especificaciones del servidor y genere un resguardo del cliente que empaque sus parámetros en el formato oficial de los mensajes. En forma similar el compilador también puede generar un resguardo del servidor que los desempaque y que llame al servidor. El hecho de disponer de ambos procedimientos de resguardo a partir de una especificación formal del servidor no sólo facilita la vida de los programadores, sino que reduce la probabilidad de error y hace que el sistema sea transparente con respecto alas diferencias en la representación interna de los elementos de los datos.

6.1.2.2.3 Preguntas de análisis

c. ¿Qué hace diferente a las RPC?

Las llamadas a procedimiento generadas por el compilador no son diferentes a cualquier otra llamada que se defina directamente en un programa. La verdadera diferencia comienza cuando se hace efectiva la llamada al procedimiento.

Los lenguajes utilizan los procedimientos como una manera de estructurar los programas. Un programa tendrá sentencias condicionales, bucles y llamadas a procedimientos. Cuando se realiza una llamada a procedimiento, normalmente se usa la pila, poniendo los parámetros en ella y reservando espacio para las variable locales.

Las RPC son realmente una manera de ocultar el protocolo de pase de mensajes. Esto proporciona una manera adecuada de permitir una interpretación a alto nivel, abstrayendo el mecanismo de comunicación a bajo nivel, es decir haciendo transparente al programador el protocolo de red. Las RPC tienen la intención de ser parecidas a una llamada a procedimiento ordinario, pero atravesando la red transparentemente.

d. Explique de manera general el funcionamiento de RPC.

RPC funciona de la siguiente manera :

Un proceso realiza una RPC poniendo sus parámetros y una dirección de retorno en la pila, y salta al comienzo del procedimiento. El procedimiento es responsable de acceder y usar la red. Cuando la ejecución remota finaliza, el procedimiento salta hacia atrás a la dirección de retorno. El proceso de llamada (cliente) entonces continua.

El mecanismo RPC proporciona un stub o resguardo que empaqueta los argumentos en una forma sencilla, para ser transmitidos bajo una red a otro sistema, donde los argumentos son desempaquetados por otro stub o resguardo en el proceso servidor y pasados a la función actual o procedimiento que ha sido llamado. El valor de retorno de el procedimiento es pasado al proceso cliente de un manera similar. Mientras todo esto ocurre, el cliente ha sido bloqueado y no se reanuda hasta que recibe el valor de retorno.

En resumen, una llamada a un procedimiento remoto se realiza mediante los siguientes pasos :

1. El procedimiento cliente llama al stub o resguardo del cliente de manera usual.
2. El stub del cliente construye un mensaje y hace un señalamiento al núcleo.
3. El núcleo envía el mensaje al núcleo remoto.
4. El núcleo remoto proporciona el mensaje al stub o resguardo del servidor.
5. El stub del servidor desempaca los parámetros y llama al servidor.
6. El servidor realiza el trabajo y regresa el resultado al resguardo.
7. El stub servidor empaca el resultado en un mensaje y hace un señalamiento al núcleo.
8. El núcleo remoto envía el mensaje al núcleo del cliente.
9. El núcleo del cliente da el mensaje al stub del cliente.
10. El resguardo desempaca el resultado y regresa al cliente.

6.1.2.3 Protocolos rpc

6.1.2.3.1 Breve orientación teórica. En teoría, cualquier protocolo antiguo puede funcionar si obtiene los bits del núcleo del cliente y los lleva al núcleo del servidor; pero en la práctica hay que tomar decisiones importantes en este punto, decisiones que pueden tener un fuerte impacto en el desempeño.

La primera decisión está entre un protocolo orientado a la conexión o un protocolo sin conexión. En el primer caso, al momento en que el cliente se conecta con el servidor, se establece una conexión entre ellos. Todo el tráfico, en ambas direcciones, utiliza esta conexión.

La ventaja de contar con una conexión es que la comunicación es más fácil. Cuando un núcleo envía un mensaje, no tiene que preocuparse por su pérdida, ni tampoco por los reconocimientos. Todo ello se maneja a nivel inferior, mediante que el software que soporta la conexión. Cuando se opera una red amplia, esta ventaja es con frecuencia irresistible.

La desventaja en una LAN, es la pérdida de desempeño. Todo ese software adicional estorba en el camino. Además, la ventaja principal (no perder los paquetes) difícilmente se necesita en una LAN, puesto que las LAN son confiables en este sentido. En consecuencia, la mayoría de los sistemas distribuidos que pretenden utilizarle en un edificio o campus utilizan los protocolos sin conexión.

La segunda opción principal está en utilizar un protocolo de propósito general o alguno diseñado de forma específica para RPC. Puesto que no existen estándares en esta área, el uso de un protocolo RPC adaptado quiere decir por lo general que cada quien diseñe el suyo.

Algunos sistemas distribuidos utilizan IP (o UDP, integrado a IP) como el protocolo básico.

6.1.2.3.2 Preguntas de análisis

c. ¿En RPC se debe utilizar protocolos orientados a la conexión o sin conexión?

En el primer caso, al momento en que el cliente se conecta con el servidor, se establece una conexión entre ellos. Todo el tráfico, en ambas direcciones, utiliza esta conexión.

La ventaja de contar con una conexión es que la comunicación es más fácil. Cuando un núcleo envía un mensaje, no tiene que preocuparse por su pérdida, ni tampoco por los reconocimientos. Todo ello se maneja a nivel inferior, mediante que el software que soporta la conexión. Cuando se opera una red amplia, esta ventaja es con frecuencia irresistible.

La desventaja en una LAN, es la pérdida de desempeño. Todo ese software adicional estorba en el camino. Además, la ventaja principal (no perder los paquetes) difícilmente se necesita en una LAN, puesto que las LAN son confiables en este sentido. En consecuencia, la mayoría de los sistemas distribuidos que pretenden utilizarle en un edificio o campus utilizan los protocolos sin conexión.

d. ¿Qué ventajas y desventajas traería el uso de IP (o UDP, integrado a IP) como el protocolo básico?

Esta opción tiene varios puntos a su favor:

1. El protocolo ya ha sido diseñado, lo que ahorra un trabajo considerable.

2. Se dispone de muchas implantaciones, lo cual, de nuevo, ahorra trabajo.
3. Estos paquetes se pueden enviar y recibir en cualquier sistema UNIX.
4. Los paquetes IP y UDP son soportados por muchas redes existentes.

En general IP y UDP son fáciles de utilizar y se ajustan bien a los sistemas UNIX existentes y a redes como Internet. Esto hace que sea directo escribir clientes y servidores que se ejecuten en sistemas UNIX, lo cual ayuda a que el código se pueda ejecutar y se pruebe pronto.

Como es usual, el lado malo es el desempeño. IP no se diseño como un protocolo para usuario final. Se diseño como base para poder establecer conexiones confiables TCP entre las redes recalcitrantes. Por ejemplo, puede trabajar con compuertas que fragmentan paquetes en pedazos pequeños, de forma que puedan pasar a través de las redes con un tamaño máximo de paquete.

6.1.2.4 Las grandes lineas del protocolo

6.1.2.4.1 Breve orientación teórica. El protocolo debe permitir:

- La identificación del procedimiento
- La autentificación de la petición

La identificación del procedimiento

El principio es el de agrupar los diferentes procedimientos en un programa. Los diferentes procedimientos que constituyen un mismo programa contribuyen a la realización de un servicio específico. Por ejemplo, el protocolo NFS constituye un programa de protocolo RPC y contiene diferentes procedimientos cuyo punto común es que todos permitan manipular archivos a distancia.

Un programa se identificará por un número entero y cada procedimiento de un programa será igualmente identificado por un entero. A título de ejemplo el programa NFS lleva el número 100003 y los procedimientos de lectura y escritura llevan los números 6 y 8. Finalmente para permitir la evolución de los programas, cada uno posee un número de versión.

La autenficación

La definición del protocolo prevee la posibilidad de que un cliente se identifique a un servidor, lo que permite asegurar la seguridad de los accesos a los objetos del sistema distante. Los mensaje intercambiados en el curso de llamadas a procedimientos remotos llevan información relativa a esta identificación. Como el protocolo es independiente del sistema subyacente, están previstos diferentes estilos de autentificación (por ejemplo, la ausencia de autentificación o una autentificación UNIX), dejando la posibilidad de definir otros nuevos.

6.1.2.4.2 Preguntas de concepto

d. Ordene de lo particular a lo general

Procedimientos, servicios, programas

Servicios, programas, procedimientos.

Procedimientos, programas, servicios.

Programas, servicios, procedimientos.

El orden correcto es:

Procedimientos, programas, servicios.

e. ¿A través de qué, se establece la identificación de los procedimientos y programas de un servicio RPC?

A través de un número entero.

f. Todo procedimiento tiene un procedimiento de número cero que no permite hace ningún cálculo. ¿Entonces, cual es su funcionalidad?

Permite comprobar la disponibilidad de un servicio de número dado.

6.1.2.4.3 Preguntas de análisis

c. RPC permite el llamado a procedimientos que se encuentran en máquinas remotas. ¿A través de que mecanismo se puede implementar la seguridad en dichos accesos?

La definición del protocolo provee la posibilidad de que un cliente se identifique a un servidor, lo que permite asegurar los accesos a los objetos del sistema distante.

d. ¿Qué hace la orden *rpcinfo*?

Permite obtener información sobre los diferentes servicios (o un servicio en particular) disponibles sobre una máquina de nombre dado.

6.1.2.5 Los diferentes niveles de utilización

6.1.2.5.1 Breve orientación teórica. Hay tres niveles de utilización del mecanismo de RPC tal como ha sido implantado en UNIX.

Cada uno de estos niveles ofrece transparencia al usuario, es decir, demanda un conocimiento más o menos fino (hasta nulo) del protocolo y de los mecanismos de más bajo nivel, tales como, por ejemplo, los socket.

El nivel alto

Este es el que oculta el máximo de detalles al usuario. Este solo debe realizar una llamada de función de una biblioteca, especificando el nombre de la máquina objetivo y los diferentes parámetros de la llamada. Evidentemente, a este nivel no es posible desarrollar nuevos servicios RPC (si no es por composición de los servicios existentes).

El nivel intermedio

Es incontestablemente el más interesante para el desarrollador. Supone un conocimiento mínimo de los protocolos XDR y RPC y es suficiente para desarrollar la mayoría de las aplicaciones. Descarga totalmente al usuario de la manipulación de los sockets.

La interfaz disponible a este nivel se apoya sobre el protocolo UDP. Esto significa que el tamaño de los mensajes intercambiados (y. por tanto, el tamaño cié los parámetros v de los resultados de las funciones llamadas) es limitado. Cuando esta limitación es molesta, es necesario utilizar el interfaz propuesto por el nivel bajo. La sucesión de operaciones a realizar para definir un servicio de este tipo consiste en:

- Escribir las diferentes funciones sobre el servidor;
- Después de haber escogido los números de programa y de versión, solicitar la anotación de las diferentes funciones por el demonio «portmap» por medio de la función **regiserrpc.**

La llamada a una función desde una posición remota se realiza por medio de la función **callrpc.**

El nivel bajo

Su utilización es mucho más compleja y supone un buen conocimiento de los mecanismos concernientes a los sockets. Se muestra necesaria para las aplicaciones donde las opciones elegidas en el nivel intermedio

6.1.2.5.2 Preguntas de concepto

c. Mencione una instrucción del nivel alto de programación en RPC

rwall, rnusers, etc.

d. A través de que instrucción se coloca a disposición un servicio RPC en una máquina

svc_run();

6.1.2.5.3 Preguntas de análisis

c. ¿Es posible que un desarrollador implemente un nuevo servicio RPC usando las llamadas a las funciones de la biblioteca del primer nivel de programación?

No es posible, a menos que se desarrolle uno con base en en servicios ya existentes.

d. ¿Basta con hacer un nuevo servicio RPC y registrarlo en el sistema?

No, no basta. Hay que activar el servicio, lo cual se logra con la sentencia svc_run().

6.1.2.6 Grupo general de practicas rpc

6.1.2.6.1 Práctica I

6.1.2.6.1.1 *Enunciado.* Realizar una aplicación RPC que diga el número de usuarios conectados en una maquina remota, usando la función rnusers o en su defecto su equivalente a través de una llamada con callrpc.

6.1.2.6.1.2 Justificación. El estudiante de una forma amena y real puede introducirse en el uso de funciones RPC, de primer nivel, al tiempo que observa un ejemplo de la aplicación de este tipo de llamadas con miras a la formación experimental del concepto de un sistema distribuido.

6.1.2.6.1.3 Objetivos

- Que el estudiante comprenda la manera de utilizar las funciones **rnusers** y **callrpc.**
- Lograr que el estudiante descubra las diferencias y semejanzas que se presentan al hacer uso de estas funciones.
- Fortalecer los conceptos teóricos que el estudiante posee sobre el funcionamiento de dichas funciones.

6.1.2.6.1.4 Solución

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>

int main(int argc, char *argv[]) {

unsigned int nusers;

  nusers=0;
  fflush(stdout);
  if
(callrpc(argv[1],RUSERSPROG,RUSERSVERS,RUSERSPROC_NUM,xdr_void,NULL,xdr
_int,&nusers)!=0) {
     printf("Error haciendo llamada al procedimiento\n");
     exit(1); }

  fprintf(stdout, "Hay %i usuarios en %s\n", nusers, argv[1]);
    exit(0);
}
```

1.1.1.2 Práctica II

1.1.1.2.1 Enunciado. Realizar una aplicación RPC, del primer nivel de programación, que permita saber el puerto asociado a un servicio RPC activo en una maquina remota.

1.1.1.2.2 Justificación. Es una de las formas más elementales de iniciarse en el uso de las funciones RPC. Sugiere el uso de las funciones getrpcbynumber y getrpcport. Además induce al estudiante a la manipulación de herramientas complementarias del servicio RPC como es el caso de *rpcinfo* para mirar primero la información referente a los servicios activos en la maquina par luego introducir un valor significativo durante la ejecución de su aplicación.

1.1.1.2.3 Objetivos

- Inducir al estudiante a manipular las funciones a utilizar en este caso (getrpcbynumber y getrpcport) y a comprender su forma de operar.
- Que el estudiante se entrene en la utilización de las funciones RPC, del primer nivel de programación mencionadas anteriormente.

1.1.1.2.4 Solución

```
/*programa que anuncia el numero de puerto de un servicio RPC para 1 */
/*maquina el numero de servicio se da por teclado */
```

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

char *nom[]={"galix"};
main() {
  int i,servicio;
  struct rpcent *p;
  printf("numero de servicio?? ");
  scanf("%d",&servicio);
  printf("%s => ",nom[0]);
  if ((p=getrpcbynumber(servicio))==NULL)
    printf("%d: numero de servicio desconocido\n",servicio);
  else{
    printf(" %d: servicio %s ",servicio, p->r_name);
    printf("puerto %d\n",getrpcport(nom[0],servicio,1,IPPROTO_UDP));
  }
}
```

6.1.2.6.2 Práctica III

6.1.2.6.2.1 Enunciado. Realizar una aplicación RPC, que posea*:*

- Un servidor que registre una función RPC (registerrpc) y ponga a disposición dicha función (svc_run)
- Un cliente que haga el llamado a la función (callrpc) y le transfiera el correspondiente parámetro.

Para hacer más práctico y uniforme el ejemplo se sugiere que la función halle el factorial de un número.

6.1.2.6.2.2 *Justificación.* Esta práctica se convierte en una buena introducción para lo que será la programación de nivel intermedio en RPC, a la vez que el estudiante con base en la abstracción que le ofrece el utilizar las sentencias sugeridas en el enunciado se le facilitará la asimilación de conceptos básicos de RPC que le ayudará en una concepción más específica de lo que es un sistema distribuido.

6.1.2.6.2.3 Objetivos

- Introducir al estudiante en la programación de nivel intermedio de RPC.
- Entrenar al estudiante en lo relacionado con la utilización de las funciones sugeridas en el enunciado, así como la transferencia de parámetros en RPC.

6.1.2.6.2.4 Solución

Servidor

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <math.h>
#define PROGRAM 0x20000100
#define VERSION 1
#define ROUTINE 1

extern  double  sqrt();
double  *compute_result();

main()
{
      if(registerrpc(PROGRAM,VERSION,ROUTINE,compute_result,
                  xdr_int, xdr_double) == -1)
      {
            perror("registerrpc");
            exit(1);
      }
      system("clear");
      printf("Listo el servidor. Esperando llamada remota...\n");
      svc_run();
      fprintf(stderr,"svc_run() call failed");
      exit(1);
}
```

```
double  *
compute_result(input)
int     *input;
{
int     count;
static  int  output;

        output=1.0;
        for(count= *input; count>1; count--)
                output *= count;
        /*output = sqrt(output);*/
        return(&output);
}
```

Cliente

```
/*
  * Programa cliente que calcula factorial de un entero
  * valendose para ello funciones de RPC
  */

#include <stdio.h>
#include <rpc/rpc.h>
#define PROGRAM 0x20000100
#define VERSION 1
```

```
#define ROUTINE 1


main(argc, argv)
int     argc;
char    **argv;
{
int     input,
        ret_val;
int  result;
char    input_buf[25];


        system("clear");
        printf("Entre un Entero=>");
        fflush(stdout);
        fgets(input_buf,25,stdin);
        input = atoi(input_buf);
       if((ret_val=callrpc(argv[1],PROGRAM,VERSION,ROUTINE,
                        xdr_int,&input,xdr_double,&result))
                != 0)
        {
                clnt_perrno(ret_val);
                exit(1);
        }
        printf("Resultado = %d\n",result);
}
```

1.1.1.3 Práctica IV

1.1.1.3.1 Enunciado. Realizar una aplicación RPC para calcular una la solución de una ecuación de segundo grado. El cliente deberá tomar los coeficientes de la ecuación y transmitirlos como parámetros al servidor, donde se llevaran acabo los cálculos.

1.1.1.3.2 Justificación. A través de esta aplicación el estudiante podrá:

- Manipular las estructuras empleadas para la transmisión de parámetros.
- Registrar completamente su propio programa dentro de los servicios RPC que ofrece la máquina.
- Adiestrarse en la programación de nivel intermedio de RPC que es para muchos el nivel más interesante para un desarrollador.

1.1.1.3.3 Objetivos

- Que el estudiante conozca y se entrene en la forma de manipular las estructuras empleadas para la transmisión de parámetros.
- Afianzar al estudiante en la programación de nivel intermedio de RPC.
- Lograr que el estudiante visualice la utilización de RPC en problemas cotidianos

1.1.1.3.4 Solución

Archivo de cabecera

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

```
#include <rpc/xdr.h>
#define ESG_PROG 0x33333333          /*numero de programa*/
#define ESG_VERS 1                   /*numero de version*/
#define ESG_PROC 1                   /*numero del procedimiento esgrado */

struct datos {
  float  e1,e2,e3;
};
int xdr_datos();
```

xdr_datos

```
#include "cabecera.h"

xdr_datos(XDR *xdrp, struct datos *p) {
  return(xdr_float(xdrp,&p->e1) && xdr_float(xdrp,&p->e2) && xdr_float(xdrp,&p->e3));
}
```

Registro

```
/* Registro del procedimiento del programa que halla la solucion de una
ecuacion de segundo grado*/

#include <stdio.h>
#include "cabecera.h"
```

```
char *esg();


main() {
  int rep;
  rep=registerrpc(ESG_PROG,ESG_VERS,ESG_PROC,esg,xdr_datos,xdr_float);
  if (rep==-1) {
    printf("Error registrando esg\n");
    exit(2);
  }
  svc_run();
  fprintf(stderr,"Error al hacer svc_run\n");
  exit(3);
}
```

Servidor

```
#include "cabecera.h"
#include "math.h"


char *esg(struct datos *p) {
  float d;
  static struct datos resp;
  d=sqrt((p->e2*p->e2)-(4*p->e1*p->e3));
  resp.e1=(-p->e2+d)/(2*p->e1);
```

```
  resp.e2=(-p->e2-d)/(2*p->e1);
  return((char *) &resp);
}
```

Cliente

```
/*Procedimiento que llama al servicio sobre una maquina distante cuyo nombre
se da como parametro*/

#include <stdio.h>
#include "cabecera.h"
//#define BUFF_SIZE 20

main(int n, char *v[]) {
  struct datos entrada, resp, x;
  int op,m;
  char soda1[20],soda2[20],soda3[20];

  system("clear");
  printf("Programa que calcula la solucion de una Ec. de segundo grado\n");
  printf("AXý+BX+C=0\n\n");
  printf("Entre valor de A: ");
  gets(soda1);
  entrada.e1=atoi(soda1);
  printf("Entre valor de B: ");
```

```
  gets(soda2);
  entrada.e2=atoi(soda2);
  printf("Entre valor de C: ");
  gets(soda3);
  entrada.e3=atoi(soda3);
  m=callrpc(v[1],ESG_PROG,ESG_VERS,ESG_PROC,xdr_datos,&entrada,xdr_float,&x);
  if (m==0)
    printf("\n\nResultado\nX1= %f\nX2=%f\n",x.e1,x.e2,x);
  else
    printf("Error %m en llamado a la funcion",m);
}
```

Servidor

```
#include "cabecera.h"
#include "math.h"

char *esg(struct datos *p) {
  float d;
  static struct datos resp;

  d=sqrt((p->e2*p->e2)-(4*p->e1*p->e3));
  resp.e1=(-p->e2+d)/(2*p->e1);
  resp.e2=(-p->e2-d)/(2*p->e1);
  return((char *) &resp);
}
```

6.1.2.6.3 Práctica V

6.1.2.6.3.1 Enunciado. Analizar, compilar y ejecutar la aplicación "hora", disponible en el servidor Linux, que como su nombre lo indica solicita la hora a una máquina remota. Anote sus experiencias y resultados.

6.1.2.6.3.2 Justificación. Esta es una forma diferente de realización de aplicaciones RPC, la cual corresponde al nivel bajo. Aunque es más compleja su realización, se hace necesaria cuando las opciones del nivel intermedio no son apropiadas.

Con esta práctica el estudiante tendrá la necesidad de utilizar herramientas complementarias al servicio RPC como es el caso de *rpcgen* (compilador de RPC), el cual toma como base un archivo de tipo *.x y genera el código del cliente y servidor de la aplicación.

La aplicación cuenta con un par de archivos de texto que señalan el camino a seguir por el estudiante para llegar hasta la ejecución final de la misma, camino que enriquecerá sus conocimientos en este ultimo (aunque difícil) nivel de programación.

6.1.2.6.3.3 Objetivos

- Ilustrar al estudiante en lo referente a la programación de nivel bajo de RPC e inducirlo al análisis y comprensión de la misma.
- Enseñar al estudiante el uso de herramientas complementarias al servicio RPC, como lo es **rpcgen**(compilador de RPC).

6.1.3 Comunicación en grupo

6.1.3.1 Conceptos generales

6.1.3.1.1 Orientacion teorica. Un grupo es una colección de procesos que actúan juntos en cierto sistema o alguna forma determinada por el usuario.

La propiedad fundamental de todos los grupos es que cuando un mensaje se envía al propio grupo, todos los miembros de éste lo reciben. Es una forma de comunicación **uno-muchos** (un emisor, muchos receptores) y contrasta con la **comunicación puntual** de la figura 2.

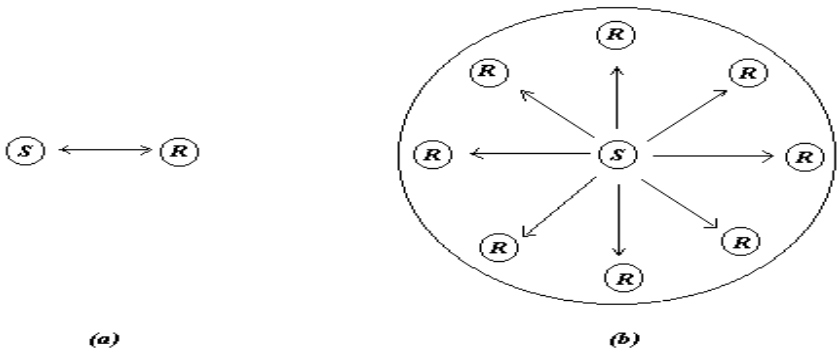


Figura 2. La comunicación puntual y la comunicación uno-muchos .

Los grupos son dinámicos. Se pueden crear nuevos grupos y destruir grupos anteriores. Un proceso se puede unir a un grupo o dejar otro. Un proceso puede ser miembro de varios grupos a la vez. En consecuencia, se necesitan mecanismos para el manejo de grupos y la membresia de los mismos.

Los grupos son algo parecido a las organizaciones sociales. Una persona puede ser miembro de un club de lectores, un club de tenis y una organización ambientalista. En un día particular, él podría recibir correo (mensajes) que le avisen de un nuevo libro para hornear pasteles de cumpleaños, el torneo anual del día de las madres y el inicio de una campaña para salvar a las marmotas del sur. En cualquier momento, él es libre de dejar todos o alguno de estos grupos y unirse a otros.

La finalidad de presentar los grupos es permitir a los procesos que trabajen con colecciones de procesos como una abstracción. Así, un proceso puede enviar un mensaje a un grupo de servidores sin tener que conocer su número o su localización, que puede cambiar de una llamada a la siguiente.

En ciertas redes, es posible crear una dirección especial de red (por ejemplo, indicada al hacer que uno de los bits de orden superior tome el valor 1) a la que pueden escuchar varias máquinas. Cuando se envía un mensaje a una de estas direcciones, se entrega de manera automática a todas las máquinas que escuchan a esa dirección. Esta técnica se llama **multitransmisión.**

Las redes que no tienen multitransmisión a menudo siguen teniendo **transmisión simple,** lo que significa que los paquetes que contienen cierta dirección (por ejemplo, 0) se entregan a todas las máquinas.

Por último, si no se dispone de la multitransmisión o la transmisión simple, se puede implantar la comunicación en grupo mediante la transmisión por parte del emisor de paquetes individuales a cada uno de los miembros del grupo. El envío de un mensaje de un emisor a un receptor se llama a veces **unitransmisión,** para distinguirla de los otros tipos de transmisión.

6.1.3.1.2 Preguntas de concepto

a. ¿Qué entiende usted por GRUPO y cual es su propiedad fundamental?

Un grupo es una colección de procesos que actúan juntos en cierto sistema o alguna forma determinada por el usuario. La propiedad fundamental de todos los grupos es que cuando un mensaje se envía al propio grupo, todos los miembros de éste lo reciben.

b. ¿Cuál es la diferencia primordial entre la comunicación puntual y la comunicación Uno-Muchos? Explique con un gráfico.

La diferencia principal entre estos dos tipos de comunicación radica en que la comunicación puntual se establece de un emisor a un receptor mientras La comunicación uno-muchos se establece entre un emisor y varios receptores.

c. ¿Qué entiende usted por Multitransmision?

En ciertas redes, es posible crear una dirección especial de red (por ejemplo, indicada al hacer que uno de los bits de orden superior tome el valor 1) a la que pueden escuchar varias máquinas. Cuando se envía un mensaje a una de estas direcciones, se entrega de manera automática a todas las máquinas que escuchan a esa dirección. Lo

anterior es lo que se conoce como **multitransmisión.** La implantación de los grupos mediante multitransmisión es directa: sólo hay que asignar a cada grupo una dirección de multitransmisión distinta.

d. Además de la multitransmisión y la transmisión simple ¿ qué otro tipo de transmisión se puede implantar?

Si no se dispone de la multitransmisión o la transmisión simple, se puede implantar la comunicación en grupo mediante la transmisión por parte del emisor de paquetes individuales a cada uno de los miembros del grupo. Para un grupo con *n* miembros, se necesitan *n* paquetes, en vez de un paquete en el caso de la multitransmisión o la transmisión simple. Aunque es menos eficiente, esta implantación también funciona, en particular con grupos pequeños. Este tipo de transmisión se conoce con el nombre de **Unitransmision**.

6.1.3.1.3 Preguntas de Análisis

a. Realice una analogía entre Grupo y algún acontecimiento de la vida cotidiana.

Los grupos son algo parecido a las organizaciones sociales. Una persona puede ser miembro de un club de lectores, un club de tenis y una organización ambientalista. En un día particular, él podría recibir correo (mensajes) que le avisen de un nuevo libro

para hornear pasteles de cumpleaños, el torneo anual del día de las madres y el inicio de una campaña para salvar animales en via de extinción.

b. ¿La transmisión simple puede ser utilizada para trabajar en comunicación en grupo? ¿Qué desventaja presenta?

La transmisión simple también se puede utilizar para implantar los grupos, pero es menos eficiente. Cada máquina recibe su transmisión simple, por lo que su software debe verificar si el paquete va dirigido a ella. Si no, el paquete se descarta, pero se pierde cierto tiempo durante el procesamiento de la interrupción. Sin embargo, un paquete llega a todos los miembros del grupo.

6.1.3.2 Aspectos del diseño

6.1.3.2.1 Orientacion teorica. La comunicación en grupo tiene posibilidades de diseño similares a la transferencia regular de mensajes, como el almacenamiento en buffers *vs.* el no almacenamiento, bloqueo *vs.* no bloqueo, etc.

Sin embargo, también existen un gran número de opciones adicionales por realizar, ya que el envío a un grupo es distinto de manera inherente del envío a un proceso. Además, los grupos se pueden organizar internamente de varias formas. También se pueden direccionar de formas novedosas que no son importantes en la comunicación puntual

Grupos cerrados *vs.* grupos abiertos

Los sistemas que soportan la comunicación en grupo se pueden dividir en dos categorías, según quién pueda enviar a quién. Algunos sistemas soportan los grupos cerrados, donde sólo los miembros del grupo pueden enviar hacia el grupo. Los extraños no pueden enviar mensajes al grupo como un todo, aunque pueden enviar mensajes a miembros del grupo en lo Individual. En contraste, otros sistemas soportan los **grupos** abiertos, que no tienen esta propiedad. Si se utilizan los grupos abiertos, cualquier proceso del sistema puede enviar a cualquier grupo. La diferencia entre los grupos cerrados y abiertos se muestra en la figura 3.

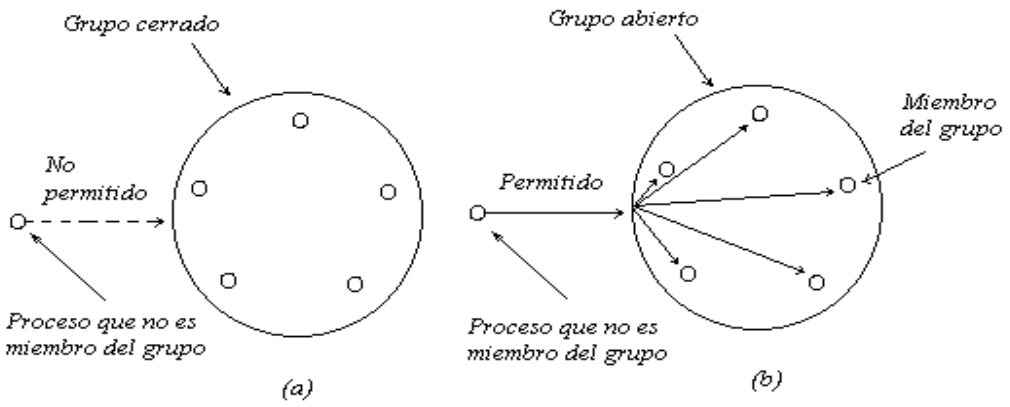


Figura 3. Comparación entre grupos abiertos y cerrados.

Grupos de compañeros *vs.* grupos jerárquicos

La distinción entre los grupos cerrados y abiertos se relaciona con la pregunta de quién se puede comunicar con el grupo. Otra distinción importante tiene que ver con la estructura interna del grupo. En algunos grupos, todos los procesos son iguales. Nadie es el jefe y todas las decisiones se toman en forma colectiva. En otros grupos, existe cierto tipo de jerarquía. Estos patrones de comunicación se muestran en la figura 4.

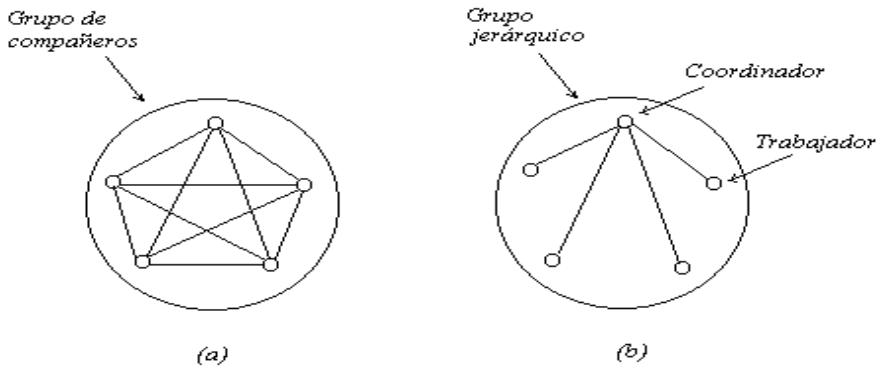


Figura 4 .La comunicación en un grupo de compañeros y en un simple grupo jerárquico.

Cada una de estas organizaciones tiene sus propias ventajas y desventajas. El grupo de compañeros es simétrico y no tiene punto de falla. Si uno de los procesos falla, el grupo sólo se vuelve más pequeño, pero puede continuar.

El grupo jerárquico tiene las propiedades opuestas. La pérdida del coordinador lleva a todo el grupo a un agobiante alto, pero mientras se mantenga en ejecución, puede tomar decisiones sin molestar a los demás.

Membresía del grupo

Si se utiliza la comunicación en grupo, se requiere cierto método para la creación y eliminación de grupos, así como para permitir a los procesos que se unan o dejen grupos. Un posible método es tener un **servidor de grupos** al cual se pueden enviar todas las solicitudes. El servidor de grupos puede mantener entonces una base de datos de todos los grupos y sus membresías exactas. Este método es directo, eficiente y fácil de implantar. Por desgracia, comparte una desventaja fundamental con todas

las técnicas centralizadas: un punto de falla. Si el servidor de grupos falla, deja de existir el manejo de los mismos. Es probable que la mayoría o todos los grupos deban reconstruirse a partir de cero, terminando con todo el trabajo realizado hasta entonces.

El método opuesto es manejar la membresía de grupo en forma distribuida. En un grupo abierto, un extraño puede enviar un mensaje a todos los miembros del grupo para anunciar su presencia. En un grupo cerrado se necesita algo similar (de hecho, incluso los grupos cerrados deben estar abiertos a la opción de admitir otro miembro). Para salir de un grupo, basta que el miembro envíe un mensaje de despedida a todos.

6.1.3.2.2 Preguntas de Concepto

a. ¿Cuáles son las categorias en las cuales se pueden dividr los sistemas que soportan la comunicación en grupo?

Los sistemas que soportan la comunicación en grupo se pueden dividir en dos categorías, según quién pueda enviar a quién. Algunos sistemas soportan los grupos cerrados, donde sólo los miembros del grupo pueden enviar hacia el grupo. Los extraños no pueden enviar mensajes al grupo como un todo, aunque pueden enviar mensajes a miembros del grupo en lo Individual. En contraste, otros sistemas soportan los **grupos** abiertos, que no tienen esta propiedad. Si se utilizan los grupos abiertos, cualquier proceso del sistema puede enviar a cualquier grupo. La diferencia entre los grupos cerrados y abiertos se muestra en la figura 3.

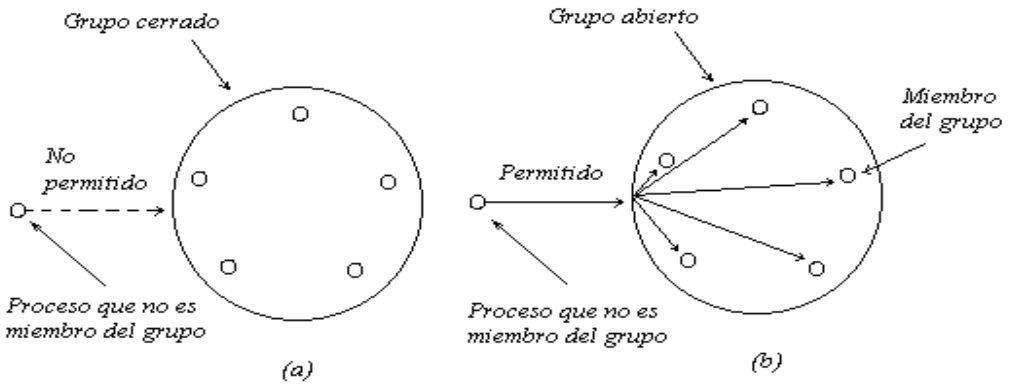


Figura 3. Comparación entre grupos abiertos y cerrados.

b. ¿Cuándo se debe utilizar los grupos abiertos y los grupos cerrados? Explique.

Los grupos cerrados se utilizan en general para el procesamiento paralelo. Por ejemplo, una colección de procesos que trabajan de manera conjunta para jugar una partida de ajedrez podrían formar un grupo cerrado. Tienen su propio objetivo y no interactúan con el mundo exterior.

Por otro lado, cuando la idea de grupos pretende soportar servidores duplicados, entonces es importante que los procesos que no sean miembros (clientes) puedan enviar mensajes hacia el grupo. Además, podría ser necesario que los miembros del grupo utilizaran la comunicación en grupo; por ejemplo, para decidir quién debe llevar a cabo una solicitud particular. La distinción entre los grupos abiertos y cerrados se hace por lo general por razones de implantación.

c. Explique el concepto de grupo jerarquico y grupo de compañeros.

La distinción entre los grupos cerrados y abiertos se relaciona con la pregunta de quién se puede comunicar con el grupo. Otra distinción importante tiene que ver con la estructura interna del grupo. En algunos grupos, todos los procesos son iguales. Nadie es el jefe y todas las decisiones se toman en forma colectiva. En otros grupos, existe cierto tipo de jerarquía. Por ejemplo, un proceso es el coordinador y todos ¡os demás son trabajadores. En este modelo, si se genera una solicitud de un trabajo, ya sea de un cliente externo o uno de los trabajadores, ésta se envía al coordinador. Éste decide entonces cuál de los trabajadores es el más adecuado para llevarla a cabo y se envía. Por supuesto, también son posibles jerarquías más complejas. Estos patrones de comunicación se muestran en la figura 4.

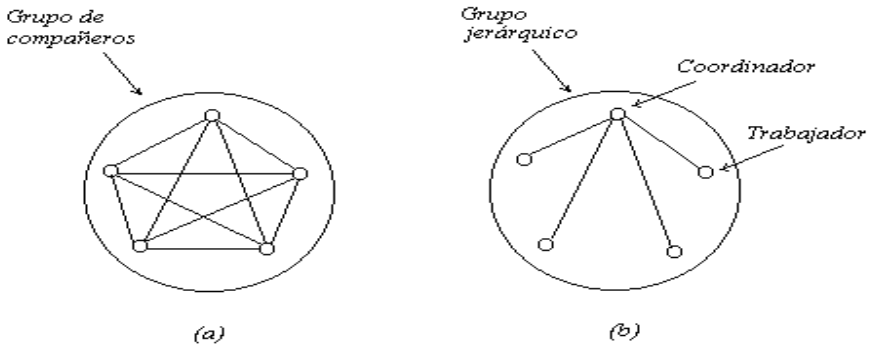


Figura 4. La comunicación en un grupo de compañeros en un simple grupo jerárquico.

d. ¿Qué métodos se pueden utilizar para la creación y eliminación de grupos, así como para permitir a los procesos que se unan o dejen grupos?

Un posible método es tener un **servidor de grupos** al cual se pueden enviar todas las solicitudes. El servidor de grupos puede mantener entonces una base de datos de todos los grupos y sus membresías exactas. Este método es directo, eficiente y fácil de implantar. Por desgracia, comparte una desventaja fundamental con todas las técnicas centralizadas: un punto de falla. Si el servidor de grupos falla, deja de existir el manejo de los mismos. Es probable que la mayoría o todos los grupos deban reconstruirse a partir de cero, terminando con todo el trabajo realizado hasta entonces.

El método opuesto es manejar la membresía de grupo en forma distribuida. En un grupo abierto, un extraño puede enviar un mensaje a todos los miembros del grupo para anunciar su presencia. En un grupo cerrado se necesita algo similar (de hecho,

incluso los grupos cerrados deben estar abiertos a la opción de admitir otro miembro). Para salir de un grupo, basta que el miembro envíe un mensaje de despedida a todos.

6.1.3.2.3 Preguntas de Analisis

a. ¿Cuáles son las ventajas y desventajas que en su concepto presentan los grupos jerarquicos y los grupos decompañeros? Explique con ejemplos.

El grupo de compañeros es simétrico y no tiene punto de falla. Si uno de los procesos falla, el grupo sólo se vuelve más pequeño, pero puede continuar. Una desventaja es que la toma de decisiones es más difícil. Para tomar una decisión, hay que pedir un voto, lo que produce cierto retraso y costo.

El grupo jerárquico tiene las propiedades opuestas. La pérdida del coordinador lleva a todo el grupo a un agobiante alto, pero mientras se mantenga en ejecución, puede tomar decisiones sin molestar a los demás. Por ejemplo, un grupo jerárquico podría ser adecuado para un programa de ajedrez en paralelo. El coordinador torna el tablero actual, genera todos los movimientos válidos a partir de él y los envía a los trabajadores para *su* evaluación. Durante ésta, se generan nuevos tableros y envían de regreso al coordinador. Cuando un trabajador está inactivo, solicita al coordinador un nuevo tablero en el cual trabajar. De esta forma, el coordinador controla la estrategia de búsqueda y desarrolla el árbol del juego (por ejemplo, mediante el método de búsqueda alfa-befa), pero deja a los trabajadores la evaluación real.

b. Explique los problemas más comunes asociados con la membresía de los grupos.

En primer lugar, si un miembro falla, en realidad sale del grupo. El problema es que no existe un anuncio apropiado de este hecho como en el caso en que un proceso salga del grupo en forma voluntaria. Los demás miembros deben descubrir esto en forma experimental, al observar que el miembro ya no responde. Una vez verificado que el miembro en realidad está inactivo, puede eliminarse del grupo.

Otro aspecto problemático es que la salida y la entrada al grupo debe sincronizarse con el envío de mensajes. En otras palabras, a partir del momento en que un proceso se ha unido a un grupo, debe recibir todos los mensajes que se envíen al mismo. De manera similar, tan pronto un proceso salga del grupo, no debe recibir más mensajes de éste y los otros miembros no deben recibir más mensajes de dicho proceso. Una forma de garantizar que una entrada o salida se integra al flujo de mensajes en el lugar correcto es convertir esta operación en un mensaje a todo el grupo.

Un aspecto final relacionado con la membresía es la acción a realizar si fallan tantas máquinas que el grupo ya no pueda funcionar. Se necesita cierto protocolo para reconstruir el grupo. De manera invariable, alguno de los procesos deberá tomar la iniciativa, pero ¿qué ocurre si dos o tres lo intentan al mismo tiempo? El protocolo debe poder resolver esto.

6.1.3.3 Direccionamiento

6.1.3.3.1 Orientacion teorica. Para enviar un mensaje a un grupo, un proceso debe tener una forma de especificar dicho grupo.

En otras palabras, los grupos deben poder direccionarse, al igual que los procesos. Una forma es darle a cada grupo una dirección, parecida a una dirección de proceso. Si la red soporta la multitransmisión, la dirección del grupo se puede asociar con una dirección de multitransmisión, de forma que cada mensaje enviado a la dirección del grupo se pueda multitransmitir

Si el hardware no soporta la multitransmisión pero sí la transmisión simple, el mensaje se puede transmitir. Cada núcleo lo recibirá y extraerá de él la dirección del grupo. Si ninguno de los procesos en la máquina es un miembro del grupo, entonces se descarta la transmisión. En caso contrario, se transfiere a todos los miembros del grupo.

Por último, si no se soporta la multitransmisión o la transmisión simple, el núcleo de la máquina emisora debe contar con una lista de las máquinas que tienen procesos pertenecientes al grupo, para entonces enviar a cada una un mensaje puntual. Estos tres métodos de implantación se muestran en la figura 5.

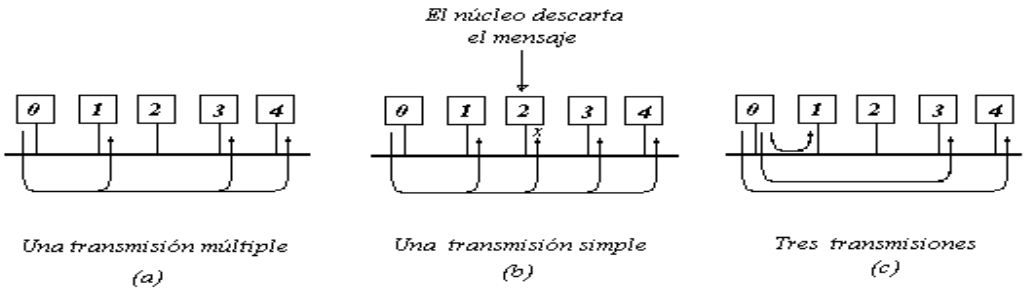


Figura 5. El proceso 0 enviado a un grupo que consta de los procesos 1, 3 y 4.

Un segundo método de direccionamiento de grupo consiste en pedir al emisor una lista explícita de todos los destinos (por ejemplo, direcciones IP). Si se utiliza este método, el parámetro de la llamada *send* que especifica el destino es un apuntador a una lista de direcciones.

La comunicación en grupo también permite un tercer método, un tanto novedoso, de direccionamiento, que llamaremos **dircccionamiento de predicados.** Con este sistema, se envía cada mensaje a todos los miembros del grupo (o tal vez a todo el sistema) mediante uno de los métodos ya descritos, pero con nuevo giro. Cada mensaje contiene un predicado (expresión booleana) para ser evaluado. El predicado puede utilizar el número de máquina del receptor, sus variables locales u otros factores. Si el valor del predicado es verdadero, se acepta el mensaje. SÍ es falso, el mensaje se descarta.

Primitivas send y receive

En forma ideal, la comunicación puntual y la comunicación en grupo deberían combinarse en un conjunto de primitivas. Sin embargo, si RPC es el mecanismo usual de comunicación del usuario, en vez de los simples *send y receive,* entonces es difícil combinar RPC y la comunicación en grupo. El envío de un mensaje a un grupo no se puede modelar como llamada a un procedimiento. La principal dificultad es que, con RPC, el cliente envía un mensaje al servidor y obtiene de regreso una respuesta. Con la comunicación en grupo, existen en potencia *n* respuestas diferentes. ¿Cómo podría trabajar una llamada de procedimiento con *n* respuestas? En consecuencia, un método común es abandonar el modelo solicitud/respuesta (en los dos sentidos) subyacente en RPC y regresar a las llamadas explícitas para el envío y recepción (modelo de un sentido).

Atomicidad

Una característica de la comunicación en grupo a la que hemos aludido varias veces es la propiedad del todo o nada. La mayoría de los sistemas de comunicación en grupo están diseñados de forma que, cuando se envíe un mensaje a un grupo, éste llegue de manera correcta a todos los miembros del grupo o a ninguno de ellos. No se permiten situaciones en las que ciertos miembros reciben un mensaje y otros no. La propiedad del todo o nada en la entrega se llama **atomicidad o transmisión atómica.**

La atomicidad es deseable, puesto que facilita la programación de los sistemas distribuidos. Si un proceso envía un mensaje al grupo, no tiene que preocuparse por qué hacer si alguno de ellos no lo obtiene.

Ordenamiento de mensajes

Para que la comunicación en grupo sea fácil de comprender y utilizar, se necesitan dos propiedades. La primera es la transmisión atómica, ya analizada. Ésta garantiza que un mensaje enviado al grupo llegue a todos los miembros o a ninguno. La segunda propiedad se refiere al ordenamiento de mensajes.

La mejor garantía es la entrega inmediata de todos los mensajes, en el orden en que fueron enviados. Si el proceso O envía el mensaje *A* y un poco después el proceso 4 envía el mensaje *B,* el sistema debe entregar en primer lugar *A* a todos los miembros del grupo y después entregar *B* a todos los miembros del grupo. De esta forma, todos los receptores obtiene todos los mensajes en el mismo orden. Este patrón de entrega es algo comprensible para los programadores y en el cual basan su software. Lo llamaremos **ordenamiento con respecto al tiempo global,** puesto que entrega todos los mensajes en el orden preciso con el que fueron enviados .

Grupos traslapados

Como hemos mencionado, un proceso puede ser miembro de varios grupos a la vez. Este hecho puede provocar un nuevo tipo de inconsistencia. Para ver el problema,

observemos la figura 2-35, la cual muestra dos grupos, 1 y 2. Los procesos *A, B y C* son miembros del grupo 1 y los procesos *B, C* y *D* son miembros del grupo 2.

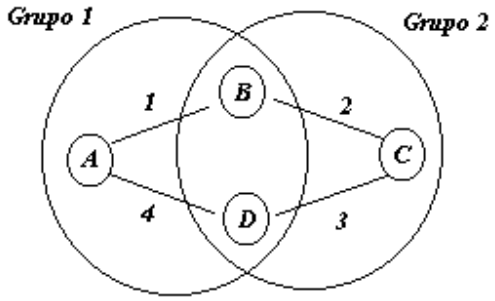


Fig. 2-35. Cuatro procesos A, B, C y D y cuatro mensajes. Los mensajes B y C obtienen los mensajes de A y D en orden diferente.

El problema aquí es que, aunque existe un ordenamiento con respecto al tiempo global dentro de cada grupo, no es necesario que exista coordinación entre varios grupos. Algunos sistemas soportan un ordenamiento bien definido entre los grupos traslapados y otros no.

Escalabilidad

Nuestro aspecto final del diseño es la escalabilidad. Muchos algoritmos funcionan bien mientras todos los grupos tengan unos cuantos miembros, pero ¿qué ocurre cuando existen decenas, centenas o incluso miles de miembros por grupo? ¿O miles de grupos? Además, ¿qué ocurre si el sistema es tan grande que no cabe en una LAN,

de modo que se necesiten varias LAN y compuertas? ¿Qué ocurre si los grupos están diseminados en varios continentes?

La presencia de compuertas puede afectar muchas propiedades de la implantación. Para comenzar, la multitransmisión es más complicada. Consideremos, por ejemplo, la red que se muestra en la figura 2-36. Consta de cuatro LAN y cuatro compuertas, con el fin de protegerse contra la falla de cualquier compuerta.

Imaginemos que una de las máquinas de la LAN 2 ejecuta una multitransmisión. Cuando el paquete de multitransmisión llega a las compuertas *Gl* y *G3,* ¿qué deben hacer éstas? Si lo descartan, la mayoría de las máquinas nunca lo verán, lo cual destruye su valor como multitransmisión. Sin embargo, si el algoritmo sólo tiene compuertas hacia todas las mul-titransmisiones, entonces el paquete se copiará a la LAN 1 y la LAN 4 y poco después a la LAN 3 dos veces. Peor aún, la compuerta *G2* verá la multitransmisión de *G4,* la copiará a la LAN 2 y viceversa Es claro que se necesita un algoritmo más complejo, que mantenga un registro de los paquetes anteriores, con el fin de evitar un crecimiento exponencial en el número de multitransmisiones de paquetes.

6.1.3.3.2 Preguntas de Concepto

a. ¿Cuáles son los métodos más utlizados en el direccionamiento de los grupos. Explique su funcionamiento.

Una forma es darle a cada grupo una dirección, parecida a una dirección de proceso. Si la red soporta la multitransmisión, la dirección del grupo se puede asociar con una dirección de multitransmisión, de forma que cada mensaje enviado a la dirección del grupo se pueda multitransmitir. De esta forma, el mensaje será enviado a todas las máquinas que lo necesiten y a ninguna más.

Si el hardware no soporta la multitransmisión pero sí la transmisión simple, el mensaje se puede transmitir. Cada núcleo lo recibirá y extraerá de él la dirección del grupo. Si ninguno de los procesos en la máquina es un miembro del grupo, entonces se descarta la transmisión. En caso contrario, se transfiere a todos los miembros del grupo.

Por último, si no se soporta la multitransmisión o la transmisión simple, el núcleo de la máquina emisora debe contar con una lista de las máquinas que tienen procesos pertenecientes al grupo, para entonces enviar a cada una un mensaje puntual. Estos Un segundo método de direccionamiento de grupo consiste en pedir al emisor una lista explícita de todos los destinos (por ejemplo, direcciones IP). Si se utiliza este método, el parámetro de la llamada *send* que especifica el destino es un apuntador a una lista de direcciones. Este método tiene la seria desventaja de que obliga a los

procesos del usuario (es decir, a los miembros del grupo) a ser conscientes de quién es miembro de cada grupo. En otras palabras, no es transparente. Además, cuando cambia la membresía del grupo, los procesos del usuario deben actualizar sus listas de miembros. Estos tres metodos de implantacion se muestran en la figura 5

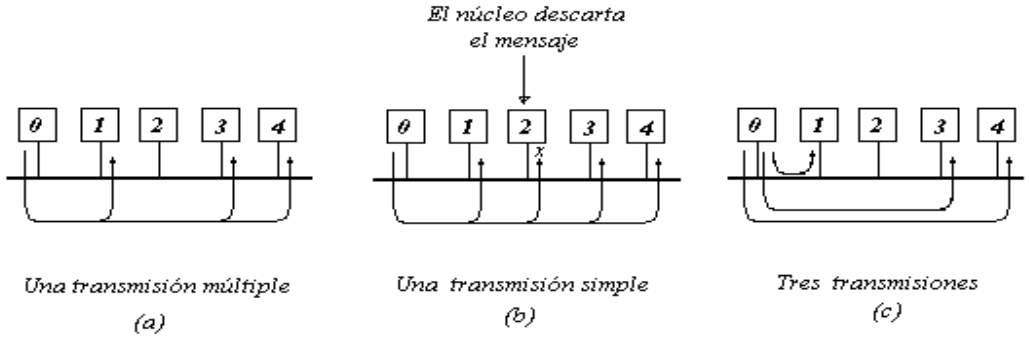


Figura 5. El proceso 0 enviado a un grupo que consta de los procesos 1, 3 y 4**.** (a) Implantación de transmisión múltiple, (b) Implantación de transmisión simple, (c) Implantación de unitransmisión.

La comunicación en grupo también permite un tercer método, un tanto novedoso, de direccionamiento, que llamaremos **dircccionamiento de predicados.** Con este sistema, se envía cada mensaje a todos los miembros del grupo (o tal vez a todo el sistema) mediante uno de los métodos ya descritos, pero con nuevo giro. Cada mensaje contiene un predicado (expresión booleana) para ser evaluado. El predicado puede utilizar el número de máquina del receptor, sus variables locales u otros

factores. Si el valor del predicado es verdadero, se acepta el mensaje. SÍ es falso, el mensaje se descarta. Mediante este esquema, es posible, por ejemplo, enviar un mensaje sólo a aquellas máquinas que tengan al menos 4M de memoria libre y estén dispuestas a ocuparse de un nuevo proceso.

b. ¿Cuál es la principal dificultad que se presenta al trabajar comunicación en grupo con RPC?

Es difícil combinar RPC y la comunicación en grupo. El envío de un mensaje a un grupo no se puede modelar como llamada a un procedimiento. La principal dificultad es que, con RPC, el cliente envía un mensaje al servidor y obtiene de regreso una respuesta. Con la comunicación en grupo, existen en potencia *n* respuestas diferentes.

c. Explique el funcionamiento de SEND y RECEIVE en la comunicación en grupo.

Supongamos, por el momento, que queremos combinar las dos formas de comunicación. Para enviar un mensaje, uno de los parámetros de *send* indica el destino. Si es una dirección de un proceso, se envía un mensaje a ese proceso en particular. SÍ es una dirección de grupo (o un apuntador a una lista de destinos), se envía un mensaje a todos los miembros del grupo. Un segundo parámetro de *send* apunta hacia el mensaje por enviar.

La llamada se puede o no almacenar en un buffer, puede o no utilizar bloqueo, ser confiable o no confiable, para cualquiera de los casos puntual o de grupo. Por lo general, los diseñadores del sistema eligen entre estas opciones y las dejan fijas.

En forma análoga, *receive* indica una disposición para aceptar un mensaje y es posible que se bloquee hasta disponer de uno. Si se combinan las dos formas de comunicación, entonces *receive* termina su labor cuando llega un mensaje puntual o un mensaje de grupo.

En el diseño recién descrito, la comunicación tiene un sentido. Las respuestas son mensajes independientes y no están asociadas con solicitudes previas. A veces es deseable esta asociación, para intentar tener un poco de RPC.

d. ¿Qué se entiende por atomicidad?

Una característica de la comunicación en grupo a la que hemos aludido varias veces es la propiedad del todo o nada. La mayoría de los sistemas de comunicación en grupo están diseñados de forma que, cuando se envíe un mensaje a un grupo, éste llegue de manera correcta a todos los miembros del grupo o a ninguno de ellos. No se permiten situaciones en las que ciertos miembros reciben un mensaje y otros no. La propiedad del todo o nada en la entrega es lo que se conoce como **atomicidad o transmisión atómica.**

e. Explique el método de ordenamiento con respecto al tiempo global.

Este método consiste en entregar todos los mensajes en el orden preciso con el que fueron enviados (aquí ignoramos por conveniencia el hecho de que, de acuerdo con la teoría de la relatividad de Einstein, no existe una cosa tal como el tiempo global absoluto).

La mejor garantía es la entrega inmediata de todos los mensajes, en el orden en que fueron enviados. Si el proceso O envía el mensaje *A* y un poco después el proceso 4 envía el mensaje *B,* el sistema debe entregar en primer lugar *A* a todos los miembros del grupo y después entregar *B* a todos los miembros del grupo. De esta forma, todos los receptores obtienen todos los mensajes en el mismo orden.

6.1.3.3.3 Preguntas de Análisis

a. ¿Cómo podría trabajar una llamada de procedimiento remoto con *n* respuestas?

Una posible solucion es abandonar el modelo solicitud/respuesta (en los dos sentidos) subyacente en RPC y regresar a las llamadas explícitas para el envío y recepción (modelo de un sentido).

Los procedimientos de biblioteca llamados por los procesos para realizar la comunicación en grupo pueden ser iguales o distintos a los que se utilizan para la comunicación puntual. Si el sistema se basa en **RPC,** los procesos usuario nunca llaman en forma directa a *send* y *receive,* por lo que existen menos incentivos para la fusión de las primitivas puntuales y de grupo. Si los programas usuario llaman en forma directa a *sendy receive,* hay que hacer algo para realizar la comunicación en grupo con estas primitivas ya existentes, en lugar de inventar un conjunto nuevo.

b. Teniendo en cuenta la Atomicidad ¿ los metodos descritos anteriormente (Transmision simple, multiple y unitransmision) fallan? ¿Cómo podría solucionarse?

Los metodos sobre los cuales se hace mencion, fallan, puesto que es posible la sobre-ejecución del receptor en una o varias máquinas. La única forma de garantizar que cada destino recibe todos sus mensajes es pedirle que envíe de regreso un reconocimiento después de recibir el mensaje.

c. Describa un algoritmo sencillo en el cual exista la posibilidad de transmisión atómica.

El emisor comienza con el envío de un mensaje a todos los miembros del grupo. Los cronómetros se activan y se envían las retransmisiones en los casos necesarios. Cuando un proceso recibe un mensaje, si no ha visto ya este mensaje particular, lo envía también a todos los miembros del grupo (de nuevo con cronómetros y retransmisiones en los casos necesarios). Si ya ha visto el mensaje, este paso no es necesario y el mensaje se descarta. No importa el número de máquinas que fallen o el número de paquetes perdidos: en cierto momento, todos los procesos sobrevivientes obtendrán el mensaje.

6.1.3.4 Practicas de comunicación en grupos

6.1.3.4.1 Enunciado. Analice, contraste con la teoría de comunicación en grupos y elabore conclusiones al respecto del código dispuesto más adelante.

Descripción de su funcionamiento:

El programa principal se va a encargar de leer dos matrices y comprobar si se pueden multiplicar. Acto seguido van a arrancar tantos procesos como le hayamos indicado en la línea de órdenes para multiplicar las matrices. Cada proceso se va a encargar de generar una fila de la matriz producto mientras queden filas por generar. Naturalmente, las matrices que intervienen en la operación deben estar en memoria compartida. Para controlar la fila que debe generar cada proceso, vamos a utilizar un semáforo que se inicializa con en el total de filas de la matriz producto y se va decrementando por cada fila generada. El proceso principal se queda esperando a que todos los demás terminen para presentar el resultado.

6.1.3.4.2 Justificación. El anterior sirve también como un ejemplo de paralelismo, sin embargo en sistemas monoprocesadores como los que disponemos este método no resulta eficiente, pero es un buen ejercicio de sincronismo y comunicación en grupo.

6.1.3.4.3 Objetivos

- Fortalecer los conocimientos teóricos que el estudiante posee acerca del funcionamiento y en general todo lo concerniente a la comunicación en grupo.
- Inducir al estudiante a analizar, comparar y elaborar conclusiones con relación a la teoría de comunicación en grupo y lo que se aprecia en el código presentado.
- Lograr en el estudiante un entendimiento considerable de la forma de operar de la comunicación en grupo y las características de esta.

6.1.3.4.4 Código

```
/***
PROGRAMA: matrices.c
DESCRIPCION: Progama para multiplicar matrices en paralelo.
FORMA DE USO:
  matrices n_de_procesos ***/

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>

/* Definicion de matriz. */
typedef struct {
```

```
 int shmid; /* Identificador de la zona de memoria compartida donde
               va a estar la matriz. */
 int filas, columnas;
 float **coef; /* Coeficientes de la matriz. */
}matriz;

/* FUNCION: crear_matriz.
DESCRIPCION: Función que crea la memoria compartida donde va a estar la matriz. */

matriz *crear_matriz (filas, columnas)
int filas, columnas;
{
  int shmid, i;
  matriz *m;

  /* Peticion de memoria compartida. */
  shmid = shmget (IPC_PRIVATE,sizeof (matriz) + filas*sizeof(float *) +
filas*columnas*sizeof(float), IPC_CREAT | 0600);
  if (shmid == -1) {
    perror ("crear_matriz (shmget)");
    exit(-1);
  }

  /* Nos atamos a la memoria. */
  if ((m = (matriz *)shmat (shmid, 0, 0)) == (matriz *)-1) {
    perror ("crear_matriz (shmdt)");
    exit(-1);
  }
```

```
  /* Inicialización de la matriz. */
  m->shmid = shmid;
  m->filas = filas;
  m->columnas = columnas;
  /* Le damos formato a la memoria para poder direccionar los coeficientes de la matriz. */
  m->coef = (float **)&m->coef + sizeof(float **);
  for (i = 0; i < filas; i++)
    m->coef[i] = (float *)&m->coef[filas] + i*columnas*sizeof (float);
  return m;
}

/* FUNCION: leer_matriz
DESCRIPCION: Lee una matriz del fichero estandar de entrada. */

matriz *leer_matriz () {
  int filas, columnas, i, j;
  matriz *m;

  printf ("Filas: ");
  scanf("%d",&filas);
  printf ("Columnas: ");
  scanf ("%d", &columnas);
  m=crear_matriz(filas,columnas);
  for(i =0; i < filas; i++)
    for (j = 0; j < columnas; j++) {
      printf("Digite valor de la fila %d columna %d: ",i,j);
      scanf("%f",&m->coef[i][j]);
```

```
    }
  return m;
}
/* FUNCION: multiplicar_matrices
DESCRIPCION: Multiplica dos matrices y crea una nueva para el resultado.
El trabajo se distribuye entre el total de procesos que indique numproc. */

matriz *multiplicar_matrices (a, b, numproc)
  matriz *a, *b;
  int numproc;
{
  int p, semid, estado;
  matriz *c;

  if (a->columnas != b->filas)
  return NULL;
  c = crear_matriz (a->filas, b->columnas);

  /* Creacion de dos semáforos. Uno de ellos se inicializa con el
     total de filas de la matriz producto. */
  semid = semget (IPC_PRIVATE, 2, IPC_CREAT | 0600);
  if (semid == -1) {
    perror ("multiplicar_matrices (semget)");
    exit(-1);
  }
  semctl (semid, 0, SETVAL, 1);
  semctl (semid, 1, SETVAL, c->filas+1);
```

```
/* Creacion de tantos procesos como indique numproc. */
for (p = 0; p < numproc; p++) {
  if (fork () == 0) {
    /* Codigo para los procesos hijo. */
    int i, j, k;
    struct sembuf operacion;

    operacion.sem_flg = SEM_UNDO;
    while (1) {
      /* Cada proceso hijo se encarga de generar una fila de la matriz producto.
         Para saber que columna tiene que generar, consulta el valor del semáforo. */

      /* Operacion P sobre el semaforo 0. */
      operacion. sem_num = 0;
      operacion.sem_op = -1;
      semop (semid, &operacion, 1);
      /* Consultamos el valor del semaforo. */
      i = semctl (semid, 1, GETVAL, 0);
      if (i > 0) {
        /* Decrementamos el valor del semaforo 1 en una unidad.   */
        semctl (semid, 1, SETVAL, --i);
        /* Operacion V sobre el semaforo 0. */
        operacion.sem_num = 0;
        operacion. sem_op = 1;
        semop (semid, &operacion, 1);
      }
      else
        exit (0);
```

```
      /* Calculo de la fila i-esima de la matriz producto, */
      for (j = 0; j < c->columnas; j++) {
        c->coef[i][j] = 0;
        for (k = 0; k < a->columnas; k++)
          c->coef[i][j] += a->coef[i][k]*b->coef[k][j];
      }
    }/*while*/
  } /* if */
 } /* for */
 /* Esperamos a que terminen todos los procesos */
 for (p = 0; p < numproc; p++)
   wait (&estado);

 /* Borramos el semaforo. */
 semctl (semid, 0, IPC_RMID, 0);
 return c;
}

/*
FUNCION: destruir_matriz
DESCRIPCION: Funcion encargada de liberar una zona de memoria compartida. */

destruir_matriz (m)
matriz *m;
{

 shmctl (m->shmid, IPC_RMID, 0);
}
```

```
/* FUNCION: imprimir_matriz
DESCRIPCION: Esta funcion presenta una matriz en el fichero estandar de
salida  */

imprimir_matriz (m)
matriz *m;
{
  int i,j;

  for (i = 0; i < m->filas; i++) {
    for (j = 0; j < m->columnas; j++)
      printf("|%g|", m->coef[i][j]);
    printf ("\n");
  }
}

/*** Funcion principal. ***/

main (argc, argv)
int argc; char *argv [];
{
  int numproc;
  matriz *a, *b, *c;

  /* Analisis de los parametros de la linea de ordenes.*/
  if (argc != 2)
    numproc = 2;
```

```
  else
    numproc = atoi (argv [1]) + 1;
  /* Lectura de las matrices. */
  system("clear");
  printf("                Programa de multiplicacion de matrices.\n");
  printf("              Como ejemplo de la comunicacion en grupos\n\n\n");
  printf("Entre los datos de la primera matriz\n");
  a = leer_matriz();
  printf("\nEntre los datos de la segunda matriz\n");
  b = leer_matriz();

  /* Procesamiento de las matrices. */
  c = multiplicar_matrices (a, b, numproc);
  if (c != NULL) {
    printf("\nMatriz resultado\n");
    imprimir_matriz (c);
  }
  else
    fprintf (stderr, "Las matrices no se pueden multiplicar.");
  printf("\nLo importante de este ejemplo no es la codificacion del proceso\n");
  printf("de multiplicacion de las matrices, sino la manera como trabajan los\n");
  printf("procesos que se generan en la consecucion del calculo matricial\n\n");
  destruir_matriz (a);
  destruir_matriz (b);
  destruir_matriz (c);
}
```

6.2 PROGRAMACION PARALELA

6.2.1 Generalidades de la programacion en paralelo

6.2.1.1 ***Breve orientación teórica.*** Una instrucción puede especificar, además de varias operaciones aritméticas, la dirección de un dato que puede ser leído o escrito en memoria y/o la dirección de la próxima instrucción a ser ejecutada.

Un computador es susceptible de ser programado en términos de éste modelo básico usando el lenguaje de maquina, aunque para la mayoría de los casos resulta complejo, pues se tiene que guardar el "track" de millones de posiciones de memoria y organizar la ejecución de miles de instrucciones de máquinas. Técnicas de diseño modular son aplicadas en la construcción de complejos programas partiendo de simples componentes y tales componentes están estructurados en términos de altos niveles de abstracción tales como estructuras de datos, ciclos interactivos y procedimientos. Abstracciones tales como los procedimientos hacen más fácil el seguimiento de la modularidad observando los objetos que van a ser utilizados, sin tener en cuenta su estructura interna. Entonces hacer lenguajes de alto nivel tales como Fortran, Pascal, C y Ada los cuales guían el diseño expresado en termino de esas abstracciones para ser traducidas automáticamente al código ejecutable.

La programación paralela introduce fuentes adicionales de la complejidad: si programáramos en el nivel más bajo, no sólo aumentaría el número de las instrucciones ejecutadas, sino que también necesitaríamos manejar explícitamente la

ejecución de millares de procesadores y coordinar millones de interacciones del interprocessor. Por lo tanto, la abstracción y la modularidad son por lo menos tan importantes como en la programación secuencial. En hecho, acentuaremos la modularidad como cuarto requisito fundamental para el software paralelo, además de la ejecución simultánea, la escalabilidad y la localización.

Algunos conceptos importantes

Un cómputo: consiste en un conjunto de tareas

Un canal: es una cola de mensaje desde la cual un remitente puede poner mensajes y de cuál un receptor puede quitar un mensaje, y los bloquea si los mensajes no están disponibles. Una colección de mensajes “conectan” a las tareas entre si.

Una tarea: encapsula un programa y una memoria local y define un conjunto de los puertos que definen la interfaz a su ambiente.

El cómputo paralelo consiste en unas o más tareas. Las tareas se ejecutan en paralelo. El número de tareas puede variar durante la ejecución de programa.

6.2.1.2 Preguntas de concepto

d. ¿En qué consiste el computo en paralelo?

El computo en paralelo consiste en la ejecución simultanea de dos o más tareas.

e. Consulte el modelo de programación en paralelo, tarea/canal, y anote sus principales características.

Funcionamiento: La tarea y el canal tienen una semejanza directa con el multicomputer. Una tarea representa un pedazo del código que se pueda ejecutar secuencialmente, en un solo procesador. Si dos tareas que comparten un canal se asocian a diversos procesadores, la conexión del canal se implementa como una comunicación entre procesadores; si se asocian al mismo procesador, más mecanismos eficientes pueden ser utilizados.

Independencia asociada: Las tareas obran recíprocamente con el mismo mecanismo (canales) sin importar la localización de la tarea, el resultado computado por un programa no depende de donde las tareas se ejecutan.

Modularidad: En el diseño modular de un programa, varios componentes de un programa se desarrollan por separado, como módulos independientes, y después se combinan para obtener un programa completo.

Determinismo: Un algoritmo o un programa es determinista si la ejecución con una información de entrada determinada rinde siempre la misma salida. Es no determinístico si las ejecuciones múltiples con la misma información de entrada dan diversas salidas.

f. Consulte otros modelos de programación paralela y diga sus principales características.

El paso de mensajes: El paso de mensajes es probablemente, hoy, el modelo más extensamente usado de la programación paralela. Los programas de paso de mensajes, como los programas de tarea/canal, crean tareas múltiples, y cada tarea encapsula datos locales. Cada tarea es identificada por un nombre único, y las tareas obran recíprocamente enviando y recibiendo mensajes a y desde tareas nombradas.

Paralelismo de los datos: Paralelismo de los datos, llamado así por la explotación de la concurrencia que deriva de la aplicación de la misma operación a los múltiples elementos de una estructura de datos, por ejemplo, "sumar 2 a todos los elementos de un vector" o "incrementar el sueldo de todos los empleados con 5 años de servicio". Un programa dato-paralelo consiste en una secuencia de tales operaciones.

Memoria compartida: En el modelo de programación de memoria compartida, las tareas comparten un espacio de común de direcciones, en el cual ellas leen y escriben

asincronamente. Varios mecanismos tales como bloqueos y semáforos se pueden utilizar para controlar el acceso a la memoria compartida.

6.2.2 El sistema pvm

6.2.2.1 Breve orientación teórica

Introducción

PVM es un conjunto integrado de herramientas y librerías de software, que emulan un marco de trabajo de computación concurrente de propósito general, flexible y heterogéneo. Todo esto sobre un conjunto de computadoras de distintas arquitecturas interconectadas entre sí. El principal objetivo del sistema PVM, es permitir que tal conjunto de máquinas, sean usadas en forma cooperativa para hacer computación concurrente o paralela.

Principios

Resumidamente, los principios sobre los cuales se basa PVM incluyen:

- El conjunto de hosts que conforman la máquina virtual puede ser configurado por el usuario (pool de hosts).
- Acceso translucido al hardware.
- Computación basada en procesos.
- Modelo de pasaje de mensajes explícito.
- Soporte heterogéneo (en cuanto a: arquitecturas de hardware, redes y aplicaciones).
- Soporte para Multiprocesadores.

Partes del sistema

El sistema PVM esta compuesto por dos partes. La primera es un demonio, llamado pvmd3 que reside en todas las máquinas que conforman la máquina virtual.

La segunda parte del sistema es una librería de interfaces de rutinas de PVM, que contiene un repertorio de primitivas necesarias para la cooperación entre las tareas de una aplicación. Tales rutinas pueden ser llamadas por el usuario y permiten realizar pasaje de mensajes, expansión de procesos, coordinación de tareas y modificación de la máquina virtual. Las interfaces están provistas para los lenguajes Fortran y C, siendo implementadas como subrutinas en el primer caso y como funciones en el último.

Todas las tareas en PVM son identificadas por un entero llamado TID (del inglés task identifier), estos enteros son suministrados por el demonio local de PVM.

PVM incluye el concepto de grupos de tareas, para ello provee un conjunto de funciones, que permiten a una tarea - entre otras funciones - unirse o dejar un grupo determinado de tareas

6.2.2.2 Grupo general de prácticas

6.2.2.2.1 Práctica I

6.2.2.2.1.1 Enunciado. Desarrollar una aplicación con PVM formada por dos pequeños programas, entre los cuales se establezca una transferencia de mensajes. Mensaje sugerido: “Hola mundo”, el mensaje debe ir acompañado por:

- El nombre de la maquina que envía el mensaje
- TID (Task Identificator) o identificador de tarea.

6.2.2.2.1.2 *Justificación.* Es necesario que el estudiante inicie su experimentación en la programación con PVM desde lo más sencillo, y no por ello menos importante, como lo es la creación de una tarea y el paso de mensajes.

6.2.2.2.1.3 Objetivos

- Introducir al estudiante en la programación en PVM.

Entrenar al estudiante en todo lo relacionado con la creación de tareas y de sus respectivos envíos de mensajes.

6.2.2.2.1.4 Solución

Codigo principal

```
static char rcsid[] =
        "$Id: hello.c,v 1.2 1997/07/09 13:24:44 pvmsrc Exp $";
```

```
#include <stdio.h>
#include "pvm3.h"

main()
{
        int cc, tid;
        char buf[100];

        printf("i'm t%x\n", pvm_mytid());

        cc = pvm_spawn("hello_other", (char**)0, 0, "", 1, &tid);

        if (cc == 1) {
                cc = pvm_recv(-1, -1);
                pvm_bufinfo(cc, (int*)0, (int*)0, &tid);
                pvm_upkstr(buf);
                printf("from t%x: %s\n", tid, buf);

        } else
                printf("No puedo iniciar hello_other\n");

        pvm_exit();
        exit(0);
}
```

Código del secundario

```
static char rcsid[] =
```

```
        "$Id: hello_other.c,v 1.2 1997/07/09 13:24:45 pvmsrc Exp $";

#include "pvm3.h"

main()
{
        int ptid;
        char buf[100];

        ptid = pvm_parent();

        strcpy(buf, "hola, mundo desde ");
        gethostname(buf + strlen(buf), 64);

        pvm_initsend(PvmDataDefault);
        pvm_pkstr(buf);
        pvm_send(ptid, 1);

        pvm_exit();
        exit(0);
}
```

6.2.2.2.2 Práctica II

6.2.2.2.2.1 *Enunciado.* Crear una aplicación del tipo “Maestro/Esclavo”, donde el proceso maestro cree y dirija algún número de procesos esclavos que cooperen para hacer un trabajo.

Trabajo sugerido: Creación de tres tareas que intercambien mensajes entre si y su resultado se visualizado por pantalla.

6.2.2.2.2.2 Justificación. En la programación paralela con el fin de crear un sistema distribuido, conocer la forma de coordinar varios procesos esclavos a través de un proceso maestro es de vital importancia.

6.2.2.2.2.3 Objetivos

- Que el estudiante comprenda y domine la forma de coordinar varios procesos esclavos mediante un proceso maestro.
- Lograr en el estudiante el dominio del concepto Maestro/Esclavo.

6.2.2.2.2.4 Solución

Maestro

```
static char rcsid[] =
        "$Id: master1.c,v 1.4 1997/07/09 13:25:09 pvmsrc Exp $";
```

```
#include <stdio.h>
#include "pvm3.h"
#define SLAVENAME "slave1"

main()
{
   int mytid;
          int tids[32];
   int n, nproc, numt, i, who, msgtype, nhost, narch;
   float data[100], result[32];
   struct pvmhostinfo *hostp;
   /* enroll in pvm */
   mytid = pvm_mytid();
   pvm_config( &nhost, &narch, &hostp );
   nproc = nhost * 3;
   if( nproc > 32 ) nproc = 32 ;
          printf("Spawning %d worker tasks ... " , nproc);

    numt=pvm_spawn(SLAVENAME, (char**)0, 0, "", nproc, tids);
   if( numt < nproc ){
     printf("\n Trouble spawning slaves. Aborting. Error codes are:\n");
     for( i=numt ; i<nproc ; i++ ) {
       printf("TID %d %d\n",i,tids[i]);
     }
     for( i=0 ; i<numt ; i++ ){
       pvm_kill( tids[i] );
     }
     pvm_exit();
```

```
    exit(1);
  }
        printf("SUCCESSFUL\n");
  n = 100;
  for( i=0 ; i<n ; i++ ){
    data[i] = 1.0;
  }
        pvm_initsend(PvmDataDefault);
        pvm_pkint(&nproc, 1, 1);
        pvm_pkint(tids, nproc, 1);
        pvm_pkint(&n, 1, 1);
        pvm_pkfloat(data, n, 1);
  pvm_mcast(tids, nproc, 0);
  msgtype = 5;
  for( i=0 ; i<nproc ; i++ ){
    pvm_recv( -1, msgtype );
    pvm_upkint( &who, 1, 1 );
    pvm_upkfloat( &result[who], 1, 1 );
    printf("I got %f from %d; ",result[who],who);
    if (who == 0)
                        printf( "(expecting %f)\n", (nproc - 1) * 100.0);
    else
                        printf( "(expecting %f)\n", (2 * who - 1) * 100.0);

  }
  pvm_exit();
}
```

Esclavo

```
static char rcsid[] =
        "$Id: slave1.c,v 1.2 1997/07/09 13:25:18 pvmsrc Exp $";

#include <stdio.h>
#include "pvm3.h"

main()
{
   int mytid;
   int tids[32];
        int n, me, i, nproc, master, msgtype;
        float data[100], result;
   float work();
        mytid = pvm_mytid();
   msgtype = 0;
   pvm_recv( -1, msgtype );
        pvm_upkint(&nproc, 1, 1);
        pvm_upkint(tids, nproc, 1);
        pvm_upkint(&n, 1, 1);
        pvm_upkfloat(data, n, 1);
     for( i=0; i<nproc ; i++ )
      if( mytid == tids[i] ){ me = i; break; }

    result = work( me, n, data, tids, nproc );

    pvm_initsend( PvmDataDefault );
   pvm_pkint( &me, 1, 1 );
   pvm_pkfloat( &result, 1, 1 );
```

```
    msgtype = 5;
    master = pvm_parent();
    pvm_send( master, msgtype );

    pvm_exit();
}

float
work(me, n, data, tids, nproc )
    int me, n, *tids, nproc;
    float *data;
{
    int i, dest;
    float psum = 0.0;
    float sum = 0.0;
    for( i=0 ; i<n ; i++ ){
      sum += me * data[i];
    }
    pvm_initsend( PvmDataDefault );
    pvm_pkfloat( &sum, 1, 1 );
    dest = me+1;
    if( dest == nproc ) dest = 0;
    pvm_send( tids[dest], 22 );
    pvm_recv( -1, 22 );
    pvm_upkfloat( &psum, 1, 1 );
    return( sum+psum );
}
```

6.2.2.2.3 Práctica III

6.2.2.2.3.1 Enunciado. Realizar un ejemplo de SPMD (Single Program Multiple Data) que use pvm_siblings para determinar el número de tareas y sus TIDs.
La aplicación debe usar un simple "Token Ring" y paso de mensajes.

6.2.2.2.3.2 Justificación. Es necesario que el estudiante maneje de forma práctica el modelo SPMD, utilizado por grandes computadoras en cómputos donde se hace necesario la repetición de cálculos en varios conjuntos de datos.
PVM se convierte en una opción económica y viable para hacer la simulación de dichos sistemas. Además la forma de trabajar con "Token Ring" y paso de mensajes es muy utilizado en la sincronización en los sistemas distribuidos.

6.2.2.2.3.3 Objetivos

- Introducir al estudiante al manejo de forma practica del modelo SPMD, el cual es de vital importancia.
- Lograr que el estudiante comprenda el uso y funcionamiento del mencionado modelo.
- Ilustrar al estudiante de manera practica el concepto de Token Ring y paso de mensajes.

6.2.2.2.3.4 Solución

```
static char rcsid[] =
        "$Id: spmd.c,v 1.3 1997/07/09 13:25:19 pvmsrc Exp $";

#include <stdio.h>
#include <sys/types.h>
#include "pvm3.h"
#define MAXNPROC 32

main()
{
   int mytid;
   int *tids;
   int me;
   int i;
        int ntids;

   /* enroll in pvm */
   mytid = pvm_mytid();

        ntids = pvm_siblings(&tids);
        for (i = 0; i < ntids; i ++)
                if ( tids[i] == mytid)
                {
                        me = i;
                        break;
                }
        if (me == 0)
        {
```

```
                printf("Pass a token through the %3d tid ring:\n", ntids);
                for (i = 0; i < ntids; i ++)
                {
                        printf( "%6d -> ", tids[i]);
                        if (i % 6 == 0 && i > 0)
                                printf("\n");
                }
                printf("%6d \n", tids[0]);
        }
dowork( me, ntids, tids );
pvm_exit();
    exit(1);
}

dowork( me, nproc, tids )
    int me;
    int nproc;
        int tids[];
{
    int token;
    int src, dest;
    int count  = 1;
    int stride = 1;
    int msgtag = 4;

if ( me == 0 )
                src = tids[nproc -1];
        else
```

```
                    src = tids[me -1];
if (me == nproc - 1)
      dest = tids[0];
    else
      dest = tids[me + 1];
if( me == 0 )
    {
      token = dest;
      pvm_initsend( PvmDataDefault );
      pvm_pkint( &token, count, stride );
      pvm_send( dest, msgtag );
      pvm_recv( src, msgtag );
      printf("token ring done\n");
    }
    else
    {
      pvm_recv( src, msgtag );
      pvm_upkint( &token, count, stride );
      pvm_initsend( PvmDataDefault );
      pvm_pkint( &token, count, stride );
      pvm_send( dest, msgtag );
    }
}
```

6.2.2.2.4 Práctica IV

6.2.2.2.4.1 *Enunciado.* Desarrolle una aplicación que ilustre como medir el ancho de banda en una red a través de la utilización del conjunto de librerías que ofrece PVM.

6.2.2.2.4.2 Justificación. Resulta interesante el uso de PVM en la construcción de aplicaciones de red y más aún cuando estas se usan para obtener información de el funcionamiento de la misma red.

Además, es una forma de adiestrarse en la programación con PVM.

6.2.2.2.4.3 Objetivos

- Adiestrar al estudiante en la construcción de aplicaciones de RED, haciendo uso de un nuevo paradigma de programación (paralela).
- Que el estudiante compare el resultado de esta programación paralela con el resultado que el estudiante pudiera obtener con la programación de esta misma practica con un método secuencial.
- Afianzar en el estudiante el manejo de las librerías que ofrece PVM.

6.2.2.2.4.4 Solución

Maestro

```
static char rcsid[] =
        "$Id: timing.c,v 1.3 1997/07/09 13:26:18 pvmsrc Exp $";

#ifdef HASSTDLIB
#include <stdlib.h>
#endif
#include <stdio.h>

#ifndef WIN32
#include <sys/time.h>
#endif

#include <time.h>
#include <sys/types.h>
#include <fcntl.h>
#include <math.h>
#include "pvm3.h"


#define SLAVENAME "timing_slave"
#define ENCODING  PvmDataRaw

main(argc, argv)
        int argc;
        char **argv;
{
        int mytid;
```

```
        int stid = 0;
        int reps = 20;
        struct timeval tv1, tv2;      /* for timing */
        int dt1, dt2;
        int at1, at2;
        int numint;
        int n;
        int i;
        int *iarray = 0;

        if ((mytid = pvm_mytid()) < 0) {
                exit(1);
        }
        printf("i'm t%x\n", mytid);
        if (pvm_spawn(SLAVENAME, (char**)0, 0, "", 1, &stid) < 0 || stid < 0) {
                fputs("can't initiate slave\n", stderr);
                goto bail;
        }

pvm_setopt(PvmRoute, PvmRouteDirect);
pvm_recv( stid, 0 );
        printf("slave is task t%x\n", stid);

        puts("Doing Round Trip test, minimal message size\n");
        at1 = 0;
        pvm_initsend(ENCODING);
        pvm_pkint(&stid, 1, 1);
        puts(" N     uSec");
```

```
for (n = 1; n <= reps; n++) {
        gettimeofday(&tv1, (struct timezone*)0);

        if (pvm_send(stid, 1)) {
                fprintf(stderr, "can't send to \"%s\"\n", SLAVENAME);
                goto bail;
        }
        if (pvm_recv(-1, -1) < 0) {
                fprintf(stderr, "recv error%d\n" );
                goto bail;
        }
        gettimeofday(&tv2, (struct timezone*)0);
        dt1 = (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec - tv1.tv_usec;
        printf("%2d %8d\n", n, dt1);
        at1 += dt1;
}
printf("RTT Avg uSec %d\n", at1 / reps);

puts("\nDoing Bandwidth tests\n");
for (numint = 25; numint < 1000000; numint *= 10) {
        printf("\nMessage size %d\n", numint * 4);
        at1 = at2 = 0;
        iarray = (int*)malloc(numint * sizeof(int));
        puts(" N  Pack uSec  Send uSec");
        for (n = 1; n <= reps; n++) {
                gettimeofday(&tv1, (struct timezone*)0);

                pvm_initsend(ENCODING);
```

```
            pvm_pkint(iarray, numint, 1);
            gettimeofday(&tv2, (struct timezone*)0);
            dt1 = (tv2.tv_sec - tv1.tv_sec) * 1000000
                    + tv2.tv_usec - tv1.tv_usec;
            gettimeofday(&tv1, (struct timezone*)0);

            if (pvm_send(stid, 1)) {
                    fprintf(stderr, "can't send to \"%s\"\n", SLAVENAME);
                    goto bail;
            }
            if (pvm_recv(-1, -1) < 0) {
                    fprintf(stderr, "recv error%d\n" );
                    goto bail;
            }
            gettimeofday(&tv2, (struct timezone*)0);
            dt2 = (tv2.tv_sec - tv1.tv_sec) * 1000000
                    + tv2.tv_usec - tv1.tv_usec;
            printf("%2d   %8d   %8d\n", n, dt1, dt2);
            at1 += dt1;
            at2 += dt2;
    }
    if (!(at1 /= reps))
            at1 = 1;
    if (!(at2 /= reps))
            at2 = 1;
    puts("Avg uSec");
    printf("     %8d   %8d\n", at1, at2);
    puts("Avg Byte/uSec");
```

```
                printf("     %8f   %8f\n",
                        (numint * 4) / (double)at1,
                        (numint * 4) / (double)at2);
        }

        pvm_freebuf(pvm_getsbuf());
        puts("\ndone");

bail:
        if (stid > 0)
                pvm_kill(stid);
        pvm_exit();
        exit(1);
}
```

Esclavo

```
static char rcsid[] =
        "$Id: timing_slave.c,v 1.2 1997/07/09 13:26:19 pvmsrc Exp $";

#include <sys/types.h>
#include <fcntl.h>
#include <stdio.h>
#include "pvm3.h"

#define ENCODING  PvmDataRaw
```

```
main(argc, argv)
        int argc;
        char **argv;
{
        int mytid
        int dtid;
        int bufid;
        int n = 0;

        mytid = pvm_mytid();

   pvm_setopt(PvmRoute, PvmRouteDirect);
   pvm_initsend(ENCODING);
   pvm_send( pvm_parent(), 0 );

        pvm_initsend(ENCODING);
        pvm_pkint(&mytid, 1, 1);


        while (1) {
                bufid = pvm_recv(-1, -1);
                pvm_bufinfo(bufid, (int*)0, (int*)0, &dtid);
                pvm_freebuf(pvm_getrbuf());
                pvm_send(dtid, 2);
/*
                printf("echo %d\n", ++n);
*/
```

```
        }
    }
```

CONCLUSIONES

Después de varios meses de estudio y desarrollo de las prácticas relacionadas con los sistemas distribuidos planteadas anteriormente podemos decir lo siguiente:

- Los sistemas distribuidos estan actualmente en un proceso de desarrollo debido a que su construcción necesita de algoritmos más elaborados lo cual aumenta el tiempo que se necesita para desarrollar un producto que cumpla con todas las especificaciones básicas de estos sistemas y para asegurar su correcto funcionamiento. Además, en la actualidad no se dispone de mucha experiencia en el diseño, implantación y uso de sistemas distribuidos, tampoco se conoce que lenguajes de programación y aplicaciones son adecuados para estos sistemas, no hay un acuerdo entre que tanto debe saber el usuario de la distribucion y que tanto debe hacer el sistema sin ayuda del usuario. Los expertos tienen sus difernecias. Sín embargo se han creado sistemas operativos con importantes aproximaciones al total de características que supone un sistema como este. Ejemplos de estos sistemas son Amoeba, Mach, Chorus.

- En el camino hacia el logro de un sistema completamente distribuido se han creado modelos que su estudio permite conocer en parte como sería el

funcionamiento de un sistema distribuido como por ejemplo los modelos de Cliente/Servidor y RPC. Dicho estudio permite identificar ejemplos donde se han introducido estos modelos y que resultan hoy día algo de uso frecuente por parte de un usuario final de una computadora como es el caso de la WWW, los cajeros automáticos, etc.

- En teoría el trabajo con un sistema distribuido supone la posibilidad de trabajar con sistemas de procesamiento en paralelo, cuyo estudio simulado se puede lograr a través de la utilización de un conjunto de librerías como es el caso de PVM.

Por otra parte, una vez finalizado nuestro trabajo de grado hemos querido hacer un cuadro comparativo para mostrar nuestros logros enfrentados con lo planteado inicialmente en el anteproyecto.

Objetivo	**Logro**
Montar un servidor de LINUX que sirva como base para la implementación de las prácticas del laboratorio de sistemas distribuidos y desarrollo de proyectos futuros.	Se instaló la distribución Linux Red Hat 6.0 en un equipo brindado por la Universidad. Las especificaciones técnicas de la máquina son un poco limitadas pero se pudieron probar satisfactoriamente en ella toas las prácticas. La máquina esta actualmente en red junto

	con otras 6 máquinas "Windows" aproximadamente.
El número de prácticas que proponemos es de la siguiente manera: ➢ RPC: 5 prácticas ➢ Cliente/Servidor: 5 prácticas ➢ Comunicación en grupo: 5 prácticas ➢ PVM: 5 prácticas	 Sí. Sí. 1 práctica 4 prácticas
Proponemos como base de procedimiento de estas prácticas una estructura que ayude al estudiante en el desarrollo de las mismas, de la siguiente forma: ✓ Sustentación teórica ✓ Objetivos por alcanzar ✓ Contenido de la práctica en sí ✓ Cuestionario de retroalimentación final.	Sí.
Manual del docente y del estudiante	Sí.
Documento de tesis	Sí.
Librerías y software existente, debidamente documentadas, que permitan la realización de cada una de las prácticas	Sí.

(Formato CD).	

BIBLIOGRAFÍA

ABAD, Alfredo y MADRID, Mariano. *Redes de Area Local*.1997. McGraw-Hill. 238p.

COMER, Douglas E. *TCP/IP: principios básicos, protocolos y arquitectura*. 1995. McGraw-Hill. 524p.

TANENBAUM, Andrew. Sistemas operativos distribuidos. 1996. Prentice Hall. 617p.

ROSEN, Kenet *et al*. UNIX sistema V versión 4. 1996, McGraw-Hill. 820p.

MARQUEZ G, Fco. Manuel. UNIX programación avanzada. 1995. McGraw-Hill. 180p.

RISSLET, Jean Marie. Comunicaciones en UNIX. 1996. McGraw-Hill. 170p.