47

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

Facultad de Ingeniería

DESARROLLO DE UN SISTEMA DE PROCESAMIENTO DISTRIBUIDO USANDO BIBLIOTECAS PARA LA PROGRAMACION PARALELA EN EL MODELO DE INTERCAMBIO DE MENSAJES

T E S I S

Que para obtener el título de

INGENIERO EN COMPUTACION

P r e s e n t a:

TANIA XITLALI POBLETE MUÑOZ

DIRECTORA DE TESIS:

Ing. Laura Sandoval Montaño

MEXICO, D.F. OCTUBRE 2001

DEDICATORIAS

El desarrollo de esta tesis implica muchas cosas; el término de una etapa más de la vida, así como el inicio de otra; la culminación de un esfuerzo personal, un logro familiar, etc.

Pero lo más importante es que al pasar el tiempo te das cuenta que las bases que se formaron en la parte inicial de tu vida, son pilares importantes para el desarrollo de las siguientes etapas, es por esto, que quiero agradecer y dedicar este trabajo a dos personas muy importantes para mí.

"Papá y Mamá", no será suficiente el tiempo para agradecerles todo el apoyo, dedicación y motivación. Siempre serán fuente de orgullo y admiración.
Este logro es para ustedes.

AGRADECIMIENTOS

En cada una de las etapas de la vida, hay personas importantes, unas son transitorias, otras permanecen a lo largo de la vida. A todas ellas quiero agradecerles.

Marce, por tu comprensión y apoyo, por escucharme, por aguantarme. Por ser mi hermana. Te quiero. Gracias.

Abuelita Mila, gracias por todos esos cuidados y preocupaciones que desde niña has tenido conmigo.

Familia en Chile, porque yo sé, que a la distancia me han apoyado y han estado conmigo siempre, por enviar esos buenos deseos y palabras de aliento.

Familia en México, aunque no es de sangre, han sido un gran apoyo en todo momento y parte de una verdadera familia.

Rolando, gracias por todo ese apoyo, cariño, amor y comprensión que me has dado siempre que lo he necesitado. Gracias por estar ahí. Por ser esencial para mí.

Rafa, por ser un amigo tan importante e incondicional. Por ser tan especial.

Sheila, Aida, Dulce, Martha, por esos grandes momentos que hemos pasado juntas y esperando que sean muchos más.

Alejandro, Fabian, Omar, y Bertha, por el apoyo en mi vida académica, gracias.

Daniel, por el apoyo en la elaboración de este trabajo.

Todos ustedes, amigos y familiares, han formado, forman y seguirán formando parte de mí.

A mi asesora, Ing. Laura Sandoval.

Por la confianza y paciencia que ha tenido conmigo.

Por ser más que una asesora, por haber encontrado a una gran amiga.

A mis profesores.

Porque son el medio de transmisión de conocimientos en esta etapa.

A mi Universidad y Facultad de Ingeniería.

Por permitirme realizar una etapa básica en la vida de una persona, la formación profesional.

A Dios.

Por que ha estado presente en los momentos difíciles.

ÍNDICE TEMÁTICO

Los sistemas de computación han estado sujetos a cambios tecnológicos de una forma acelerada a partir de la década de los ochenta. Dos de los avances más importantes, han sido el desarrollo de poderosos microprocesadores a un costo menor y el desarrollo de redes de área local de alta velocidad, donde la interconexión correcta de cientos de computadoras y la instalación de software adecuado, juegan un papel importante en la transferencia de información en pequeñas fracciones de tiempo.

Como resultado de esto, se ha buscado que las tareas o procesos a realizar en una red de computadoras, se agilicen y optimicen, naciendo lo que se conoce como procesamiento distribuido.

Basándose en este tipo de procesamiento y de las ventajas que nos proporciona, como son mejoras en el ámbito costo-eficiencia, potencia del procesador e independencia de una máquina central, se pretende desarrollar un sistema de procesamiento distribuido que trabaje con los recursos que se tienen, buscando resolver rápida y eficientemente la repartición de carga de trabajo en los procesadores.

En un sistema de procesamiento distribuido, los procesadores se encuentran en diferentes máquinas conectadas mediante una red. Definiremos entonces, que un sistema de procesamiento distribuido es un conjunto de computadoras independientes, de arquitectura de bajo costo, conectadas en red y que aparentan ser una sola máquina.

Por otro lado tenemos, el concepto de Cluster. Los Clusters o cúmulos, son un conjunto de máquinas de arquitecturas homogéneas o heterogéneas conectadas en red bajo cualquier topología. Estas máquinas trabajan juntas como un sistema único. Es aquí donde podemos decir que un Cluster es un sistema de procesamiento distribuido.

Los clusters pueden ser clasificados en varias categorías con base en diversos factores. Existen dos tipos de clusters según su aplicación:

- Cluster de alto rendimiento (High Performance), cuya utilización primordial se enfoca a aplicaciones científicas y de ingeniería donde se necesitan miles de millones de operaciones de punto flotante (flops). Ejemplo Beowulf.

- Cluster de alta disponibilidad (High Availability), donde el objetivo principal es mantener el nivel de servicio (disponibilidad) del sistema.

En este caso, se trabajará con clusters de alta disponibilidad, específicamente, con clusters no dedicados, es decir, cada nodo podrá dar servicios a usuarios de manera independiente.

Podemos decir ahora, que el objetivo al que se quiere llegar con la elaboración de este trabajo, es realizar el análisis y diseño de un sistema de procesamiento distribuido. Este sistema pretende distribuir la carga de trabajo eficientemente en una red LAN de computadoras personales usando bibliotecas de libre acceso para la programación paralela y distribuida con los recursos de hardware disponibles.

Este trabajo está dividido en dos partes principales, una parte teórica y una parte práctica, en esta última se hace el análisis y diseño del sistema. Dichas partes a su vez, se han dividido en capítulos, para organizarlo de una mejor manera.

De esta forma se tiene que la sección teórica está formado por tres capítulos. El Capítulo 1 "Los sistemas operativos y sus tendencias", presenta la definición, historia, conceptos básicos y clasificación de los sistemas operativos.

En el Capítulo 2 "Los sistemas operativos distribuidos" se asientan los conceptos teóricos necesarios para el entendimiento de un sistema distribuido, como son aspectos de hardware y software para la construcción del cluster, conceptos de redes y de programación paralela. Este capítulo también se enfoca a la administración de procesos, además de hacer mención a las características del sistema operativo que se usará, Linux.

El último capítulo de esta sección, el Capítulo 3 "Programación paralela en el modelo de intercambio de mensajes", presenta los fundamentos teóricos de los diferentes mecanismos para dicha programación, y se colocan las bases teóricas de las bibliotecas mencionadas en este trabajo, que son PVM y MPI.

La segunda sección está formada por dos capítulos, el primero, Capítulo 4 "La construcción de la plataforma del sistema", hace una descripción de los pasos que se llevaron a cabo en la construcción; conexiones en hardware y software, divididas en servidor y cliente

El Capítulo 5 "Análisis y Diseño del Sistema de Procesamiento distribuido" es el último capítulo de este trabajo y es donde se hace el análisis y diseño del funcionamiento de dicho sistema; se hace uso de un lenguaje unificado de modelado (UML), pretendiendo dejar asentadas las bases para su posterior construcción. Al final de éste, se muestran algunos ejemplos realizados en PVM y se mencionan dos herramientas que tienen muchas de las características mencionadas en el análisis y diseño elaborado.

1 LOS SISTEMAS OPERATIVOS Y SUS TENDENCIAS

1.1 DEFINICIÓN DE SISTEMA OPERATIVO

Una computadora está formada por elementos de hardware (componentes físicos) y elementos de software (elementos lógicos e intangibles).

El hardware se divide, de acuerdo a la función que desarrolla, en entrada, que son los componentes fuera de la Unidad Central de Proceso (CPU) que proporcionan información e instrucciones; salida, aquellos dispositivos externos que transfieren información del CPU al usuario; y almacenamiento, que es el lugar donde se guarda la información y programas.

El software está dividido en programas de sistema, que son aquellos que manejan la operación de la computadora y en programas de aplicación, que resuelven problemas específicos de los usuarios.

El sistema operativo es un programa de sistema, que actúa como intermediario entre el usuario de una computadora y el hardware de la misma, buscando la eficiencia en el desarrollo de sus tareas.

Se caracteriza por definir la interfaz de usuario, planificar los recursos y permitir que los usuarios compartan el sistema físico y los datos. En otras palabras, el sistema operativo tiene la tarea de ofrecer una distribución ordenada y controlada de los procesadores, memoria y dispositivos de entrada/salida entre los diversos programas que compiten por éstos.

1.2 HISTORIA DE LOS SISTEMAS OPERATIVOS

Los sistemas operativos han evolucionado a través de los años, al cambiar la arquitectura de las computadoras; a continuación se hará una pequeña descripción de las generaciones de los sistemas operativos.

1.2.1 Primera generación (1945-1955)

Después de los esfuerzos frustrados de Babbage por construir una máquina analítica, se progresó en la construcción de computadoras digitales. Howard Aiken, John Von Neumann, entre otros, obtuvieron resultados óptimos en la construcción de máquinas de cálculo mediante el uso de tubos de vacío.

Mientras tanto, otro grupo de personas diseñó, construyó, programó y dió mantenimiento a cada máquina. Toda la programación se realizó en lenguaje de

máquina absoluto, a menudo alambrando tableros enchufables para controlar las funciones básicas de la máquina. Los sistemas operativos eran extraños, el modo usual de operación consistía en que el programador firmaba para tener acceso a un bloque de tiempo en la hoja de registro, entraba al cuarto de máquinas, insertaba su tablero enchufable en la computadora y pasaba las siguientes horas, esperando a que algún bulbo se fundiera durante la ejecución de su programa.

Cabe mencionar que el programador debía tener un conocimiento y contacto profundo con el hardware para determinar la causa del fallo y poder corregir su programa, además de enfrentarse nuevamente a los procedimientos de apartar tiempo del sistema y poner a punto los compiladores, ligadores y todo lo necesario; para volver a correr su programa, es decir, enfrentaba el problema del procesamiento serial.

Al inicio de la década de 1950, la rutina había mejorado un poco con la introducción de tarjetas perforadas.

1.2.2 Segunda generación (1955-1965)

La introducción del transistor a mediados de la década de 1950 provocó que las computadoras se volvieran confiables. Para correr un trabajo, un programador primero escribía el programa en papel y después lo perforaba en tarjetas para posteriormente entregárselo a uno de los operadores y esperar el término de la ejecución.

Se desperdiciaba mucho tiempo y el costo de las computadoras era alto. Por lo que rápidamente se adoptó el sistema de lote, que buscaba reducir el tiempo perdido mediante la idea de conjuntar una serie de trabajos para leerlos e imprimir las salidas. Se contaba con una unidad de cinta; mientras que el operador cargaba un programa especial, el cual leía el primer trabajo de la cinta y lo ejecutaba, la salida se escribía en una segunda cinta, en vez de imprimirse. Después de terminar cada trabajo el sistema operativo leía automáticamente el siguiente trabajo y realizaba el mismo procedimiento. Al terminarse el lote, el operador retiraba las cintas de entrada o salida, sustituyendo la cinta de entrada por el siguiente lote y la imprimía fuera de línea.

Las grandes computadoras de la segunda generación se utilizaban en su mayor parte para realizar cálculos científicos y de ingeniería. Se programaban principalmente en FORTRAN y en lenguaje ensamblador. Los sistemas operativos comunes eran FMS, (Fortran Monitor System, sistema monitor de Fortran) e IBSYS, sistema operativo de IBM.

1.2.3 Tercera generación (1965-1980)

Se construyó la primera línea de computadoras que usó circuitos integrados (de pequeña escala), ofreciendo una mayor ventaja de precio/rendimiento sobre las máquinas de la segunda generación, que se construían a partir de transistores individuales. Se pretendía que todo el software, como el sistema operativo, debía funcionar en todos los modelos; tenían que correr en sistemas pequeños para copiar o reproducir tarjetas en cinta, y en sistemas muy grandes para realizar predicciones climatológicas y otras operaciones complejas.

Además debían funcionar en medios comerciales y científicos. Sobre todo, tenían que ser eficientes para todos los diferentes usos. El resultado fue, un sistema operativo enorme y extraordinariamente complejo, el OS/360 de IBM. Constaba de millones de líneas de lenguaje ensamblador escritas por miles de programadores, y contenía miles y miles de errores ocultos, que necesitaban un flujo continuo de nuevas liberaciones.

Se popularizó la multiprogramación, que consistía en dividir la memoria en varias partes, con trabajo diferente en cada partición; por ejemplo, mientras un trabajo esperaba a que se completara la entrada/salida, otro trabajo podía estar utilizando el CPU. Mientras se mantuvieran suficientes trabajos en la memoria central se pensaba que el CPU se aprovechaba de una mejor manera.

Otra característica que está presente en los sistemas operativos de esta generación es la capacidad de leer trabajos de tarjetas contenidas en el disco tan pronto como salían del cuarto de máquinas. Por lo tanto, siempre que se terminaba un trabajo, el sistema operativo podía cargar uno nuevo del disco en la partición no vacía y ejecutarlo. Esta técnica se denomina operaciones periféricas simultáneas en línea (*spool*, por sus siglas en inglés).

El deseo de tener la computadora para ellos solos por unas cuantas horas para depurar sus programas con rapidez, marcó el camino para el tiempo compartido, (variante de la multiprogramación), donde cada usuario tenía una terminal en línea.

1.2.4 Cuarta generación (1980-1990)

Con la creación de los circuitos LSI (integración a grande escala), chips que contienen millones de transistores en un centímetro cuadrado de silicón; los precios de las computadoras comenzaron a bajar, haciendo posible que el departamento de una compañía o una universidad tuviera la propia.

Se comenzó a producir software para computadoras personales, que era amable con el usuario, es decir, estaba dirigido a usuarios que no conocían nada acerca de las computadoras y además, que no tenían la menor intención de aprender.

Dos sistemas operativos dominaron la escena de la computadora personal: MS-DOS, escrito por Microsoft, usado por máquinas con procesadores Intel y sus sucesores y UNIX, que dominaba en las computadoras personales que hacen uso de la familia de CPU Motorola.

Un avance interesante fue el desarrollo de redes de computadoras personales que corren sistemas operativos en red y sistemas operativos distribuidos. En un sistema operativo en red, los usuarios tienen conocimiento de la existencia de múltiples máquinas. Cada máquina ejecuta su sistema operativo local y tiene un número propio. En cambio, un sistema operativo distribuido, es aquél que se presenta ante usuarios como un sistema uniprocesador tradicional, aunque en realidad esté compuesto de múltiples procesadores. En un sistema distribuido real, los usuarios no tienen conocimiento de dónde se están ejecutando sus programas o de dónde están ubicados sus archivos; todo esto se debe manejar en forma automática y eficiente por medio del sistema operativo.

1.2.5 Quinta generación (1990-presente)

En la quinta generación se tiene el procesamiento en paralelo mediante arquitecturas y diseños especiales y circuitos de gran velocidad. Así como procesamientos heterogeneos.

También se maneja lenguaje natural y sistemas de inteligencia artificial (campo de estudio que trata de aplicar los procesos del pensamiento humano usados en la solución de problemas a la computadora).

1.3 CONCEPTOS BÁSICOS DE SISTEMAS OPERATIVOS

La interfaz entre el sistema operativo y los programas del usuario se define por medio del conjunto de "instrucciones extendidas" que el sistema operativo proporciona. Estas instrucciones extendidas se conocen como llamadas al sistema. Las llamadas al sistema se clasifican en dos categorías generales: aquéllas que se relacionan con procesos y las que lo hacen con el sistema de archivo.

1.3.1 Procesos

Las llamadas al sistema de manejo de proceso, son aquellas que tienen que ver con la creación y terminación de procesos.

Un proceso es básicamente un programa en ejecución. En muchos sistemas operativos, toda la información referente a cada proceso, se almacena en una

tabla de sistema operativo llamada tabla de proceso, la cual es un arreglo (o lista enlazada) de estructuras, una para cada proceso en existencia corriente.

Por lo tanto, un proceso (suspendido) consta de su espacio de direcciones, generalmente denominado imagen de núcleo y su registro de la tabla de procesos, que contiene información acerca del estado del proceso, su contador de programa, apuntador de pila, distribución de la memoria, la condición de sus archivos abiertos, su información de rendición de cuentas y planificación, y todo lo demás referente al proceso que debe guardarse cuando el proceso se cambia del estado de ejecución a listo de manera que pueda reiniciarse después como si nunca se hubiese detenido.

Estados de los procesos

Los tres estados en que puede encontrarse un proceso son:

- Ejecución: que en realidad hace uso del CPU en ese instante.
- Bloqueado: incapaz de correr hasta que suceda algún evento externo.
- Listo: ejecutable, se detiene temporalmente para permitir que se ejecute otro proceso.

En estos tres estados, son posible cuatro transiciones. La transición 1 ocurre cuando un proceso descubre que no puede continuar. La transición 2 ocurre cuando el planificador decide que el proceso en ejecución ya ha corrido el tiempo suficiente y es tiempo de permitir que otro proceso tome tiempo en el CPU. La transición 3 ocurre cuando todos los otros procesos han utilizado parte del tiempo y es hora de que el primer proceso vuelva a correr. Las transacciones 2 y 3 son ocasionadas por el planificador del proceso, que es parte del sistema operativo, sin que el proceso llegue a saber de ellas.

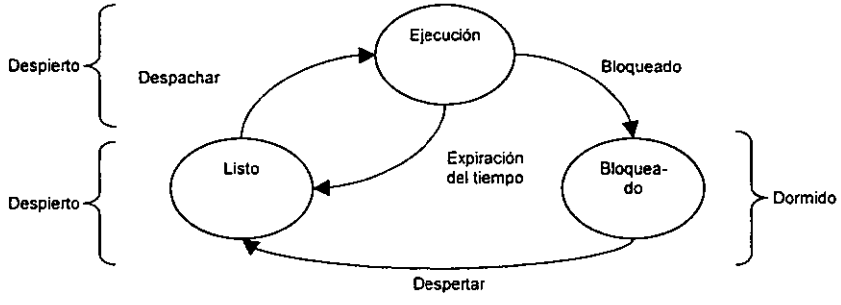

Figura 1. Transiciones de estados de los procesos

La transición 4 ocurre cuando aparece el evento externo que estaba esperando un proceso (como el arribo de alguna entrada). Si ningún otro proceso corre en ese instante, la transición 3 se activará de inmediato y el proceso iniciará

su ejecución. De lo contrario quizá tenga que esperar en estado de listo un poco más hasta que el CPU esté disponible.

Al utilizar este modelo de procesos, se vuelve mucho más sencillo pensar en lo que sucede dentro del sistema. Algunos de los procesos ejecutan programas que son comandos tecleados por un usuario, otros son parte del sistema y realizan tareas como solicitudes de servicio de archivo o el manejo de los detalles de la ejecución de un disco o de una unidad de cinta. Cuando ocurre una interrupción del disco, el sistema toma la decisión de suspender la ejecución del proceso corriente y corre el proceso del disco, que se bloqueó cuando esperaba esa interrupción. Por tanto, en vez de pensar en interrupciones, se puede considerar procesos de usuarios, procesos de disco, procesos de terminales, etc.

Entonces, el estrato inferior de un sistema operativo estructurado por procesos es el planificador, que maneja las interrupciones y los detalles del inicio y suspensión reales de los procesos.

1.3.2 Archivos

La otra categoría de llamadas al sistema se relaciona con el sistema de archivos. Una función importante del sistema operativo consiste en ocultar las peculiaridades de los discos y otros dispositivos de entrada/salida y presentar al programador un modelo abstracto limpio y agradable de archivos independientes del dispositivo. Las llamadas al sistema se necesitan para crear, eliminar, leer y escribir archivos.

Las jerarquías de procesos y archivos se organizan ambas como árboles, pero la similitud llega hasta aquí. Las jerarquías de procesos por lo general no son muy grandes, mientras que las jerarquías de archivos tienen más de dos niveles de extensión. Las jerarquías de procesos regularmente duran poco, mientras que las jerarquías de directorios pueden existir por años. La propiedad y la protección también difieren en los procesos y los archivos.

Todos y cada uno de los archivos que son tenidos en la jerarquía del directorio pueden especificarse dando su nombre de trayectoria desde la parte superior de la jerarquía del directorio, (el directorio raíz).

1.4 CLASIFICACIÓN DE LOS SISTEMAS OPERATIVOS

Los sistemas operativos según sus características se pueden clasificar en:

- Sistemas operativos por su estructura (visión interna)
- Sistemas operativos por los servicios que ofrecen
- Sistemas operativos por la forma en que ofrecen sus servicios (visión externa).

1.4.1 Sistemas operativos por su estructura

Tomando en cuenta dos tipos de requisitos cuando se construye un sistema operativo:

Requisitos de usuario: Sistema fácil de usar y de aprender, seguro, rápido y adecuado al uso al que se le quiere destinar.

Requisitos del software: Donde se engloban aspectos como el mantenimiento, forma de operación, restricciones de uso, eficiencia, tolerancia frente a los errores y flexibilidad.

A continuación se describen las distintas estructuras que presentan los actuales sistemas operativos para satisfacer las necesidades que de ellos se quieren obtener.

Estructura monolítica

Es la estructura de los primeros sistemas operativos, constituidos fundamentalmente por un solo programa, compuesto de un conjunto de rutinas entrelazadas de tal forma que cada una puede llamar a cualquier otra. La característica fundamental es que, al construir el programa objeto del sistema operativo se deben compilar todos los procedimientos individuales o archivos que contienen los procedimientos y después se combinan todos en un solo archivo objeto con el enlazador. En términos de ocultamiento de información, esencialmente no existe ninguno; todo procedimiento es visible para todos.

Sin embargo, aun en sistemas monolíticos es posible tener cuando menos una pequeña estructura. Los servicios (llamadas al sistema) proporcionados por el sistema operativo se solicitan al colocar los parámetros en sitios bien definidos, como en registros o en la pila y después, ejecutar una institución de trampa especial conocida como llamada de kernel o llamadas supervisora.

Esta instrucción cambia la máquina del modo de usuario al modo kernel y transfiere el control al sistema operativo. El sistema operativo examina después los parámetros de la llamada para determinar qué llamada al sistema se efectuará. Después el sistema operativo indiza en una tabla que contiene en el canal k un apuntador al procedimiento que realiza la llamada al sistema k. Por último, se termina la llamada al sistema y el control se devuelve al programa del usuario.

Esta organización sugiere una estructura básica del sistema operativo:

1. Un programa central que invoque el procedimiento de servicio solicitado
2. Un conjunto de procedimientos de servicios que realice las llamadas al sistema
3. Un conjunto de procedimientos de uso general que ayude a los procedimientos de servicio.

Generalmente están hechos a medida, por lo que son eficientes y rápidos en su ejecución y gestión, pero por lo mismo carecen de flexibilidad para soportar diferentes ambientes de trabajo o tipos de aplicaciones.

Estructura jerárquica

A medida que fueron creciendo las necesidades de los usuarios y se perfeccionaron los sistemas, se hizo necesaria una mayor organización del software, del sistema operativo, donde una parte del sistema contenía subpartes y esto organizado en forma de niveles. Se dividió el sistema operativo en pequeñas partes, de tal forma que cada una de ellas estuviera perfectamente definida y con una clara interfaz con el resto de elementos.

Se constituyó una estructura jerárquica o de niveles en los sistemas operativos, el primero de los cuales fue denominado THE (Technische Hogeschool, Eindhoven), de Dijkstra, que se utilizó con fines didácticos. Se puede pensar también en estos sistemas como si fueran 'multicapa'.

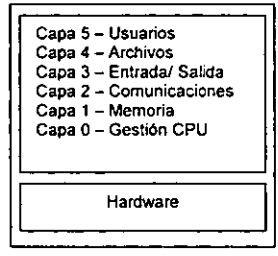

Figura 2. Sistema Jerárquico THE

En la estructura anterior se basan prácticamente la mayoría de los sistemas operativos actuales. Otra forma de ver este tipo de sistema es la denominada de anillos concéntricos o "rings"

El sistema tenía 6 estratos. El estrato 0 trabajaba con la distribución del procesador, cambiando entre procesos cuando ocurrían interrupciones o los relojes expiraban. Sobre el estrato 0, el sistema constaba de procesos secuenciales, cada uno de los cuales podía programarse sin tener que preocuparse por el hecho de que múltiples procesos estuvieran corriendo en un solo procesador. En otras palabras, el estrato 0 ofrecía la multiprogramación básica del CPU.

El estrato 1 realizaba el manejo de la memoria. Éste distribuía espacio para procesos contenidos en la memoria central y en un tambor de 512 K palabras, que se usaba para contener partes de procesos (páginas), para las cuales no había espacio en la memoria central, el software del estrato 1 se hacía cargo de asegurar que las páginas se trajeran a la memoria siempre que se necesitaran.

El estrato 2 manejaba la comunicación entre cada proceso y la consola del operador. El estrato 3 se hacía cargo de manejar los dispositivos de E/S y de separar la información en flujo que entraba y salía de ellos. El estrato 4 era donde se encontraban los programas de los usuarios.

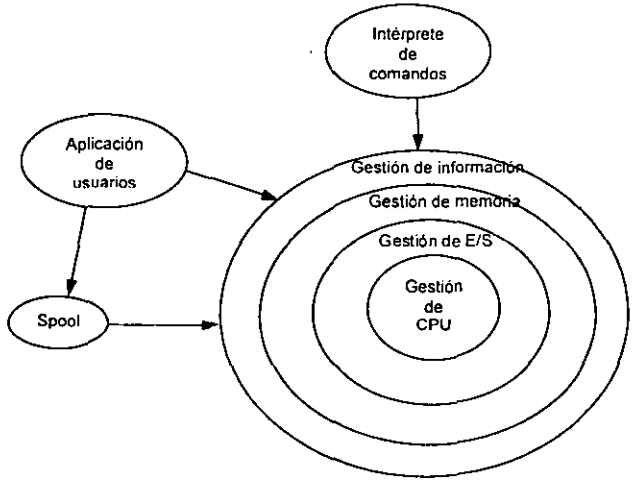

Figura 3. Organización jerárquica (anillos)

En el sistema de anillos, cada uno tiene una apertura, conocida como puerta o trampa (trap), por donde pueden entrar las llamadas de las capas inferiores. De esta forma, las zonas más internas del sistema operativo o núcleo del sistema estarán más protegidas de accesos no deseados desde las capas más externas. Las capas más internas serán, por tanto, más privilegiadas que las externas.

Máquina Virtual.

Se trata de un tipo de sistema operativo que presenta una interfaz a cada proceso, mostrando una máquina que parece idéntica a la máquina real subyacente. Estos sistemas operativos separan dos conceptos que suelen estar unidos en el resto de sistemas: la multiprogramación y la máquina extendida. El objetivo de los sistemas operativos de máquina virtual es el de integrar distintos sistemas operativos dando la sensación de ser varias máquinas diferentes.

El núcleo de estos sistemas operativos se denomina monitor virtual y tiene como misión llevar a cabo la multiprogramación, presentando a los niveles superiores tantas máquinas virtuales como se soliciten. Estas máquinas virtuales no son máquinas extendidas, sino una réplica de la máquina real, de manera que

en cada una de ellas se pueda ejecutar un sistema operativo diferente, que será el que ofrezca la máquina extendida al usuario.

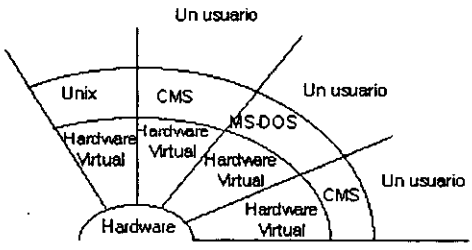

Figura 4. Máquina Virtual

Puesto que cada máquina virtual es idéntica al hardware real, cada una puede correr cualquier sistema operativo que correrá directamente en el hardware

Cliente-servidor (Microkernel)

El tipo más reciente de sistema operativo es el denominado Cliente-servidor, que puede ser ejecutado en la mayoría de las computadoras, ya sean grandes o pequeñas.

Este sistema sirve para toda clase de aplicaciones por tanto, es de propósito general y cumple con las mismas actividades que los sistemas operativos convencionales.

Una tendencia en los sistemas operativos modernos consiste en tomar esta idea de desplazar código a estratos superiores, pero en mayor proporción, y eliminar lo más que sea posible del sistema operativo, dejando un kernel mínimo. El método general consiste en implementar la mayoría de las funciones del sistema operativo en procesos del usuario. Para solicitar un servicio, como la lectura de un bloque de un archivo, un proceso de usuario (o proceso cliente) envía la solicitud a un proceso servidor, que después realiza el trabajo y devuelve la respuesta.

En este modelo, todo lo que el kernel hace es manejar la comunicación entre clientes y servidores. Al dividir el sistema operativo en partes, cada una de las cuales se encarga del manejo de una faceta del sistema, como el servicio del archivo, el servicio del proceso, servicio de la terminal o el servicio de la memoria, cada parte se vuelve pequeña y difícil de manejar. Además, como todos los servidores corren como procesos en modo de usuario, y no en modo de kernel, no tienen acceso directo al hardware. Como consecuencia, si se activa un error oculto

en el servidor de archivo, el servicio al archivo puede fracasar pero esto por lo general no hará fallar a toda la máquina

Otra ventaja del modelo del servidor del cliente es su adaptabilidad para utilizarse en sistemas distribuidos. Si un cliente se comunica con un servidor mediante la emisión de mensajes, el cliente no necesita saber si el mensaje se maneja en forma local o en su máquina, o bien si se envió a través de una red a un servidor en una máquina remota. Hasta donde concierne al cliente, sucede lo mismo en ambos casos, envió una solicitud y se devolvió una contestación.

La opción de un kernel de manejar sólo el transporte de mensajes de clientes a servidores y de regreso no es completamente real. Algunas funciones del sistema operativo son complicadas si no es que imposibles de ejecutar a partir de programas en el espacio del usuario.

Existen dos maneras de manejar este problema. Una de ellas consiste en hacer que algunos procesos importantes corran en realidad en modo kernel, con acceso completo a todo el hardware, pero que se siga comunicando con otros procesos mediante el uso del mecanismo de mensajes normal.

La otra manera consiste en construir una mínima cantidad de mecanismo en el kernel, pero dejar las decisiones de política a los servidores en el espacio del usuario. Por ejemplo, el kernel podría reconocer que un mensaje enviado a cierta dirección significa tomar el contenido de ese mensaje y cargarlo en los registros del dispositivo de E/S de algún disco, a fin de iniciar una lectura del disco. En este ejemplo, el kernel no llegaría a inspeccionar los bytes del mensaje para ver si eran válidos o significativos; simplemente los copia en los registros de dispositivo del disco.

El núcleo tiene como misión establecer la comunicación entre los clientes y los servidores. Los procesos pueden ser tanto servidores como clientes. Por ejemplo, un programa de aplicación normal es un cliente que llama al servidor correspondiente para acceder a un archivo o realizar una operación de entrada/salida sobre un dispositivo concreto. A su vez, un proceso cliente puede actuar como servidor para otro. Este paradigma ofrece gran flexibilidad en cuanto a los servicios posibles en el sistema final, ya que el núcleo provee solamente funciones muy básicas de memoria, entrada/salida, archivos y procesos, dejando a los servidores proveer la mayoría que el usuario final o programador puede usar. Estos servidores deben tener mecanismos de seguridad y protección que, a su vez, serán filtrados por el núcleo que controla el hardware.

1.4.2 Sistemas Operativos por Servicios

Esta clasificación es la más comúnmente usada y conocida desde el punto de vista del usuario final.

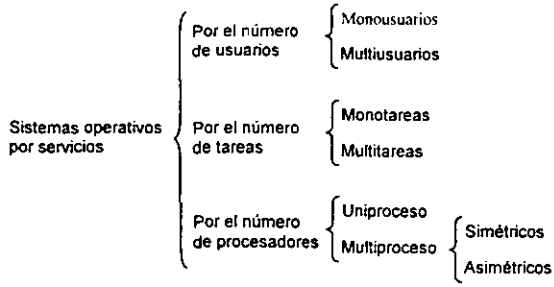

Figura 5. Sistemas operativos por servicios

Monousuarios

Los sistemas operativos monousuarios son aquéllos que soportan a un usuario a la vez, sin importar el número de procesadores que tenga la computadora o el número de procesos o tareas que el usuario pueda ejecutar en un mismo instante de tiempo. Las computadoras personales típicamente se han clasificado en este renglón.

Multiusuarios

Los sistemas operativos multiusuarios son capaces de dar servicio a más de un usuario a la vez, ya sea por medio de varias terminales conectadas a la computadora o por medio de sesiones remotas en una red de comunicaciones. No importa el número de procesadores en la máquina ni el número de procesos que cada usuario puede ejecutar simultáneamente.

Monotareas

Los sistemas monotarea son aquellos que sólo permiten una tarea a la vez por usuario. Puede darse el caso de un sistema multiusuario y monotarea, en el cual se admiten varios usuarios al mismo tiempo pero cada uno de ellos puede estar haciendo sólo una tarea a la vez.

Multitareas

Un sistema operativo multitarea es aquél que le permite al usuario estar realizando varias labores al mismo tiempo. Por ejemplo, puede estar editando el código fuente de un programa durante su depuración mientras compila otro programa, a la vez que está recibiendo correo electrónico en un proceso en *background*. Es común encontrar en ellos una interfaz gráfica orientada al uso de menús y el ratón, lo cual permite un rápido intercambio entre las tareas para el usuario, mejorando su productividad.

Uniproceso

Un sistema operativo uniproceso es aquél que es capaz de manejar solamente un procesador de la computadora, de manera que si la computadora

tuviese más de uno le sería inútil. El ejemplo más típico de este tipo de sistemas es el DOS y MacOS.

Multiproceso

Un sistema operativo multiproceso se refiere al número de procesadores del sistema, que es más de uno y éste es capaz de usarlos todos para distribuir su carga de trabajo. Generalmente estos sistemas trabajan de dos formas: simétrica o asimétricamente. Cuando se trabaja de manera asimétrica, el sistema operativo selecciona a uno de los procesadores el cual jugará el papel de procesador maestro y servirá como pivote para distribuir la carga a los demás procesadores, que reciben el nombre de esclavos. Cuando se trabaja de manera simétrica, los procesos o partes de ellos (*threads*) son enviados indistintamente a cualesquiera de los procesadores disponibles, teniendo, teóricamente, una mejor distribución y equilibrio en la carga de trabajo bajo este esquema.

Se dice que un *thread* es la parte activa en memoria y corriendo de un proceso, lo cual puede consistir de una área de memoria, un conjunto de registros con valores específicos, la pila y otros valores de contexto. Un aspecto importante a considerar en estos sistemas es la forma de crear aplicaciones para aprovechar los varios procesadores. Existen aplicaciones que fueron hechas para correr en sistemas monoproceso que no toman ninguna ventaja a menos que el sistema operativo o el compilador detecte secciones de código paralelizable, los cuales son ejecutados al mismo tiempo en procesadores diferentes. Por otro lado, el programador puede modificar sus algoritmos y aprovechar por sí mismo esta facilidad, pero esta última opción las más de las veces es costosa en horas hombre y muy tediosa, obligando al programador a ocupar tanto o más tiempo a la paralelización que a elaborar el algoritmo inicial.

1.4.3 Sistemas Operativos por la Forma de Ofrecer sus Servicios

Esta clasificación también se refiere a una visión externa, que en este caso se refiere a la del usuario, el cómo accede los servicios. Bajo esta clasificación se pueden detectar dos tipos principales: sistemas operativos de red y sistemas operativos distribuidos.

Sistemas Operativos de Red

Los sistemas operativos de red se definen como aquellos que tiene la capacidad de interactuar con sistemas operativos en otras computadoras, a través de un medio de transmisión, con el objeto de intercambiar información, transferir archivos, ejecutar comandos remotos y un sin fin de otras actividades. El punto crucial de estos sistemas es que el usuario debe saber la sintaxis de un conjunto de comandos o llamadas al sistema para ejecutar estas operaciones, además de la ubicación de los recursos que desee acceder. Lo importante es hacer ver que el usuario puede acceder y compartir muchos recursos.

Un ejemplo típico es una red con estaciones de trabajo conectadas mediante una LAN. En este modelo, cada usuario tiene una estación de trabajo para su uso exclusivo, con su propio sistema operativo y usualmente todos los trabajos se ejecutan en forma local. Eventualmente, un usuario pudiera requerir ejecutar programas que estén en una estación remota. Para ello el usuario debe realizar una operación de conexión remota mediante algún comando, convirtiendo su estación de trabajo en un terminal de la máquina remota. La transferencia de archivos entre máquinas se realiza en forma explícita.

Sistemas Operativos Distribuidos

Integrar recursos (impresoras, unidades de respaldo, memoria, procesos, unidades centrales de proceso) en una sola máquina virtual, que el usuario accede en forma transparente. Es decir, ahora el usuario ya no necesita saber la ubicación de los recursos, sino que los conoce por nombre y simplemente los usa como si todos ellos fuesen locales a su lugar de trabajo habitual.

Todo lo anterior es el marco teórico de lo que se desearía tener como sistema operativo distribuido, pero en la realidad no se ha conseguido crear uno del todo, por la complejidad que suponen: distribuir los procesos en las varias unidades de procesamiento, reintegrar sub-resultados, resolver problemas de concurrencia y paralelismo, recuperarse de fallas de algunos recursos distribuidos y consolidar la protección y seguridad entre los diferentes componentes del sistema y los usuarios.

1.5 TENDENCIAS DE LOS SISTEMAS OPERATIVOS

Para los 90's el paradigma de la programación orientada a objetos cobra auge, así como el manejo de objetos desde los sistemas operativos. Las aplicaciones intentan crearse para ser ejecutadas en una plataforma específica y poder ver sus resultados en la pantalla o monitor de otra. Los niveles de interacción se van haciendo cada vez más profundos.

También comenzará la era de la computación distribuida, en la cual los cómputos se dividirán en subcómputos que podrán ejecutarse en otros procesadores, en computadoras de procesadores múltiples y en redes de computadoras. Las aplicaciones aprovecharán los ciclos del procesador que se solían desperdiciar en las redes de computadoras personales y de estaciones de trabajo de los años ochenta; los subcómputos se distribuirán de modo que puedan aprovechar al máximo las computadoras de propósito especial en toda una red.

Las redes tendrán una configuración dinámica; es decir, seguirán operando aunque se añadan o eliminen dispositivos y software. Cada vez que se integre un nuevo despachador, éste se dará a conocer en la red mediante un procedimiento de registro, en el cual el despachador informa a la red de su capacidad, política de facturación, accesibilidad, etc., para lograr una flexibilidad, se podrán en contacto

con ciertas entidades de la red denominadas localizadores; éstos sabrán cuáles despachadores están disponibles, dónde están ubicados y cómo se obtiene acceso a ellos.

Esta clase de conectividad se facilitará gracias a los estándares y protocolos de los sistemas abiertos, que en la actualidad están siendo preparados internacionalmente por diferentes grupos, buscando alcanzar un ambiente de normas de aceptación internacional para la computación y las comunicaciones.

Los futuros sistemas de cómputo operarán con paralelismos a gran escala: tendrán tal número de procesadores que se podrán efectuar en paralelo todos los cómputos que se presten a ello.

La existencia de nuevas arquitecturas, nuevos sistemas y plataformas más potentes a la vez que más económicas hace que muchas organizaciones se planteen el traslado de sus aplicaciones corporativas que residen en servidores centrales o *mainframes* hacia nuevas plataformas.

Sin embargo, debido a los rápidos cambios de las tecnologías, es necesario garantizar de cierta forma la inversión que se realiza en el proyecto de rediseño de la aplicación. La estrategia que se utiliza incluye el concepto de *middleware*.

El ***middleware*** es un módulo intermedio que actúa como conductor entre sistemas permitiendo a cualquier usuario de sistemas de información comunicarse con varias fuentes de información que se encuentran conectadas por una red.

Desde un punto de vista amplio una solución basada en productos *middleware* debe permitir conectar entre sí a una variedad de productos procedentes de diferentes proveedores. De esta forma se puede separar la estrategia de sistemas de información de soluciones propietarias de un solo proveedor.

Las categorías del *middleware* podríamos definirlas de la siguiente forma:

- Monitores de proceso de transacciones distribuidos (*DTPM's Distributed Transaction Processing Monitors*). Herederos de la tecnología mainframe, son ampliamente demandados para intercomunicar distintos sistemas en distintos entornos.
- Llamadas a procedimientos remotos (*RPC's Remote procedure Call*) Diseñado como servicios síncronos para permitir gestión remota de redes.
- Middleware orientado a mensajes (*MOM Messaging Oriented Middleware*) Diseñado para servicios de mensajes con tecnología asíncrona.
- *(ORB Objects Request Broker)* Middleware para tecnologías orientadas a objetos. Objetos piden servicios de objetos que se encuentran en la red.

El estándar más conocido de esta tecnología es CORBA *Common Object Request Broker Arquitecture.*

- Middleware de acceso a Bases de Datos *(Data Base Access Middleware).* Para acceso estándar a bases de datos. Permite desarrollar sistemas independizándolo de la base de datos que lo soporte. En la actualidad representa el 50% del mercado del *middleware.*

Las ventajas que ofrece son:

- Simplifica el proceso de desarrollo de aplicaciones al independizar los entornos propietarios.
- Permite la interconectividad de los sistemas de información del organismo.
- Proporciona mayor control del negocio al poder contar con información procedente de distintas plataformas sobre el mismo soporte.
- Facilita el desarrollo de sistemas complejos con diferentes tecnologías y arquitecturas.

Dentro de los inconvenientes más importantes destacan la mayor carga de máquina necesaria para que puedan funcionar.

2. LOS SISTEMAS DISTRIBUIDOS

2.1 INTRODUCCIÓN

Los avances tecnológicos en las redes de área local y la creación de microprocesadores de 32 y 64 bits lograron que computadoras más o menos baratas tuvieran el suficiente poder en forma autónoma para desafiar en cierto grado a los mainframes, y a la vez se dio la posibilidad de intercomunicarlas, sugiriendo la oportunidad de partir procesos muy pesados en cálculo en unidades más pequeñas y distribuirlas en los varios microprocesadores para luego reunir los sub-resultados, creando así una máquina virtual en la red que exceda en poder a un mainframe.

El resultado neto de estos avances tecnológicos, es que hoy en día, no sólo es posible, sino fácil, reunir sistemas de cómputo compuestos por un gran número de CPU, conectados mediante una red de alta velocidad. Estos reciben el nombre genérico de sistemas distribuidos.

2.2 DEFINICIÓN

Un sistema distribuido es una colección de computadoras independientes que aparecen ante los usuarios del sistema como una única computadora.

2.3 CONCEPTOS BÁSICOS

Esta definición tiene dos aspectos importantes. El primero se refiere al hardware: las máquinas son autónomas. El segundo se refiere al software: los usuarios piensan que el sistema es como una única computadora. Ambos son esenciales. Como ejemplos de sistemas distribuidos podemos mencionar, el control de los cajeros automáticos en diferentes estados de la república.

2.3.1 Aspectos de hardware

Aunque los sistemas distribuidos constan de varios CPU, existen diversas formas de organizar el hardware; en particular, en la forma de interconectarlos y comunicarse entre sí.

Dado que un Sistema Distribuido es una colección de (usualmente geográficamente) hardware distribuido, el que da la impresión de ser un ambiente computacional unificado por algún software "inteligente", es necesario mirar configuraciones hardware y software diferentes.

A través de los años, se han propuesto varios esquemas de clasificación para los Sistemas Computacionales que poseen varios CPU. La más citada es la que realizó Flynn (1972). Eligió dos características consideradas por él como esenciales: el número de flujo de instrucciones y el número de flujo de datos. La clasificación entonces realizada es la siguiente:

- SISD - **Single Instruction Single Data**: se refiere a la ejecución de una instrucción en una unidad de datos para que realice la instrucción en paralelo. Las computadoras tradicionales de un procesador caen en esta categoría.

- SIMD - **Single Instruction Multiple Data:** se refiere a la ejecución de una instrucción en varias unidades de datos para que realicen la instrucción en paralelo. Estas máquinas son útiles cuando se requiere realizar el mismo cálculo sobre varios conjuntos de datos diferentes. Por ejemplo, la suma de los elementos de varios vectores independientes. Existen algunos supercomputadoras que caen en esta categoría.

- MISD - **Multiple Instruction Single Data:** se refiera a la ejecución en paralelo de múltiples instrucciones sobre un mismo flujo de datos. No existen computadoras conocidas que se clasifiquen dentro de esta categoría.

- MIMD - **Multiple Instruction Multiple Data**: se refiere a un grupo de computadoras independientes, cada uno con su propio controlador del programa y datos.

Basándonos en la clasificación anterior, se dice que todos los sistemas distribuidos pertenecen a ésta última categoría (MIMD), por lo que realiza una subdivisión de ellas en dos grupos: aquellos que tienen memoria compartida, a los que llama *multiprocesadores* y, aquellos que no tienen memoria compartida, los que denomina *multicomputadoras*. La diferencia fundamental radica en que en un multiprocesador se tiene un espacio de direcciones virtuales compartido por todos los CPU, de manera que si se realiza una modificación en una localidad, ésta es vista desde los demás procesadores, es decir, la memoria es compartida.

En contraste, en una multicomputadora, cada procesador tiene su propia memoria, de manera que las modificaciones en una localización sólo es vista por aquel procesador que tiene acceso a esa memoria en particular.

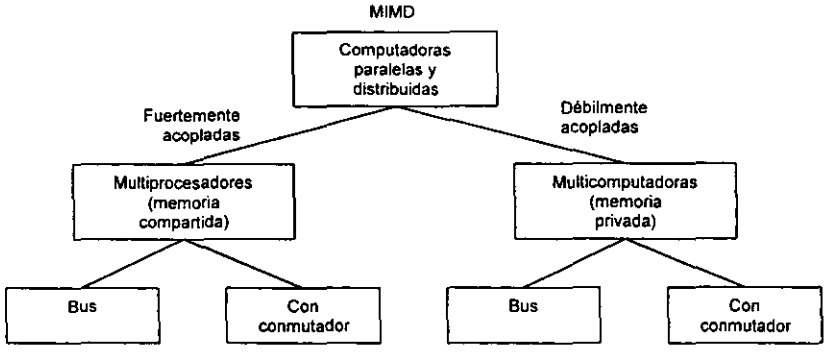

Figura 6. Taxonomía de los sistemas de cómputo paralelos y distribuidos

Cada una de estas categorías se puede subdividir, dependiendo de la arquitectura de interconexión, en bus y con conmutador. En la primera, existe un medio que conecta todas las máquinas (Ejemplo: televisión por cable). En la segunda, existen, además, medios que conectan unas con otras y pueden utilizar patrones diferentes de cableado. Los mensajes se mueven a través de los cables y la decisión de conmutación se toma en cada etapa (Ejemplo: sistema mundial de teléfonos públicos).

Por otra parte, es posible distinguir entre sistemas *fuertemente acoplados* y *débilmente acoplados*. En los sistemas fuertemente acoplados, el retraso que se experimenta al enviar un mensaje desde una máquina a la otra es bajo, con una alta tasa de transmisión de datos; en cambio, en los sistemas débilmente acoplados el retraso suele ser mayor y poseen una tasa de transmisión más baja.

Los sistemas fuertemente acoplados tienden a utilizarse más como sistemas paralelos (para trabajar con un problema) y los débilmente acoplados tienden a utilizarse más como sistemas distribuidos (para trabajar con varios problemas no relacionados entre sí), aunque esto no siempre es cierto.

En general, los multiprocesadores tienden a estar más fuertemente acoplados que las multicomputadoras, puesto que pueden intercambiar datos a velocidad de sus memorias, pero algunas multicomputadoras basadas en fibras ópticas pueden funcionar también con velocidad de memoria.

Multiprocesamiento con base en buses

Los multiprocesadores con base en buses constan de cierta cantidad de CPU, conectados a un bus común, junto con un módulo de memoria. Una configuración sencilla consta de un plano de base (*backplane*) de alta velocidad o tarjeta madre, en la cual se pueden insertar las tarjetas de memoria y el CPU.

Para leer una palabra de memoria, un CPU coloca la dirección de la palabra deseada en la líneas de direcciones del bus y coloca una señal en las líneas de control adecuadas para indicar que desea leer. La memoria responde y coloca el valor de la palabra en las líneas de datos para permitir la lectura de ésta por parte del CPU solicitante. La escritura funciona de manera similar.

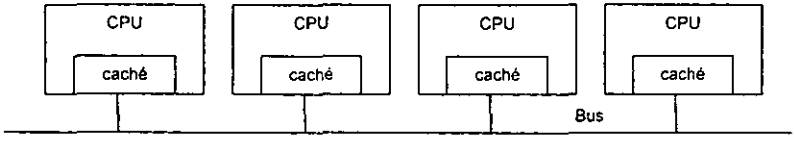

Figura 7. Multiprocesador con base en bus

El problema es que el bus estará por lo general sobrecargado y el rendimiento disminuirá en forma drástica. La solución es añadir una memoria caché de alta velocidad entre el CPU y el bus. Todas las solicitudes de la memoria pasan a través del caché. Si la palabra solicitada se encuentra en el caché, éste responde al CPU y no se hace solicitud alguna al bus. Si el caché es lo bastante grande, la probabilidad de éxito (la tasa de encuentros) será alta y la cantidad de tráfico en el bus por cada CPU disminuirá en forma drástica, lo que permite un número mayor de CPU en el sistema.

Multiprocesadores con conmutador

Para construir un multiprocesador con más de 64 procesadores, es necesario un método distinto para conectar cada CPU con la memoria. Una posibilidad es dividir la memoria en módulos y conectarlos a las CPU con un conmutador de cruceta, como se muestra en la figura 8. En cada intersección está un delgado conmutador de punto de cruce electrónico que el hardware puede abrir y cerrar. Cuando un CPU desea tener acceso a una memoria particular, el conmutador del punto de cruce que los conecta se cierra de manera momentánea, para permitir dicho acceso. La virtud del conmutador de cruceta es que muchos CPU pueden tener acceso a la memoria al mismo tiempo, aunque si dos CPU intentan tener acceso a la misma memoria en forma simultánea, uno de ellos deberá esperar.

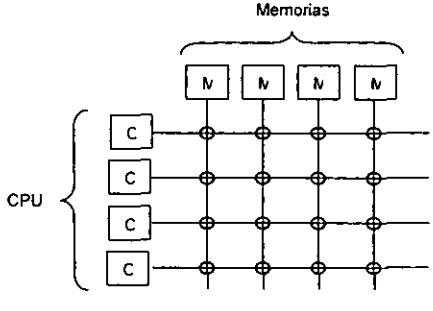

Figura 8. Conmutador de cruceta

La desventaja del conmutador de cruceta es que con n CPU y n memoria, se necesitan n^2 conmutadores en los puntos de cruce. Si n es grande, este número puede ser prohibido. Otro problema es el retraso.

Se ha buscado y encontrado otras redes de conmutación que necesiten menos conmutadores. La red omega es un ejemplo; esta red contiene conmutadores 2x2, cada uno de los cuales tiene dos entradas en cualquiera de las salidas. En el caso general, con n CPU y n memorias, la red omega necesita $log_2 n$ etapas de conmutación, cada una de las cuales tiene $n/2$ conmutadores, para un total de $(n log_2 n)/2$ conmutadores. Aunque es menor, sigue siendo considerable.

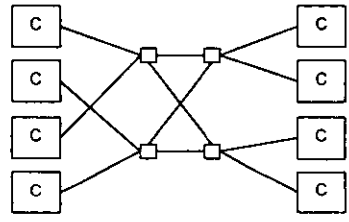

Figura 9. Red omega de comunicación

Al intentar reducir el costo por los múltiples conmutadores se ha hecho uso de los sistemas jerárquicos, donde cada CPU tiene asociada cierta memoria. Este diseño da lugar a la llamada máquina NUMA *(Non Uniform Memory Access)*. Estas máquinas tienen un mejor promedio de acceso pero la complicación es la colocación de los programas y datos convirtiéndose en un factor crítico.

En resumen, los multiprocesadores basados en buses, incluso con cachés monitores, quedan limitados a lo más a 64 CPU por la capacidad del bus. Para rebasar estos límites es necesaria una red con conmutador, como uno de cruceta, una red omega o algo similar. Los grandes conmutadores de cruceta y las grandes redes omega son muy caros y lentos. Las máquinas NUMA necesitan complejos algoritmos para la buena colocación del software. La conclusión es clara: la construcción de un multiprocesador grande, fuertemente acoplado y con memoria compartida es difícil y cara.

Multicomputadoras con base en buses

Cada CPU tiene conexión directa con su propia memoria local. El único problema restante es la forma en que los CPU se comunicarán entre sí. Es claro que aquí también se necesita cierto esquema de interconexión, pero como sólo es para la comunicación entre un CPU y otro, el volumen de tráfico será de varios órdenes menores en relación con el uso de una red de interconexión para el tráfico CPU-memoria.

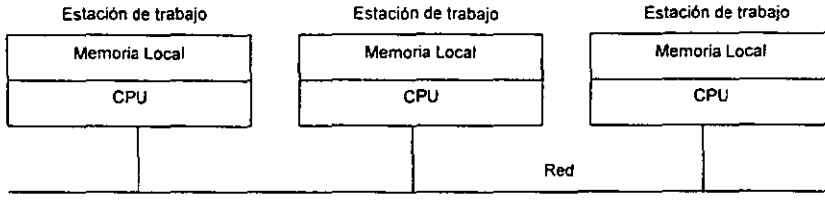

Figura 10. Multicomputadora que consta de estaciones de trabajo en una LAN

Multicomputadoras con conmutador

La última categoría es la de multicomputadoras con conmutador. Se han propuesto y construido varias redes de interconexión, pero todas tiene la propiedad de que cada CPU tiene acceso directo y exclusivo a su propia memoria particular. Dos topologías populares son, la retícula y el hipercubo.

Las retículas son fáciles de comprender y se basan en las tarjetas de circuitos impresos. Se adecúan mejor a problemas con naturaleza bidimensional inherente, como la teoría de gráficas o la visión.

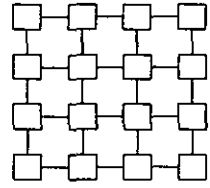

Figura 11. Retícula

Un hipercubo es un cubo n-dimensional. El hipercubo de la figura 12 es de dimensión 4. Se puede pensar como dos cubos ordinarios, cada uno de los cuales cuenta con 8 vértices y 12 aristas. Cada vértice es un CPU. Cada arista es una conexión entre dos CPU. Se conectan los vértices correspondientes de cada uno de los cubos.

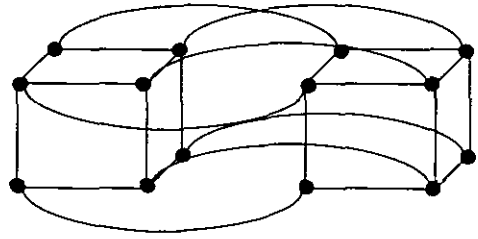

Figura 12. Hipercubo

Así, la complejidad del cableado aumenta en proporción logarítmica con el tamaño. Puesto que sólo se conectan los vecinos más cercanos, muchos mensajes deben realizar varios saltos antes de llegar a su destino. Sin embargo, la trayectoria de mayor longitud también crece en forma logarítmica junto con el tamaño, en contraste con la retícula, donde ésta crece conforma a la raíz cuadrada del número de CPU.

2.3.2 Redes de computadoras

Una red de computadoras está conectada tanto por hardware como por software. El hardware incluye tanto las tarjetas de interfaz de red como los cables que las unen, y el software incluye los controladores (programas que se utilizan para administrar los dispositivos y el sistema operativo de red).

Componentes de Redes

A continuación se listan los componentes de una red:

- **Servidor.** Ejecuta el sistema operativo de red y ofrece los servicios de red a las estaciones de trabajo. Existen dos tipos de servidores, los dedicados y los no dedicados. Los servidores no dedicados, aparte de compartir sus recursos pueden ser utilizados como estaciones de trabajo; mientras que un servidor dedicado no puede ejecutar ningún otro trabajo aparte del requerido. El servidor está para compartir sus recursos con los nodos de la red.

- **Estaciones de trabajo.** Es una computadora capaz de aprovechar los recursos de otras computadoras (servidores). Una estación de trabajo no comparte sus propios recursos con otras computadoras y, los demás nodos no pueden usar ningún recuso de ella.

- **Tarjetas o Placas de Interfaz de Red.** Toda computadora que se conecta a una red necesita de una tarjeta de interfaz de red que soporte un esquema de red específico. Hay tarjetas de interfaz de red disponibles de diversos fabricantes. Se pueden elegir entre distintos tipos, según se desee configurar o cablear la red. Los tres tipos más usuales son ArcNet, Ethernet y Token Ring. Las diferencias entre estos distintos tipos de red se encuentran en el método y velocidad de comunicación, así como el precio.

- **Sistema de Cableado.** El sistema de la red está constituido por el cable utilizado para conectar entre sí, el servidor y las estaciones de trabajo. El cable coaxial fue uno de los primeros que se usaron, pero el par trenzado ha ido ganando popularidad. El cable de fibra óptica se utiliza cuando es importante la velocidad.

- **Recursos periféricos y compartidos.** Entre los recursos compartidos se incluyen los dispositivos de almacenamiento ligados al servidor, las unidades de discos ópticos, las impresoras y el resto de equipos que puedan ser utilizados por cualquiera en la red.

Arquitecturas de Redes

La siguiente figura ilustra las topologías más habituales de los circuitos físicos. Estas topologías son:

- Estrella
- Parcialmente conexa
- Totalmente conexa
- Anillo
- Red jerárquica o con estructura de árbol

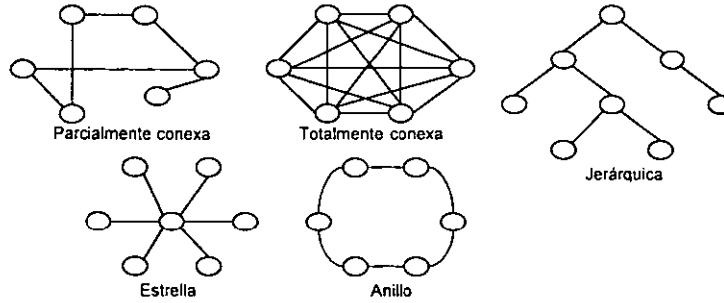

Figura 13. Topologías de red

En general, en un sistema con un número fijo de máquinas, el incremento en el número de circuitos físicos de comunicación produce un aumento en la disponibilidad y velocidad, además de una disminución de los retardos. Desgraciadamente también incrementa el costo del sistema.

La configuración estrella encamina todas las comunicaciones a través de un único punto central, que puede ser un nodo o un conmutador. La estrella tiene un retardo fijo de comunicación entre máquinas, de dos saltos, pero tiene como inconveniente un punto de fallo único.

Los sistemas totalmente conexos son rápidos y fiables pero costosos ya que el número de enlaces crece como el cuadrado del número de máquinas. Las redes parcialmente conexas y en malla tienen enlaces directos entre algunos de los nodos pero no de todos. Rebajando el número de conexiones físicas se reducen los costos. Como hay máquinas que no están directamente conectadas, aumenta el peligro de particionamiento de red, en donde el fallo de un solo enlace puede romper la subred de comunicación en dos o más subconjuntos disjuntos incapaces de comunicarse entre sí.

Las redes anillo de un solo enlace tienen un costo bajo, pero retardos variables y potencialmente largos, especialmente cuando una máquina desea comunicarse con su vecino inmediato en la dirección opuesta a la del flujo del anillo. Los anillos son generalmente sensibles a fallos de enlace, especialmente en implementaciones simples, en donde cada nodo debe actuar como repetidor activo para todo el tráfico.

Las conexiones jerárquicas pueden ser adecuadas para ciertos tipos de organizaciones y sistemas, tales como el control de procesos en donde la comunicación fluye de forma natural de una manera jerárquica. Esta configuración es pobre en sistemas con frecuentes interacciones entre nodos de un mismo nivel, ya que deben ser encaminadas arriba y abajo en la jerarquía.

Independientemente de la topología de las conexiones, el software de red suele permitir que cualquier máquina se comunique lógicamente directamente con cualquier otra. Como resultado, los procesadores que necesitan entregar mensajes a procesadores con los cuales no están directamente conectados, deben pasar a través de procesadores intermedios. Esta posibilidad requiere que la subred de comunicación proporcione encaminamiento de mensajes entre procesadores que no están directamente conectados. Las estrategias de encaminamiento más frecuentemente utilizadas incluyen:

- Encaminamiento fijo
- Circuito virtual
- Encaminamiento dinámico

En el encaminamiento fijo, las rutas entre todos los pares específicos emisor-receptor se terminan de una vez, antes o durante la inicialización del sistema, permanecen fijas desde entonces. Este esquema es obviamente sencillo de implementar, pero es inflexible e incapaz de adaptarse a variaciones en la carga de comunicaciones y fallo de los procesadores.

El método del circuito virtual establece y fija un camino entre nodos extremos válidos durante una sesión de comunicación individual. El camino puede ser compartido con otros procesos que requieran la misma conexión física, pero cuando tal compartición se produce es transparente a los procesos. La comunicación sobre circuitos virtuales es como si se asignase un enlace físico privado a un par de procesos en comunicación.

El encaminamiento dinámico establece caminos de extremo a extremo sobre la marcha, típicamente mensaje a mensaje. Su flexibilidad permite que el procesador originario proporciona únicamente una especificación de encaminamiento parcial, que es completada por los procesadores intermedios visitados por el mensaje en su viaje hacia el destino. Con encaminamiento dinámico, mensajes consecutivos entre dos máquinas pueden tomar rutas diferentes en respuesta a la carga y disponibilidad individuales de canales y

procesadores. Este esquema impone una mayor carga computacional sobre los procesadores y debe ser capaz de solucionar el problema de la entrega de mensajes desordenados, que puede confundir a algunos algoritmos distribuidos.

Otro aspecto del diseño de la subred de comunicación es la estrategia de conexión. El problema aquí es, durante cuánto tiempo debería estar dedicado un enlace de comunicación a un par determinado emisor-receptor. Las estrategias más comunes a este problema incluyen:

- Conmutación de circuitos
- Conmutación de mensajes
- Conmutación de paquetes

En conmutación de circuitos se asemeja a un sistema telefónico. Funciona estableciendo y manteniendo un circuito físico dedicado de extremo a extremo entre las máquinas en comunicación durante la sesión de comunicación entera.

En conmutación de mensajes se establece un enlace físico temporal entre el emisor y el receptor durante la transferencia de un solo mensaje. Los enlaces físicos son asignados a los usuarios durante breves períodos de tiempo y se conmutan rápidamente con objeto de incrementar la utilización de la línea física.

La conmutación de paquetes intenta incrementar aún más la utilización de las líneas, la productividad del sistema y algunos aspectos de la gestión de búferes de los procesadores dividiendo los mensajes largos trozos de tamaño fijo denominados paquetes. Los diferentes paquetes pueden ser entonces enviados a través de rutas diferentes y reunidos en el destino para su entrega al receptor.

En general, la elección de uno de estos esquemas incluye consideraciones tales como el tiempo de preparación, recargo por mensaje, los retardos de comunicación de mensaje, la productividad del sistema y la utilización de los circuitos de comunicación. La conmutación de circuitos requiere un tiempo de preparación pero tiene un menor recargo por mensaje y retardos más breves. La conmutación de mensajes y de paquetes requieren un menor tiempo de preparación y proporciona potencialmente mayores tiempos de utilización, Como aspecto negativo tienden a tener mayores recargos ya que deben incluir direcciones y posiblemente información de secuenciamiento de cada mensaje o paquete, también tienden a complicar el software de los procesadores añadiendo requisitos de servicios tales como asignación de búferes, formación y reensamblaje de paquetes y procedimientos de recuperación y gestión de errores más complejos.

Tipos de Redes

Dependiendo de la distancia física que alcanzan las redes de comunicación, éstas se clasifican en general como:

- Redes de área extensa (WAN, Wide Area Networks)
- Redes de área local (LAN, Local Area Networks)

Aunque no se ha definido ninguna distancia delimitadora específica, es razonable pensar que una red de área local abarca los confines de un único edificio o de un campus pequeño. Las redes de área extensa, por otro lado, pueden conectar máquinas que están separadas por muchos kilómetros, incluso incluyendo distancias intercontinentales.

Por razones principalmente tecnológicas, las WAN tienden a tener anchos de banda comparativamente bajos y retardos de comunicación elevados. Por otro lado, las LAN suelen estar caracterizadas por elevados anchos de banda y bajos retardos de comunicación. Estas diferencias fundamentales tienen un impacto significativo sobre las opciones de diseño y la selección de algoritmos distribuidos cuyas características pueden hacerles más adecuados para un tipo de red que para el otro.

Redes de área extensa

La figura 13 ilustra una arquitectura típica de una red de área extensa. Como se indica, la propia subred de comunicación consta de una serie de procesadores de comunicación conectados mediante líneas de comunicación físicas. Los procesadores de comunicación dedicados actúan como elementos de conmutación entre dos o más líneas de comunicación (también llamadas líneas de transmisión, circuitos o canales). Los procesadores suelen denominarse procesadores de interfaz de mensajes o PIM en abreviatura.

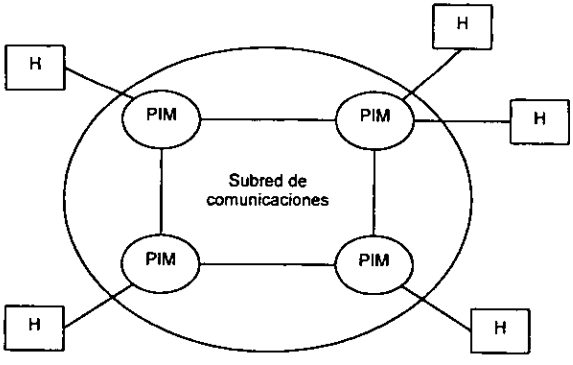

Figura 14. Arquitectura de una red de área extensa

Un procesador que desee comunicarse con otro, presenta típicamente su petición al PIM designado. Generalmente cada máquina aprovecha los servicios de un PIM específico, aunque un mismo PIM puede atender a varias máquinas. Los canales de comunicación PIM a PIM pueden ser uno de dos tipos:

- Punto a punto
- Difusión

Los enlaces punto a punto son líneas físicas dedicadas utilizadas para conectar un par específico de PIM. El número efectivo y la topología de estos enlaces se determinan tomando en cuenta factores tales como el costo, los retardos de comunicación y la fiabilidad.

Redes de área local

Las redes de área local (LAN) se caracterizan por enlaces de comunicación con elevado ancho de banda y bajo retardo. Suelen apoyarse en un medio de comunicación de acceso común que transporta el tráfico de mensajes a velocidades relativamente altas. Las velocidades de comunicación en LAN pueden ser desde el orden de varios Mbps hasta el orden de Gbps con cableado especial o con fibras ópticas.

El propio medio de comunicación LAN es pasivo en el sentido de que no proporciona potencia de procesamiento, encaminamiento, ni las funciones de almacenamiento que se encuentran en las subredes de comunicación de tipo almacenar y reexpedir. En las LAN, las funciones de comunicación y procesamiento las proporcionan los nodos directamente mediante coprocesadores de comunicación dedicados.

El medio de comunicación pasivo y la ausencia de procesamiento en ruta dan lugar a una potencialmente alta fiabilidad de los subsistemas con comunicación por redes de área local. La topología de red es generalmente un bus o un anillo. En sistemas de propósito especial y en algunos de los primeros diseños puede encontrarse una topología en estrella.

En sistemas basados en anillos, los mensajes circulan alrededor del anillo. Los mensajes son entonces reconocidos y copiados por sus destinatarios. Dependiendo del esquema de reconocimiento utilizado, puede ser el nodo destino o el nodo fuente el encargado de retirar del anillo el mensaje consumido. La topología en anillo parece ser muy vulnerable a disfunciones de nodos y enlaces, ya que cualquier fallo simple rompe aparentemente el anillo y por tanto paraliza el tráfico. Las implementaciones comerciales en anillo evitan este problema utilizando relés para conectar los nodos al anillo. El relé está diseñado de tal modo que debe ser activamente alimentado con el fin de permitir que el tráfico fluya a través del nodo asociado. Si se detectan fallos o el nodo no está enchufado, el relé se cierra y puentea el nodo fallido. Ésta y otras técnicas similares proporcionan continuidad en el anillo en presencia de malfunciones de nodo, posibilitando la comunicación sin interrupciones entre los nodos no afectados.

Otra topología habitual en las redes de área local es el bus. En una topología de bus típica, un conductor pasivo comúnmente accesible transporta el tráfico de mensajes, Todos los nodos tiene una toma de conexión al bus,

escuchan todo el tráfico y extraen los mensajes que le son dirigidos. Las propias tomas son generalmente pasivas y rompen físicamente el bus. Como resultado, las LAN orientadas a bus son fiables y capaces de sostener comunicaciones entre nodos sanos en presencia de múltiples fallos de nodos.

Conceptos importantes

Se mencionarán algunos conceptos teóricos importantes de red que serán necesarios para la configuración de ésta, vía software.

Dirección IP

Es la dirección única de cada máquina, formada por cuatro números en la base que se desee separados entre éstos por un punto. Si está configurando únicamente el modo "*loopback*", la dirección IP será la 127.0.0.1.

Toda red debe trabajar con un determinado protocolo de transmisión de datos, que indica cómo se efectúa la transferencia de información entre las computadoras de la red. El protocolo utilizado por Internet se llama TCP/IP (*Transmisión Control Protocol / Internet Protocol*). Si la conexión a Internet es vía telefónica mediante un módem, el protocolo a usar son unas variantes especiales de TCP/IP denominadas SLIP o PPP. Tanto *SLIP (Serial Line Interface Protocol)* como *PPP (Point to Point Protocol)* son versiones de *TCP/IP* diseñadas para establecer comunicación TCP/IP a través del puerto serie.

Una dirección IP no identifica por sí misma a una computadora, sino más bien la conexión de una computadora con la red. Las direcciones IP están compuestas por 32 bits, que se componen de dos partes diferenciables.

Dirección IP = Dirección de red + Dirección de máquina

Una primera parte, identifica la dirección de la red (NETID). Esta parte es asignada por el Network Information Center (NIC), de la Red de Datos de Defensa (RDS), gobernada por Network Solutions en Cantilly, Virginia. Evidentemente, si la red no va a conectarse a otras redes no es necesario solicitar a este organismo una dirección de red concreta sino que el administrador de la red puede asignar una cualquiera. El número de bits que ocupa esta parte depende del tamaño de la red y puede estar formado de 8, 16 o 24 bits.

La segunda parte se trata de la identificación de la máquina dentro de un segmento. Mientras que las direcciones de red o NETID son asignadas por una organización concreta para evitar la duplicidad de direcciones de red, las direcciones de máquinas son asignadas por el administrador.

Una de las características del formato de las direcciones IP es que permite establecer clases de redes.

Cada clase puede direccionar más o menos redes, con la consecuencia de un incremento o decremento del número de máquinas que se pueden asignar a ésta.

Clase	Tamaño de la dirección de red (en octetos)	Primer número	Número de direcciones locales
A	1	0-127	16 777 216
B	2	128-191	65 536
C	3	192-223	256

En los inicios de la Internet, a las organizaciones con redes muy grandes, se les concedía rangos de direcciones IP de clase (A). La parte de red de una dirección de clase (A) tiene una longitud de un octeto. Los tres octetos restantes de una dirección IP de clase (A) pertenecen a la parte local y se usan para asignar números a los nodos.

Existen muy pocas direcciones de clase (A) y la mayoría de las organizaciones de gran tamaño han tenido que conformarse con un bloque de direcciones de clase (B) de tamaño medio. La parte de red de una dirección de clase (B) es de dos octetos. Los dos octetos restantes de una dirección de clase (B) pertenecen a la parte local y se usan para asignar números a los nodos.

Las organizaciones pequeñas reciben una o más direcciones de clase (C). La parte de red de una dirección de clase (C) es de tres octetos. De esta forma sólo queda un octeto para la parte local que se usan para asignar números a los nodos.

Además de las clases A, B y C, existen dos formatos especiales de direcciones, la clase D y la clase E. Las direcciones de clase D se usan para Multienvío de IP. El Multienvío permite distribuir un mismo mensaje a un grupo de computadoras dispersas por una red. Las direcciones de clase E se han reservado para uso experimental. Las direcciones de clase D empiezan con un número entre 224 y 239, mientras que las direcciones de clase E empiezan con un número entre 240 y 255.

Máscara de red ("netmask")

La máscara de red es un patrón de bits, que al ser superpuesto a una dirección de red, dirá en qué sub-red se encuentra esa dirección.

Esto es muy importante para el rutado y, si nota que puede comunicarse con gente de redes externas pero no con gente de su misma red, el error estará en la mala colocación de la máscara.

Todo esto debe aplicarse también a la configuración "*loopback*". Dado que la dirección "*loopback*" es siempre la 127.0.0.1, la máscara será 255.0.0.0.

Dirección de red

Es el resultado de la operación lógica AND, entre la dirección IP y la máscara. Por ejemplo, si la dirección IP es la 128.253.154.32 y la máscara es 255.255.255.0, la dirección de red será la 128.253.154.0. Si utiliza sólo la configuración en "*loopback*", la dirección de red no existe.

Dirección de "broadcast"

Se utiliza para lanzar paquetes que deben recibir todas las máquinas de la sub-red. La dirección de "*broadcast*" es el resultado de la operación lógica OR entre la dirección de red y 0.0.0.255. La dirección "*broadcast*" tampoco tiene utilidad en una configuración en "*loopback*".

Por ejemplo, si el número IP es el 128.253.154.32, y la máscara es la 255.255.255.0, la dirección de "broadcast" seria la 128.253.154.255.

Dirección de pasarela

Se trata de la dirección de la máquina que va a ser la pasarela a otras máquinas que no estén en su misma sub-red.

Muchas veces es una dirección IP terminada en ".1". Por ejemplo, si la dirección IP es la 128.253.154.32, la de la pasarela podría ser la 128.253.154.1.

En ocasiones puede tener varias pasarelas. Una pasarela o *gateway* es simplemente una máquina que se encuentra a la vez en dos sub-redes (tiene una dirección IP por cada una), y reparte los paquetes entre ellas. En muchas sub-redes existe una sola pasarela para comunicarse con las redes externas, pero en otras hay varias, una para cada sub-red adicional.

Si la red está aislada de otras, o la máquina se encuentra en configuración "*loopback*", no necesitará dirección de pasarela.

2.3.3 Clusters

Definición

Los Clusters o cúmulos son un conjunto de máquinas de arquitecturas homogéneas o heterogéneas conectadas en red bajo cualquier topología para atacar problemas de cómputo distribuido o paralelo a bajo costo. Estas máquinas trabajan juntas como un sistema único.

Las configuraciones de clusters se utilizan para obtener:

- Disponibilidad: Al momento de una falla del sistema en el cluster, el software de éste responde distribuyendo el trabajo del sistema con problemas, a los sistemas que quedan en el cluster.

- Capacidad de escalación: Cuando la carga general excede las capacidades de los sistemas en el cluster, es posible agregar sistemas adicionales al mismo, es decir, los usuarios podrán agregar gradualmente sistemas estándar más pequeños, según sea necesario, para satisfacer los requerimientos generales de potencia de procesamiento.

Características

Las principales características de los clusters son las siguientes:

- Alto rendimiento.
- Expandibilidad y escalabilidad.
- Soporte a alta carga de trabajo.
- Alta disponibilidad.
- Gran capacidad de cómputo.
- Bajo costo.

Un cluster consiste de componentes de hardware y software. Dentro de los componentes de hardware tenemos los siguientes:

- Procesadores (Pentium Pro, Pentium II y III, Xeon, AMD, Alpha Digital, Sun Sparc, UltraSparc).
- Memoria y cache (SIMM, SDRAM, Cache interno y externo).
- Discos y E/S (SCSI, IDE, Arreglos de RAID).
- Bus del sistema (ISA de 16 bits, VESA de 32 bits, PCI de 133 Mbytes/s).
- Interconexión de nodos (Ethernet, Fast Ethernet, Gigabit Ethernet, SCI, Myrinet,HiPPI, ATM).

Los componentes de software de un cluster son:

- Sistema operativo (Linux, Solaris, IRIX, Windows NT).
- Compiladores (GNU, Portland Group, VAST/Parallel, Digital Visual Fortran).
- Bibliotecas de comunicación (Envío de mensajes, Hebras, Memoria Compartida).
- Depuradores (Totalview, Dbx).
- Monitores de rendimiento (XPVM, XMPI, Jumpshot, CAPTools).
- Calendarizadores y balanceadores de carga (Condor,Mosix).

Clasificación

Los clusters pueden ser clasificados en varias categorías con base en factores diversos. En términos aplicativos, existen dos tipos de clusters:

- Cluster de alto rendimiento (High Performance), cuya utilización primordial se enfoca a aplicaciones científicas e ingenieriles donde se

necesitan miles de millones de operaciones de punto flotante (flops). Ejemplo Beowulf.

- Cluster de alta disponibilidad (High Availability), en donde su objetivo principal es mantener el nivel de servicio (disponibilidad) del sistema.

Además de las anteriores, existe una clase híbrida, cuyo objetivo es mantener alta disponibilidad y usar adicionalmente el sistema de forma paralela para obtener un alto rendimiento. Esta última tendencia es la más clara de la industria.

Los clusters de alta disponibilidad se dividen en:

- Cluster dedicados.
- Cluster no dedicado (Servicios a usuarios).

Arquitectura

Dos principales modelos de software se utilizan en la construcción del cluster: el disco compartido y nada compartido:

Modelo de disco compartido.

En este modelo, el software que se ejecuta en cualquier sistema dentro del cluster puede tener acceso a cualquier recurso (por ejemplo, un disco). Si dos sistemas necesitan ver los mismos datos, estos últimos se pueden leer dos veces desde el disco o copiarse de un sistema a otro.

Modelo de nada compartido.

En este modelo de software, cada sistema dentro del cluster es propietario de un subgrupo de recursos del cluster. Sólo un sistema puede poseer y acceder a la vez a un recurso en particular, aunque en caso de falla otro sistema determinado dinámicamente puede apropiarse del recurso. Además, las solicitudes de los clientes se enrutan automáticamente al sistema de quien pertenece el recurso.

Los modelos de "disco compartido" y de "nada compartido" se pueden soportar dentro del mismo cluster. Algunos tipos de software pueden explotar más fácilmente las capacidades de cluster a través del modelo de "disco compartido". Este software incluye aplicaciones y servicios que requieren sólo un acceso moderado compartido (y de lectura intensiva) a los datos, así como aplicaciones o cargas de trabajo que son difícil de dividir. Las aplicaciones que requieren escalación máxima deben utilizar el soporte de "nada compartido" del cluster.

Ventajas

Las ventajas que ofrecen los clusters son las siguientes:

- Son baratos.
- Gran poder de cómputo y gran capacidad de memoria.
- Se tiene el control completo sobre la máquina y el sistema.
- Software en desarrollo y de dominio público.
- Son buenos como supercomputadoras personales.
- Los clusters se integran fácilmente en redes existentes
- Las herramientas de desarrollo en estaciones de trabajo están muy maduras y son estándares.
- Los clusters pueden crecer fácilmente, son escalables.

Desventajas

Las desventajas que ofrecen los clusters son las siguientes:

- Sólo existe un modelo de programación eficiente: envío de mensajes.
- Recursos humanos costosos.
- Sólo son útiles en un número reducido de problemas.
- Son eficientes cuando se ejecuta un trabajo a la vez.

2.3.4 Aspectos del Software

Aunque el hardware es importante, la imagen que presenta un sistema computacional a los usuarios queda, en gran medida, determinada por el software.

Análogamente a la clasificación hecha sobre el hardware, el software también podría clasificarse como débil o fuertemente acoplado, aún cuando esta división no es clara cuando se trata del software. El software débilmente acoplado permite que las máquinas y los usuarios de un sistema distribuido sean independientes entre sí en lo fundamental, pero que interactúen en cierto grado cuando se requiera.

Como ejemplo de las combinaciones entre hardware y software tenemos a los sistemas operativos de red, que es un software débilmente acoplado con hardware débilmente acoplado (combinación más común en muchas organizaciones) y los sistemas realmente distribuidos, que es el siguiente paso en la evolución del software fuertemente acoplado en hardware débilmente acoplado.

2.3.5 Linux

Definición

Linux es un clon *libre* del sistema operativo UNIX que corre en muchas plataformas, especialmente en las computadoras personales basadas en los procesadores Intel 80386 y mejores. Soporta una gran variedad de software, incluido TeX, el Sistema X Window, el compilador GNU C/C++, TCP/IP, etc.

Historia

Linux fue desarrollado por Linus Torvalds en la Universidad de Helsinki en Finlandia, con la participación de muchos programadores de UNIX a través de Internet. Las primeras versiones datan del año 1991. Gran parte del software disponible en Linux fue desarrollado por el proyecto GNU. La última versión estable es la 2.2.14.

Características

- Sistema operativo multiusuario y multitarea (describe la habilidad de ejecutar, aparentemente al mismo tiempo, numerosos programas sin obstaculizar la ejecución de cada aplicación).
- Compatible con varios "estándares" de UNIX: IEEE POSIX.1, UNIX System V y BSD UNIX.
- Portable (hace referencia a la capacidad de transportar un sistema operativo de una plataforma a otra para que siga funcionando del mismo modo en que lo hacía).
- Código fuente distribuido libremente.

El software de Linux es generalmente liberado como una **distribución**, un conjunto de software preempacado que comprende un sistema completo, incluyendo todo lo necesario para instalarlo y correrlo.

Plataformas y distribuciones más comunes

La primera plataforma donde Linux operó fue, en las computadoras personales con procesadores 80386. Actualmente corre en todas las variantes:

- 486, Pentium, PentiumPro/II/III.
- AMD K6, K7 Athlon.
- Cyrix.
- Macintosh.
- Digital Alpha.
- Sun Sparc.
- Sistemas embebidos (aquellos que requieren de un sistema operativo).

Existen muchas compañías, asociaciones y grupos de personas que ofrecen distribuciones de Linux, entre las más importantes están:

- RedHat Linux.
- Caldera OpenLinux.
- Debian GNU/Linux.
- Slackware.
- S.u.S.E.
- Mandrake.

Requerimientos de instalación

Se ha mencionado los elementos por los que se ha elegido Linux, a continuación se explicará brevemente los requerimientos de instalación, así como el proceso mismo.

Hardware

Linux soporta la mayoría del hardware y periféricos de las computadoras personales, incluso soporta más hardware que algunas implementaciones comerciales de UNIX. A continuación se hará hincapié en los siguientes puntos:

- Motherboard y CPU.
- Requerimientos de memoria.
- Requerimientos de espacio en disco.
- Monitor y adaptador de vídeo.
- Ratón.
- Controladores de CD-ROM.
- Tarjetas ethernet.

Como ya se mencionó, Linux soporta los **procesadores** derivados del Intel 80386, esto incluye toda la familia de Pentium y los clones elaborados por otras compañías, como AMD y Cyrix. De la misma manera, están soportados todos los **coprocesadores matemáticos** estándar. Si la máquina no posee coprocesador, el kernel de Linux puede emularlo.

La mayoría de las **motherboard** están basadas en el bus PCI, pero también ofrecen slots ISA. Estas configuraciones están soportadas, como también sistemas de bus EISA y VESA.

Comparado con otros sistemas avanzados, Linux requiere poca **memoria;** se deberán tener al menos 4 megabytes de RAM, 16 o más son recomendados. Mientras más memoria se posea, más rápido será el sistema. Tener mucha memoria es tan importante como tener un procesador rápido.

La mayoría de los usuarios de Linux, reservan una parte del disco duro para espacio de intercambio ("*swapping*") que se usa como RAM virtual. Incluso si dispone de bastante memoria RAM física en la máquina, puede que quiera utilizar un área de "*swap*". El área de "*swap*" no puede reemplazar a una memoria física RAM real, pero puede permitir a su sistema ejecutar aplicaciones más grandes guardando en disco duro aquellas partes de código que están inactivas.

Linux soporta todos los controladores IDE y EIDE de **discos duros**, así como otros controladores más antiguos. Linux también soporta la mayoría de los controladores SCSI, incluidas las tarjetas Adaptec y Buslogic.

La cantidad de **espacio en disco**, depende del uso del sistema y de la cantidad de software que se desee instalar. Si bien el sistema puede correrse con

sólo 20 megabytes, requerimientos más realistas van desde 200 megabytes a 1 gigabyte o más.

Para la interfaz en modo texto, Linux soporta los **monitores** estándar, y las **tarjetas de video** Hercules, CGA, EGA, VGA, SuperVGA y aceleradas. Ambientes gráficos como el sistema X Window tienen sus propios requerimientos. La mayoría de las tarjetas de vídeo están soportadas y soporte para nuevas tarjetas se añade regularmente.

Usualmente, se usa un **ratón** en ambientes gráficos como X. Además varias aplicaciones de Linux que no están asociadas a ambientes gráficos también usan ratones. Linux soporta ratones seriales estándar como Logitech, Microsoft (2 botones), Mouse Systems (3 botones), y también la interfaz de ratón PS/2 y los dispositivos que emulan ratones como trackballs y touchpads.

Están soportados los **CD-ROM** que se conectan a interfaces IDE. También los CD-ROM tipo SCSI están soportados. Adicionalmete interfaces propietarias como NEC CDR-74 y Mitsumi están soportadas.

Muchas **tarjetas de red** Ethernet y adaptadores LAN están soportados por Linux. Linux también soporta FDDI, frame relay y tarjetas token ring y todas las tarjetas Arcnet.

Si bien Linux soporta una gran variedad de hardware, es necesario que el programa de instalación soporte el hardware necesario para realizar la instalación.

Esto dependerá del método de instalación que se elija. Si es *un medio local*, el disco duro o CD-ROM, deberá estar soportado. Si es un *medio en red*, el adaptador de red deberá estar soportado.

La *instalación en modo gráfico* sólo se podrá realizar si la combinación de tarjeta de video, monitor y ratón está sopórtada por el programa de instalación.

Software

El software necesario para iniciar la instalación dependerá del método de instalación escogido. Generalmente bastará con el disco de instalación de la distribución a usar.

Si la máquina no puede arrancar del CD-ROM, se necesitará de un *disco de arranque*. Si la instalación se realizará a través de la red se requerirá de un disco de arranque de red. Si se requiere soportar algún dispositivo PCMCIA, se requerirá el disco de arranque PCMCIA.

2.3.6 Programación paralela

En los últimos años se están aproximando más conceptos de supercomputación a la informática de bajo costo. Uno de ellos es el de la programación paralela. A continuación se analizarán algunos conceptos fundamentales asociados a la programación paralela.

Paralelismo

Paralelizar, significa dividir la carga computacional entre varios procesadores obteniendo una mejora en la relación costo y rendimiento. El concepto de paralelismo supone la introducción de varios procesadores para resolver un problema. Sabemos que un procesador diez veces más potente que un procesador de potencia normal, es mucho más caro que diez procesadores de potencia normal. Con menos inversión en hardware estamos obteniendo mucha más potencia computacional.

Sin embargo, existen varios factores tanto de índole computacional como de índole puramente físico; las comunicaciones entre procesadores son lentas en comparación con la velocidad de transmisión de datos y de cómputo dentro de un procesador y esto implica, que nunca se obtenga un escalado en el rendimiento igual o superior que el escalado en el número de procesadores, es decir, diez procesadores no van a ir diez veces más rápido que un solo procesador. Sin embargo, el incremento de velocidad se aproxima al número de procesadores que se integran en el sistema.

A todo esto hay que añadir que el factor costo sigue siendo determinante para optar por soluciones paralelas aunque no sean óptimas, ya que, aunque no obtengamos el rendimiento máximo teórico, un incremento lineal en la potencia del procesador de una máquina implica un crecimiento exponencial del precio de ésta, mientras que, con el incremento del número de procesadores, el incremento del precio es mucho menor.

Alguna desventaja deben tener los sistemas paralelos, cuando aún existen los no paralelos; y éste, es el costo humano del desarrollo del código. Los sistemas paralelos son bastante complejos de programar, y su verificación es más compleja todavía. Por ello, estos sistemas sólo encuentran su justificación en aquellos casos en los que el incremento de costes en la programación realmente está justificado porque esa potencia de cálculo es precisa. Para estas aplicaciones, sin embargo, los sistemas paralelos tienen gran éxito, encontrando como únicas barreras para su imposición final la complejidad del desarrollo y la escasez de herramientas para hacer más cómoda la verificación de programas paralelos.

Como ejemplo de estos sistemas, tenemos aquellos donde la potencia de cálculo es precisa como:

- Los grandes sistemas gestores de bases de datos
- Operaciones de búsqueda y proceso de datos a gran escala (*data minning)*; este problema tiene grandes implicaciones tanto en ciencia como en análisis de datos estadísticos para problemas de bolsa y de análisis de mercados.
- Problemas de simulación. Estos problemas son de gran interés para la industria, ya que ahorran gran cantidad de dinero en experimentos de campo que no necesitan ser realizados por poder ser analizados en etapas de desarrollo del producto mediante complejas simulaciones por computadora.

Distributividad

El único problema que tenemos en el paralelismo es que una máquina paralela es muy cara. Ahora, sí tenemos disponibilidad de un conjunto de máquinas heterogéneas de pequeño o mediano porte, cuya potencia computacional sumada es considerable podemos introducir el concepto de distributividad, en la que ya no repartimos el trabajo entre procesadores, sino entre máquinas distintas; y la información no será transmitida por el bus, sino por una red de área local.

Distribuitividad es, dividir el trabajo entre máquinas diferentes mediante una red. Este planteamiento es interesante, puesto que nos permite reaprovechar todas las máquinas que se tienen.

Se han hecho trabajos muy interesantes para permitir sistemas distribuidos de muy bajo costo y gran potencia computacional. Además, los proyectos se complementan entre sí, pudiendo ser empleados los conceptos y las soluciones propuestos por todos ellos. Destacan, entre los proyectos para hacer supercomputación distribuida a bajo costo el proyecto *PAPERS* del *MIT*, el proyecto *Beowulf* desarrollado en la *NASA* y *Extreme Linux*, proyectos que buscan hacer un supercomputador para computación distribuida con ordenadores de bajo costo y Linux.

Al igual que ocurría con el caso de la computadora paralela, van a existir factores, como la lentitud de la red frente a la velocidad del bus de la computadora paralela que van a hacer que sean necesarias más computadoras de pequeña potencia que las teóricas para igualar el rendimiento al de una computadora paralela. Sin embargo, aún teniendo en cuenta esto, la solución es mucho más barata. Y ahí entra la programación distribuida.

Programa Concurrente

Un programa concurrente es un programa que tiene más de una línea lógica de ejecución. Dicho de otra forma, es un programa que parece que varias partes del mismo se ejecutan simultáneamente (aunque no tenga por que ser así). Esto es de gran ventaja de cara al programador, ya que para muchas aplicaciones es más intuitivo el desarrollo cuando éste se realiza orientado a la concurrencia.

Un ejemplo de esto es un programa que realice determinada función y, simultáneamente, exponga datos en la pantalla. Si corren concurrentemente la interfaz gráfica y el motor de cálculo, es mucho más fácil programar ambos, ya que no tenemos que desarrollar un mecanismo de consulta continuo, interrumpiendo constantemente el motor de cálculo para pasar a ejecutar la interfaz gráfica.

También se pueden hacer correr de forma concurrente entre sí, distintas ventanas del programa, para evitar que cuando una ventana realiza una tarea, las otras se quedan colgadas hasta que acaba la ventana que está empleando el procesador. O partes del programa entre sí que sean innatamente concurrentes, para facilitar la tarea del programador.

Obsérvese que, a diferencia del paralelismo o la distribución, un programa concurrente puede correr en varios procesadores simultáneamente o no. La concurrencia es, pues, importante tanto cuando queremos hacer varias cosas a la vez.

Esta importancia de la concurrencia es especialmente destacable en sistemas operativos como Linux, que, además de concurrentes, presentan unos mecanismos de concurrencia estables, sobradamente conocidos y documentados, como son todos los mecanismos de Unix estándar: los segmentos de memoria compartida, los FIFOS, los Sockets.

Tanto sistemas paralelos como distribuidos son concurrentes; pero un sistema concurrente puede no ser ni paralelo ni distribuido, como acontece, por ejemplo, con los sistemas operativos monoprocesadores y multitarea, que son concurrentes pero no son ni paralelos ni distribuidos.

El grado de acoplamiento

El grado de acoplamiento es uno de los factores críticos a la hora de diseñar aplicaciones concurrentes y hace referencia a la cantidad de información que se comparten los procesadores. Un grado de acoplamiento alto significa que los distintos procesos comparten mucha información. Desde el punto de vista práctico, mayor acoplamiento significa mayor intercambio de información entre procesadores.

Habitualmente muchas de las soluciones más simples de plantear y las más intuitivas, emplean un grado de acoplamiento alto, ya que compartimos gran cantidad de información entre todos los nodos. Además de esto, para gran cantidad de problemas, el rendimiento, sin que tengamos en cuenta la velocidad de transmisión de información, es mayor en algoritmos con alto grado de acoplamiento, ya que es más fácil permitir un reparto de carga equivalente entre procesadores o entre máquinas, según el caso.

Sin embargo, cuanto mayor es el grado de acoplamiento, mayor es el tráfico del medio de comunicación. Este alto tráfico suele ser el cuello de botella de muchas aplicaciones con alto grado de acoplamiento, lo que supone un grave problema a la hora de programar. En soluciones distribuidas, esto puede producir una caída de rendimiento dramática, y, en redes sujetas a problemas de colisión el bloqueo total de la red, ya que se producen colisiones continuas y no va a correr la aplicación, ni ningún programa de red que necesite usar la red. Por ello, el programador ha de llegar a un difícil compromiso entre el grado de acoplamiento que es óptimo para la red (bajo) y el grado de acoplamiento de la solución más cómoda e intuitiva (alto).

En caso de hablar de una computadora paralela, se produce exactamente el mismo efecto, solamente que en lugar de producirse en la red se produce en el bus compartido, de existir éste. Computadoras que no empleen bus compartido, sino conexiones punto a punto entre algunos procesadores, topologías de hipercubo o algún mecanismo similar para evitar los bloqueos permitirán entonces, desarrollar aplicaciones paralelas con alto grado de acoplamiento. Sin embargo, estos mecanismos encarecen mucho la máquina.

2.4 ADMINISTRACIÓN DE LOS PROCESOS

Los procesos son el mecanismo básico para informar al sistema operativo respecto a actividades independientes que pueden ser planificadas para ejecución concurrente.

Dependiendo del tipo de sistema operativo y del entorno del objetivo de ejecución de programas, la división del trabajo en tareas que serán ejecutadas como procesos independientes y la asignación inicial de los atributos de los procesos puede ser efectuada o bien por el sistema operativo o bien por el programador de sistemas. En otras palabras, lo que constituirá un proceso separado en tiempo de ejecución puede provenir de:

- Una división implícita en tareas (definida por el sistema), o
- Una división explícita en tareas (definida por el programador)

2.4.1 División implícita

En general, la división implícita en tareas se aplica en sistemas operativos multitarea para multiplexar la ejecución de una serie de programas y explotar los beneficios de la concurrencia entre diferentes aplicaciones. La división explícita en tareas permite mejoras adicionales en el rendimiento al explotar la concurrencia incluida dentro de una aplicación o programa determinado.

La división implícita en tareas significa que los procesos son definidos por el sistema. Esta división implícita aparece comúnmente en sistemas de

multiprogramación de propósito general tales como los de tiempo compartido. En este enfoque, cada programa remitido para ejecución es tratado por el sistema operativo como un proceso independiente. El sistema operativo asigna valores iniciales a los atributos del proceso, tales como la prioridad de planificación y los derechos de acceso en el momento de creación del proceso basándose en el perfil del usuario y a valores predeterminados del sistema. Los procesos creados de esta manera son generalmente transitorios en el sentido que son destruidos y eliminados por el sistema después de cada ejecución.

2.4.2 División explícita

La división explicita en tareas significa que los programadores definen explícitamente cada proceso y algunos de sus atributos. Típicamente, una sola aplicación lógica se divide en varios procesos relacionados con el fin de mejorar su rendimiento. La división explícita se utiliza en situaciones en donde se desea elevar el rendimiento o controlar explícitamente las actividades del sistema. Después de dividir a mano el trabajo de la aplicación en el número deseado de tareas independientes, el programador de sistema define las fronteras de cada proceso individual. Habitualmente añade entonces un proceso padre para crear el entorno y controlar la ejecución de los procesos de aplicación individuales.

2.4.3 Relaciones entre procesos

Existen dos relaciones fundamentales entre los procesos concurrentes:

- Competición
- Cooperación

En virtud de la compartición de recursos de un solo sistema, todos los procesos concurrentes compiten unos con otros por la asignación de los recursos del sistema necesarios para sus operaciones respectivas. Además, de una colección de procesos relacionados que representen colectivamente una sola aplicación lógica suele cooperar entre sí. La cooperación es habitual entre los procesos creados como resultado de una división explícita en tareas. Los procesos cooperativos intercambian datos y señales de sincronización necesarias para orquestar su progreso colectivo.

Tanto en la competición como en la cooperación de procesos requieren el adecuado soporte por parte del sistema operativo. La competición requiere una cuidadosa asignación y protección de los recursos en términos de aislamiento de los diferentes espacios de direcciones. La cooperación depende de la existencia de mecanismos para la utilización controlada de los datos compartidos y el intercambio de señales de sincronización.

Los procesos cooperativos comparten típicamente algunos recursos y atributos, además de interactuar unos con otro. Por esas razones, con frecuencia se agrupan en lo que se denomina una familia de procesos.

Una función importante aunque raramente explícita de gestión de proceso es la asignación del procesador. Tres diferentes tipos de planificadores pueden coexistir e interactuar en un sistema operativo.

2.4.4 Planificación de procesos

La planificación hace referencia a un conjunto de políticas y mecanismos incorporados al sistema operativo que gobiernan el orden en que se ejecutan los trabajos que deben ser cumplimentados por el sistema informático. Un planificador es un módulo del sistema operativo que selecciona el siguiente trabajo que hay que admitir en el sistema y el siguiente proceso que hay que ejecutar. El objetivo primario de la planificación es optimizar el rendimiento del sistema, de acuerdo con los criterios considerados más importantes por los diseñadores del sistema.

Tipos de planificadores

En general, existen tres tipos diferentes de planificadores, que pueden coexistir en un sistema operativo complejo: planificadores a largo plazo, a medio plazo y a corto plazo

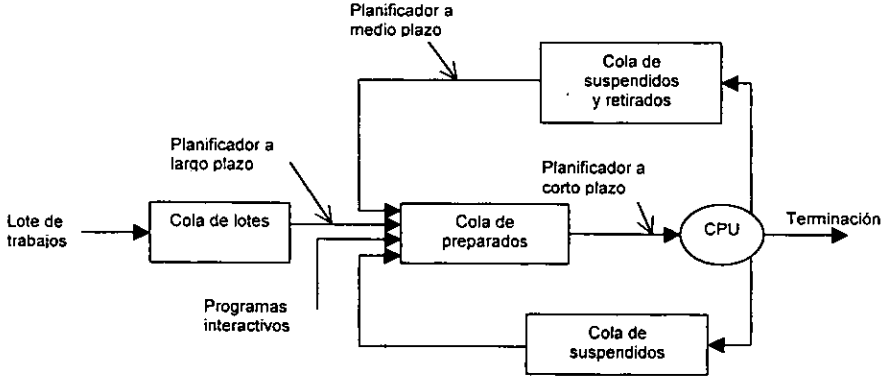

Figura 15. Planificadores

La figura 15 muestra los posibles caminos que pueden seguir los trabajos y los programas a través de los componentes y colas, representados por rectángulos de un sistema informático. Los lugares primarios de acción de los tres tipos de planificadores están marcados con flechas hacia abajo. Como se muestra en la figura, un trabajo de lotes remitido se une a la cola de lotes mientras espera ser procesado por el planificador a largo plazo. Una vez planificado para ejecución, los procesos generados por el trabajo de lotes entran a la lista de preparados para esperar la asignación del procesador efectuada por el

planificador a corto plazo. Tras ser suspendido, el proceso en ejecución puede ser retirado de memoria y colocado en almacenamiento secundario. Tales procesos son posteriormente admitidos a memoria principal por el planificador a medio plazo con el fin de ser considerados para ejecución por el planificador a corto plazo.

Planificador a largo plazo

El planificador a largo plazo, cuando está presente, trabaja con la cola de lotes y selecciona el siguiente trabajo de lotes a ejecutar. Los lotes están generalmente reservados a programas de baja prioridad y de uso intensivo de recursos (tiempo de procesador, memoria, dispositivos especiales de E/S), que pueden ser utilizados como rellenos para mantener los recursos de memoria ocupados durante periodos de baja actividad de los trabajos interactivos. Como se ha indicado, los trabajos de lotes contienen todos los datos y órdenes necesarios para su ejecución. Los trabajos de lotes contienen también generalmente estimaciones asignadas por el programador o por el sistema con respecto a sus necesidades de recursos, tales como tamaño de memoria, tiempo de ejecución esperado y necesidades de dispositivos. El conocimiento sobre el comportamiento anticipado del trabajo facilita la tarea del planificador a largo plazo.

El objetivo primordial del planificador a largo plazo es proporcionar una mezcla equilibrada de trabajos, tales como limitados por el procesador y limitados por la E/S, al planificador a corto plazo. En cierto modo, el planificador a largo plazo actúa como una válvula de admisión de primer nivel parta mantener la utilización de recursos al nivel deseado.

Por ejemplo, cuando la utilización del procesador es baja, el planificador puede admitir más trabajos para incrementar el número de procesos que se hallen en la cola de preparados, y con ello la probabilidad de disponer de alguna operación útil que espere asignación del procesador. Por el contrario, cuando el factor de utilización resulta alto y así lo refleje el deterioro del tiempo de respuesta, el planificador a largo plazo puede optar por reducir la frecuencia de admisión de los trabajos por lotes. Además, el planificador a largo plazo es invocado generalmente cada vez que un trabajo completado abandona el sistema. La frecuencia de invocación del planificador a largo plazo es por tanto dependiente del sistema y de la carga de trabajo; pero generalmente es mucho más baja que para los otros dos tipos de planificadores.

Como resultado de su relativamente infrecuente ejecución y de la disponibilidad de una estimación de las características de la carga de trabajo, el planificador a largo plazo puede incorporar algoritmos relativamente complejos y computacionalmente intensivos para admitir trabajos al sistema. En términos del diagrama de transición de estado de procesos, el planificador a largo plazo está básicamente al cargo de las transiciones de inactivo a preparado. Los procesos preparados se colocan en la cola de preparados para ser considerados por el planificador a corto plazo.

Planificador a medio plazo

Después de ejecutarse durante un tiempo, un proceso en ejecución puede resultar suspendido al efectuar una petición de E/S o al emitir una llamada al sistema. Dado que los procesos suspendidos no pueden progresar hacia su terminación hasta que la condición de suspensión sea eliminada, a veces es beneficioso retirarlos de memoria principal para dejar sitios a otros procesos.

En la práctica, la capacidad de la memoria principal puede imponer un límite al número de procesos activos en el sistema. Cuando una serie de estos procesos resultan suspendidos, permanecen residentes en memoria, la memoria principal puede llegar a reducirse hasta un nivel que afecte al funcionamiento del planificador a corto plazo dejándole pocas o ninguna opción de selección. En sistemas sin soporte de memoria virtual, ese problema puede quedar aliviado trasladando los procesos suspendidos a almacenamiento secundario. Al almacenamiento de la imagen de un proceso suspendido en memoria secundaria se le denomina intercambio, y el proceso se dice que ha sido retirado.

En el sistema representado en la figura 15, se supone que una parte de los procesos suspendidos son retirados. Los procesos restantes se supone que permanecen en memoria mientras están suspendidos.

El planificador a medio plazo tiene la misión de manejar los procesos retirados y tiene bien poco que hacer mientras un proceso permanezca suspendido. Sin embargo, una vez desaparecida la condición de suspensión de un proceso, el planificador a medio plazo intenta asignarle la cantidad necesaria de memoria principal, incorporarlo a memoria y volverlo a dejar preparado. Para funcionar adecuadamente, el planificador a medio plazo debe disponer de información respecto a las necesidades de memoria de los procesos retirados, lo cual no suele ser difícil de llevar a la práctica, ya que el tamaño real del proceso puede ser contabilizado en el momento de la retirada y almacenado posteriormente en el bloque de control del proceso afectado.

En términos del diagrama de transiciones de estado, el planificador a medio plazo controla las transiciones de suspendido a preparado de los procesos retirados. Este planificador puede ser invocado cuando quede espacio libre de memoria por efecto de la terminación de un proceso o cuando el suministro de procesos preparados caiga por debajo de un límite especificado.

Planificador a corto plazo

El planificador a corto plazo asigna el procesador entre el conjunto de procesos preparados residentes en memoria. Su principal objetivo es maximizar el rendimiento del sistema de acuerdo con el conjunto de criterios elegidos. Al estar a cargo de las transiciones de estado, de preparados a ejecución, el planificador a corto plazo debe ser invocado en cada conmutación de proceso para seleccionar el siguiente proceso a ejecutar.

En la práctica, el planificador a corto plazo es invocado cada vez que un suceso (interno o externo) hace que se modifique el estado global del sistema. Dado que cualquiera de tales cambios podría dar lugar a que el proceso en ejecución sea suspendido o a que uno o más procesos suspendidos pasen a preparados, el planificador a corto plazo tendría que ser ejecutado para determinar si tales cambios significativos han ocurrido realmente y, si verdaderamente es así, seleccionar el siguiente proceso a ejecutar.

Algunos de los sucesos introducidos hasta ahora que provocan replanificación en virtud de su capacidad de modificar el estado global del sistema son:

- Tics de reloj (interrupciones basadas en el tiempo).
- Interrupciones y terminaciones de E/S.
- La mayoría de las llamadas operacionales al sistema operativo (en oposición a las llamadas de consulta).
- El envío y recepción de señales.
- La activación de programas interactivos.

En general, cada vez que ocurre uno de estos sucesos, el sistema operativo invoca al planificador a corto plazo para determinar si debería planificarse otro proceso para ejecución.

La mayoría de los servicios de gestión de proceso del sistema operativo requieren la invocación del planificador a corto plazo como parte de su procesamiento. Por ejemplo, la creación de un proceso o la reanudación de uno suspendido añade otra entrada a la lista de preparados, y el planificador es invocado para determinar si la nueva entrada debería resultar ser el proceso a ejecutar. La suspensión de un proceso en ejecución, la modificación de la prioridad del proceso en ejecución y la terminación o aborto de un proceso son también sucesos que pueden necesitar la selección de un nuevo proceso para ejecución.

Los programas interactivos suelen entrar a la cola de preparados directamente después de ser remitidos al sistema operativo, lo que ocasiona la creación del proceso correspondiente. A diferencia de los trabajos por lotes, el flujo de programas interactivos no está limitado, y puede supuestamente saturar el sistema. Generalmente el necesario control lo proporciona indirectamente el deterioro del tiempo de respuesta, que invita a los usuarios a renunciar y probar más tarde, o al menos a reducir el ritmo de peticiones.

La figura 15 ilustra los papeles y la interacción entre los diferentes tipos de planificadores en un sistema operativo. Muestra el caso más general en que están presentes los tres tipos. Por ejemplo, un sistema operativo de grandes dimensiones podría soportar tanto programas de lotes como interactivos y confiar en la retirada a memoria para mantener una mezcla de buen comportamiento de procesos activos. Sistemas operativos de propósito especial o más pequeños pueden disponer sólo de uno o dos tipos de planificadores disponibles.

El planificador a largo plazo no se encuentra normalmente en sistemas sin soporte para lotes, y el planificador a medio plazo sólo es necesario cuando se utiliza intercambio de memoria en el sistema operativo subyacente. Cuando existe más de un tipo de planificador en un sistema operativo, la disponibilidad de soporte adecuado para comunicación e interacción es muy importante para conseguir un rendimiento satisfactorio y equilibrado. Por ejemplo, la carga de trabajo para el planificador a corto plazo la preparan los planificadores a largo y medio plazo. Si éstos no proporcionan una mezcla equilibrada de procesos limitados por cálculo y limitados por E/S, no es probable que el planificador a corto plazo rinda bien sin importar cuán sofisticado pueda ser por sí mismo.

Como resumen, podemos decir que en los sistemas de planificación generalmente se identifican tres niveles: el alto, el medio y el bajo. El nivel alto decide qué trabajos (conjunto de procesos) son candidatos a convertirse en procesos compitiendo por los recursos del sistema; el nivel intermedio decide qué procesos se suspenden o reanudan para lograr ciertas metas de rendimiento mientras que el planificador de bajo nivel es el que decide qué proceso, de los que ya están listos (y que en algún momento paso por los otros dos planificadores) es al que le toca ahora estar ejecutándose en la unidad central de procesamiento.

Criterio de planificación y rendimiento

Entre las medidas de rendimiento y los criterios de optimización más habituales que los planificadores pueden utilizar en su intento de maximizar el rendimiento del sistema se incluyen:

- Utilización del procesador.
- Productividad.
- Tiempo de retorno.
- Tiempo de espera.
- Tiempo de respuesta.

El planificador también debería tratar de aportar imparcialidad, predecibilidad y repetibilidad, de modo que cargas de trabajo similares exhiban comportamientos similares.

La **utilización del procesador** es la fracción de tiempo promedio durante la cual el procesador está ocupado. Estar ocupado se refiere generalmente a que el procesador no está inactivo, e incluye tanto el tiempo empleado ejecutando programas de usuario como ejecutando el sistema operativo.

La **productividad** se refiere a la cantidad de trabajo completada en una unidad de tiempo. Un modo de expresar la productividad es por medio del número de trabajos de usuario ejecutadas en una unidad de tiempo. Mientras mayor sea este número, más trabajo aparentemente está siendo efectuado por el sistema.

El **tiempo de retorno** se define como el tiempo que transcurre desde el momento en que un programa o trabajo es remitido hasta que es completado por un sistema. Es el tiempo consumido dentro del sistema, y puede ser expresado como la suma del tiempo de servicio (tiempo de ejecución) y el tiempo de espera del trabajo.

El **tiempo de espera** es el tiempo que un proceso o trabajo consume esperando la asignación de recursos debido a la competencia con otros en un sistema de multiprogramación. En otras palabras, el tiempo de espera es la penalidad impuesta por compartir recursos con otros procesos.

El **tiempo de respuesta** en sistemas interactivos se define como el tiempo que transcurre desde el momento en que se introduce el último carácter de una línea de orden que desencadena la ejecución de un programa o una transacción hasta que parece el primer resultado en la terminal. Generalmente se le denomina tiempo de respuesta de terminal.

Objetivos de la planificación

Una estrategia de planificación debe buscar que los procesos obtengan sus turnos de ejecución apropiadamente, conjuntamente con un buen rendimiento y minimización de la sobrecarga (*overhead*) del planificador mismo. En general, se buscan cinco objetivos principales:

- **Justicia o Imparcialidad:** Todos los procesos son tratados de la misma forma, y en algún momento obtienen su turno de ejecución o intervalos de tiempo de ejecución hasta su terminación exitosa.

- **Maximizar la Producción**: El sistema debe de finalizar el mayor número de procesos por unidad de tiempo.

- **Maximizar el Tiempo de Respuesta**: Cada usuario o proceso debe observar que el sistema les responde consistentemente a sus requerimientos.

- **Evitar el aplazamiento indefinido**: Los procesos deben terminar en un plazo finito de tiempo.
- **El sistema debe ser predecible:** Ante cargas de trabajo ligeras el sistema debe responder rápido y con cargas pesadas debe ir degradándose paulatinamente. Otro punto de vista de esto es que si se ejecuta el mismo proceso en cargas similares de todo el sistema, la respuesta en todos los casos debe ser similar.

Diseño del planificador

El diseño típico de un planificador pasa por seleccionar uno o más criterios de rendimiento primario y clasificarlos en orden relativo de importancia. El paso

siguiente es diseñar una estrategia de planificación de criterios mientras se obedecen las restricciones de diseño.

Características a considerar de los procesos

No todos los equipos de cómputo procesan el mismo tipo de trabajos, y un algoritmo de planificación que en un sistema funciona excelente puede dar un rendimiento pésimo en otro cuyos procesos tienen características diferentes. Estas características pueden ser:

- **Cantidad de Entrada/Salida**: Existen procesos que realizan una gran cantidad de operaciones de entrada y salida (aplicaciones de bases de datos, por ejemplo).

- **Cantidad de Uso de CPU**: Existen procesos que no realizan muchas operaciones de entrada y salida, sino que usan intensivamente la unidad central de procesamiento. Por ejemplo, operaciones con matrices.

- **Procesos de Lote o Interactivos**: Un proceso de lote es más eficiente en cuanto a la lectura de datos, ya que generalmente lo hace de archivos, mientras que un programa interactivo espera mucho tiempo (no es lo mismo el tiempo de lectura de un archivo que la velocidad en que una persona teclea datos) por las respuestas de los usuarios.

- **Procesos en Tiempo Real**: Si los procesos deben dar respuesta en tiempo real se requiere que tengan prioridad para los turnos de ejecución.

- **Longevidad de los Procesos**: Existen procesos que típicamente requerirán varias horas para finalizar su labor, mientras que existen otros que sólo necesitan algunos segundos.

Algoritmos de planificación

Los mecanismos de planificación, en teoría, pueden ser utilizados por cualquiera de los tres tipos de planificadores.

En general, las disciplinas de planificación pueden ser expropiativas o no expropiativas. En lotes, la no-expropiación implica que, una vez planificado, un trabajo seleccionado sigue ejecutándose hasta su terminación. Para la planificación a corto plazo, la no-expropiación implica que el proceso en ejecución retiene la propiedad de los recursos asignados, incluido el procesador, hasta que voluntariamente ceda control al sistema operativo. En otras palabras, el proceso en ejecución no se ve obligado a ceder la propiedad del procesador cuando un proceso de prioridad mayor pase a estar preparado para ejecución. Sin embargo, cuando el proceso en ejecución quede suspendido como resultado de su propia acción, digamos, por esperar la terminación de una operación de E/S, otro proceso preparado puede ser planificado.

Este esquema puede ser peligroso, ya que si el proceso contiene accidental o deliberadamente ciclos infinitos, el resto de los procesos pueden quedar aplazados indefinidamente.

Con planificación expropiativa, por otra parte, un proceso en ejecución puede ser sustituido por un proceso de mayor prioridad en cualquier instante. Esto se consigue activando el planificador cada vez que se detecta un suceso que modifica el estado del sistema. Como tales sucesos incluyen una serie de acciones además de la cesión voluntaria del control por parte del proceso en ejecución, la expropiación necesita generalmente una ejecución más frecuente del planificador. Por tanto, la planificación expropiativa responde generalmente mejor a las evoluciones de los procesos pero impone un mayor recargo, ya que cada replanificación supone una conmutación de proceso completa.

Planificación FCFS (Fist Come, Fist Served)

La disciplina de planificación más sencilla es con mucho la FCFS o primero en llegar, primero en ser atendido. La carga de trabajo se procesa simplemente en orden de llegada, son expropiaciones. La implementación del planificador FCFS es bastante directa, y su ejecución da lugar a pocos recargos.

Por no tener en consideración el estado del sistema ni las necesidades de recursos de las entidades de planificación individuales, la planificación FCFS puede dar lugar a pobres rendimientos. Como consecuencia de la no-expropiación, la utilización de componentes y la tasa de productividad del sistema puede ser bastante baja. Como no existe discriminación con base en el servicio solicitado, los trabajos cortos pueden sufrir considerables retrasos en los tiempos de retorno y de espera cuando hay uno o más trabajos largos en el sistema

La ventaja de este algoritmo es que es justo y no provoca aplazamiento indefinido. La desventaja es que no aprovecha ninguna característica de los procesos y puede no servir para un proceso de tiempo real.

Planificación SRTN (Shortest Remaning Time Next)

La planificación SRTN (a continuación el de menor tiempo restante) es una disciplina en la que la siguiente entidad de planificación, trabajo o proceso, se selecciona sobre la base del menor tiempo de ejecución restante. La planificación SRTN puede ser implementada en su variedad expropiativa o no expropiativa. La versión no expropiativa de SRTN se denomina primero el trabajo más corto (SJF, Shortest Job First). En cualquier caso, cada vez que se invoca el planificador SRTN, éste busca en la correspondiente cola (de lotes o de preparados) el proceso o trabajo con el menor tiempo de ejecución restante. La diferencia entre los dos casos se encuentra en las condiciones que conducen a la invocación del planificador y en consecuencia, en su frecuencia de ejecución. Sin expropiación, el planificador SRTN es invocada cada vez que se completa un trabajo o que el proceso en ejecución cede control al SO. En la versión expropiativa, cada vez que ocurre un sucèso que hace que un nuevo proceso esté preparado, se invoca al planificador para que compare el tiempo de ejecución restante del proceso

actualmente en ejecución con el tiempo necesario para completar la siguiente ráfaga de procesador del proceso recién llegado. Dependiendo del resultado el proceso en ejecución puede continuar o puede ser expropiado, el proceso en ejecución se une a la cola de preparados.

STRN es una disciplina de planificación probadamente óptima en cuanto a que minimiza el tiempo medio de espera de una carga de trabajo determinada. La planificación SRTN se efectúa de una manera consistente y predecible, favoreciendo los trabajos cortos. Añadiendo la expropiación, un planificador SRTN puede acomodar trabajos cortos que lleguen después de comenzar un trabajo largo. El tratamiento preferencial de los trabajos cortos en SRTN tiende a aumentar los tiempos de espera de los trabajos largos en comparación con la planificación FCFS, pero esto suele ser aceptable.

La disciplina SRTN planifica óptimamente suponiendo que los tiempos de ejecución futuros de los trabajos o procesos son conocidos con exactitud en el momento de la planificación. En el caso de la planificación a corto plazo y de las expropiaciones, se requiere de conocimiento incluso más detallado de la duración de cada ráfaga de procesador individual. La dependencia sobre el conocimiento futuro tiende a limitar la práctica de la efectividad de las implementaciones SRTN, ya que en general el comportamiento futuro del proceso es desconocido y difícil de estimar fiablemente, excepto par algunos casos deterministas muy especializados.

Las predicciones de los requisitos de ejecución de un proceso se basan generalmente en la observación del comportamiento pasado, quizás asociándolo con algunos otros conocimientos sobre la naturaleza del proceso y sus propiedades estadísticas a largo plazo, si están disponibles.

La implementación de la planificación SRTN requiere obviamente una medida bastante precisa e impone el recargo de calcular el predictor en tiempo de ejecución. Además generalmente es necesario un cierto mecanismo adicional de realimentación para efectuar correcciones cuando el predictor es manifiestamente incorrecto.

La planificación SRTN tiene importantes implicaciones teóricas y puede servir como referencia para valorar el rendimiento de otras disciplinas de planificación realizables en términos de su desviación con respecto al óptimo. En la práctica su aplicación depende de la precisión en predecir el comportamiento del trabajo y del proceso, sabiendo aumentar la precisión supone emplear métodos más sofisticados y por tanto producir un mayor recargo. La variedad expropiativa de SRTN incurre en el recargo adicional de las frecuentes conmutaciones de procesos y de invocaciones al planificador para examinar todas y cada una de las transiciones de procesos al estado preparado. Estas acciones son inútiles cuando el nuevo proceso preparado tiene un tiempo de ejecución restante mayor que el proceso en ejecución actual

La ventaja es que es muy útil para sistemas de tiempo compartido porque se acerca mucho al mejor tiempo de respuesta, además de responder a la longevidad de los procesos; su desventaja es que provoca más sobrecarga porque el algoritmo es más complejo.

Planificación por reparto del tiempo (RR, Round Robin)

En entornos interactivos, tal como en sistemas de tiempo compartido, el requisito principal es proporcionar tiempos de respuesta razonablemente buenos y, en general, compartir los recursos del sistema equitativamente entre todos los usuarios. Obviamente, sólo las disciplinas expropiativas pueden ser consideradas en tales entornos, y una de las más populares es la de reparto del tiempo, también conocida como por turnos. Básicamente, el tiempo del procesador se divide en cuotas o quantos que son asignados a los peticionarios. Ningún proceso puede ejecutarse durante más de una cuota de tiempo cuando hay otros esperando en la cola de preparados. Si un proceso necesita más tiempo para completarse después de agotar su cuota de tiempo. Se coloca al final de la lista de preparados para esperar una asignación siguiente. Esta reordenación de la lista de preparados tiene el efecto de rebajar la prioridad de planificación del proceso expropiado.

En el caso de que el proceso en ejecución ceda control al sistema operativo antes de acabar su tiempo asignado, se declara un suceso significativo y se planifica la ejecución de otro proceso. De este modo, el tiempo del procesador es asignado efectivamente a procesos con base en una prioridad rotatoria (de aquí el nombre por turnos), y cada uno de ellos recibe un 1/N aproximadamente del tiempo del procesador, en donde N es el número de procesos preparados.

La planificación por reparto del tiempo logra una cuota equitativa de los recursos del sistema. Los procesos cortos pueden ser ejecutados dentro de una única cuota de tiempo y por tanto exhiben buenos tiempos de respuesta. Los procesos largos pueden requerir varias cuotas y por tanto ser forzados a circular a través de la cola de preparados unas cuantas veces antes de terminar. Con la planificación RR, el tiempo de respuesta de los procesos largos es directamente proporcional a sus necesidades de recursos. Para procesos largos que constan de una serie de secuencias interactivas con el usuario, lo que importa principalmente es el tiempo de respuesta entre dos interacciones consecutivas. Si las necesidades computacionales entre dos de tales secuencias pueden completarse dentro de una sola cuota de tiempo, el usuario debería experimentar un buen tiempo de respuesta. RR tiende a someter a los procesos largos sin secuencias interactivas a tiempos de espera y de retorno relativamente largos. Sin embargo, tales procesos pueden ser mejor ejecutados en modo lote, y podría ser incluso deseable aconsejar a los usuarios que los remita al planificador interactivo.

La implementación de la planificación por reparto del tiempo requiere el soporte de un temporizador de intervalos. El temporizador es un programa generalmente para que interrumpa al sistema operativo cada vez que expire una cuota de tiempo forzando así la invocación del planificador. El propio planificador almacena simplemente el contexto del proceso en ejecución, lo traslada al final de

la cola de preparados y despacha el proceso que se halle a la cabeza de la cola de preparados.

La planificación por reparto del tiempo se considera a menudo una disciplina de planificación justa. También es una de las disciplinas de planificación mejor conocidas para lograr tiempos de respuesta de terminal buenos y relativamente uniformemente distribuidos. El rendimiento de la planificación por reparto del tiempo es muy sensible a la elección de la cuota de tiempo. Por esta razón, la duración de la cuota de tiempo suele ser modificable por el usuario en tiempo de generación del sistema

Planificación con expropiación basada en prioridades (DE, Event Driven)

La planificación guiada por sucesos (DE, Event Driven) es un miembro de una clase más general de planificadores basados en prioridades. En principio, cada proceso del sistema está asignado a un nivel de prioridad, y el planificador siempre elige el proceso preparado con prioridad más alta. Las prioridades pueden ser estáticas o dinámicas. En cualquier caso, sus valores iniciales son asignados por el usuario o por el sistema en el momento de la creación del proceso. El nivel de prioridad puede ser determinado como un resultado en el que influyen el valor inicial, las características, las necesidades de recursos y el comportamiento en tiempo de ejecución del proceso. En ese sentido, muchas disciplinas de planificación pueden ser consideradas como guiadas por prioridades, en donde la prioridad de un proceso representa su probabilidad de ser planificado a continuación. La planificación basada en prioridades puede ser expropiativa o no expropiativa.

Un problema habitual con la planificación basada en prioridades es la posibilidad de que los proceso de prioridad baja puedan quedar bloqueados por los de prioridad más elevada. En general la terminación de un proceso dentro de un tiempo finito a partir de su creación no puede ser garantizada con esa política de planificación. En sistemas donde tal incertidumbre no puede ser tolerada, el remedio habitual lo proporciona la prioridad de envejecimiento, en la cual la prioridad de un proceso aumenta gradualmente en función del tiempo que el proceso lleve en el sistema. Eventualmente, los procesos más antiguos conseguirán prioridades altas y esto les asegurará su terminación en tiempo finito.

Otra variante de la planificación basada en prioridades es la utilizada en los llamados sistemas de tiempo real estrictos, en donde cada proceso debe tener garantizada su ejecución antes de la expiración de un plazo. En tales sistemas, los procesos críticos se supone que tienen asignados plazos finales de ejecución. La carga de trabajo del sistema consiste en una combinación de procesos periódicos, ejecutados cíclicamente con un período conocido, y procesos aperiódicos cuyos tiempos de llegada no son generalmente predecibles. Una disciplina de planificación óptima en tales entornos es la del planificador por plazo más inmediato, que planifica para ejecución el proceso preparado cuyo plazo está más próximo a cumplirse. Otra forma de planificador, llamado planificador por mínima laxitud, también ha demostrado ser óptimo en sistemas monoprocesadores. Este

planificador selecciona el proceso preparado con menor diferencia entre el tiempo que tarda su plazo en cumplirse y su tiempo restante de computación. Es interesante que ninguno de estos planificadores es óptimo en entornos multiprocesadores.

La ventaja de este algoritmo es que es flexible en cuanto a permitir que ciertos procesos se ejecuten primero e, incluso, por más tiempo. Su desventaja es que puede provocar aplazamiento indefinido en los procesos de baja prioridad.

2.4.5 Comunicación y sincronización entre procesos

La sincronización entre procesos concurrentes cooperativos es esencial para preservar las relaciones de precedencia y para evitar los problemas de temporización relacionados con la concurrencia. Los procesos cooperativos deben sincronizarse unos con otros cuando van a utilizar recursos compartidos, tales como estructuras de datos comunes o dispositivos físicos. Dado que el sistema operativo no conoce, ni necesita conocer, la semántica de las actividades del proceso, los propios procesos cooperativos deben encargarse de sincronizar adecuadamente sus operaciones. Sin embargo, como ayuda a los usuarios y como modo de alentar una cierta disciplina, los sistemas operativos multitarea e incluso algunos lenguajes de programación proporcionan una colección de primitivas de sincronización entre procesos.

En general, hay tres formas esenciales de interacción explícitiva entre procesos:

- Sincronización entre procesos.
- Señalización entre procesos.
- Comunicación entre procesos.

Sincronización entre procesos

Es un conjunto de protocolos y mecanismos utilizados para preservar la integridad y consistencia del sistema, cuando varios procesos concurrentes comparten recursos que son reutilizables en serie. Un recurso reutilizable en serie, sólo puede ser utilizado por un proceso cada vez. Su estado, y posiblemente su operación, pueden resultar corrompidos si son manipulados concurrentemente y sin sincronización por más de un proceso, Ejemplos de ellos son las variables compartidas para lectura/escritura y los dispositivos tales como las impresoras.

Señalización entre procesos

Es el intercambio de señales de temporización entre procesos o hebras concurrentes, utilizado para coordinar su progreso colectivo. La señalización es una forma rudimentaria pero habitual de sincronización entre procesos.

Comunicación entre procesos

Los procesos cooperativos concurrentes deben comunicarse con propósitos tales como intercambiar datos, transmitir información sobre los progresos respectivos y acumular resultados colectivos. Una memoria compartida, accesible a todos los participantes, proporciona un medio sencillo y habitual de comunicación entre procesos. Para evitar errores de temporización, los procesos concurrentes deben sincronizar sus accesos a la memoria compartida.

El fallo en construir protocolos de sincronización adecuados, y obligar a utilizarlos en cada proceso que emplee recursos comunes suele dar lugar a comportamientos erráticos del sistema y a caídas del mismo, que son notoriamente difíciles de depurar. La concurrencia puede dar lugar a aumentos en la productividad cuando se implementa correctamente, pero puede degradar la fiabilidad cuando inadecuadas sincronizaciones entre procesos contaminan el sistema con errores de temporización difíciles de detectar

En conclusión podemos decir que la sincronización entre procesos es necesaria para evitar errores de temporización debidos al acceso concurrente a recursos compartidos, tales como estructuras de datos o dispositivos de E/S, por parte de procesos competidores. La sincronización entre proceso también permite el intercambio de señales de temporización (PARAR/SEGUIR) entre procesos cooperativos con el fin de preservar las relaciones específicas de precedencia impuestas por el problema que se resuelva.

La diferencia más importante entre un sistema distribuido y un sistema con un procesador es la comunicación entre procesos. En un sistema con un procesador, la mayor parte de la comunicación entre procesos supone de manera implícita la existencia de la memoria compartida. Incluso en la forma más básica de sincronización, el semáforo, hay que compartir una palabra (la propia variable del semáforo). En un sistema distribuido, no existe tal memoria compartida, por lo que toda la naturaleza de la comunicación entre procesos debe replantearse a partir de cero.

Existen ciertas reglas a las que se deben apegar los procesos respecto a la comunicación, denominadas protocolos. Para los sistemas distribuidos en un área amplia, estos protocolos toman con frecuencia la forma de varias capas, cada una con sus propios objetivos y reglas

Protocolos con capas

Para facilitar el trabajo con los distintos niveles y aspectos correspondientes a la comunicación, la organización internacional de estándares (International Standars Organization ISO) ha desarrollado un modelo de referencia que identifica en forma clara los distintos niveles. Le da nombres estandarizados y señala cuál nivel debe realizar cada trabajo. Este modelo se llama el modelo de referencia para interconexión de sistemas abiertos, conocido sólo como el modelo OSI.

El modelo OSI está diseñado para permitir la comunicación de los sistemas abiertos. Un sistema abierto es aquél preparado para comunicarse con cualquier otro sistema abierto mediante estándares que gobiernan el formato, contenido y significado de los mensajes enviados y recibidos. Estas reglas se formalizan en lo que se llama protocolos. En términos básicos, un protocolo es un acuerdo entre las partes de la forma en que debe desarrollarse la comunicación.

El modelo OSI distingue entre dos tipos generales de protocolos. Con los protocolos orientados hacia las conexiones, antes de intercambiar los datos, el emisor y receptor establecen primero en forma explícita una conexión y es probable que negocien el protocolo por utilizar. Una vez hecho esto, deben terminar la conexión, el teléfono es un ejemplo de ésto. En los protocolos sin conexión, no es necesaria una configuración previa. Simplemente, el emisor transmite el primer mensaje cuando está listo.

En el modelo OSI, la comunicación se divide en siete niveles o capas, como se muestra en la figura 16. Cada capa se encarga de un aspecto específico de la comunicación. De esta forma, el problema se puede dividir en piezas manejables, cada una de las cuales se puede resolver en forma independiente de las demás. Cada capa proporciona una interfaz con la otra capa por encima de ella. La interfaz es un conjunto de operaciones que juntas definen el servicio que la capa está preparada para ofrecer a sus usuarios.

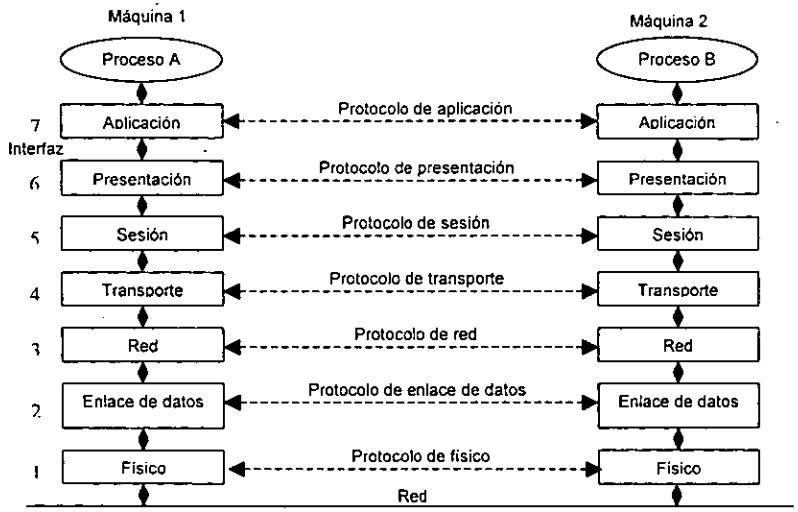

Figura 16 Capas, interfaces y protocolos en el modelo OSI

En el modelo OSI, cuando el proceso A de la máquina 1 desea comunicarse con el proceso B de la máquina 2, construye un mensaje y lo transfiere a la capa de aplicación. El software de esta capa añade un encabezado al frente del

mensaje y transfiere el mensaje resultante a través de la interfaz entre las capas 6 y 7 hacia la capa de presentación, a su vez esta capa agrega el encabezado al frente y lo transfiere y así sucesivamente, existen capas que también agregan información al final del mensaje. Al llegar a la parte inferior, la capa física transmite en realidad el mensaje y al ser recibido comienza el proceso en forma inversa, reconociendo cada capa la parte que le corresponde del mensaje.

La colección de protocolos utilizados en un sistema particular se llama una serie de protocolos o pila de protocolos.

La capa física

La capa física se preocupa por la transmisión de los ceros y unos. El número de voltios a utilizar para 0 y 1, el número de bits por segundo que se pueden enviar, el hecho de que la transmisión se lleve a cabo en ambas direcciones en forma simultánea, son todos aspectos clave en la capa física. Además, el tamaño y forma del conector en la red, así como el número de pins.

El protocolo de la capa física se encarga de la estandarización de las interfaces eléctricas, mecánicas y de señalización, de forma que. Cuando una máquina envíe un bit 0, sea en realidad recibido como un bit 0 y no como un bit 1. Uno de los estándares de esta cama es el RS-232-C para las líneas de comunicación serial.

La capa de enlace de datos

La capa física sólo envía bits. Mientras no ocurran errores todo está bien. Sin embargo, las redes reales de comunicación están sujetas a errores, por lo que es necesario cierto mecanismo para detectarlos y corregirlos. Este mecanismo es la tarea principal de la capa de enlace de datos. Lo que hace es agrupar los bits en unidades que a veces se llaman marcos, y revisar que cada marco se reciba en forma correcta.

La capa de enlace de datos realiza su trabajo por medio de la colocación de un patrón especial de bits al inicio y al final de cada marco, para señalarlos, a la vez que calcula una suma de verificación, mediante la suma de todos los bytes del marco de cierta forma. La capa de enlace de datos añade la suma de verificación al marco. Al llegar el marco, el receptor vuelve a calcular la suma de verificación a partir de los datos y compara el resultado con la suma de verificación que sigue después del marco. Si coinciden, se considera que el marco es correcto y lo acepta. En caso contrario, el receptor pide al emisor que los vuelva a transmitir. Los marcos tienen asignados números secuenciales de forma que se pueda saber con exactitud cuál es cuál.

La capa de red

En una LAN, por lo general, no existe la necesidad de que el emisor localice al receptor. Sólo tiene que colocar el mensaje en la red y el receptor lo recibe. Sin embargo, una red de área amplia consta de un gran número de máquinas, cada una de ellas con cierto número de líneas hacia otras máquinas. Para que un

mensaje llegue del emisor al receptor, tiene que hacer un cierto número de saltos y, en cada uno de ellos, elegir una línea por utilizar. La cuestión de la elección de la mejor ruta se llama ruteo y es la tarea principal de la capa de red.

El problema se complica con el hecho de que la ruta más corta no siempre es la mejor. Lo que importa en realidad es la cantidad de retraso en una ruta dada. Lo cual, a su vez, se relaciona con la cantidad de tráfico y el número de mensajes formados para la transmisión en las distintas líneas. Así el retraso puede cambiar con el curso del tiempo

Dos protocolos en la capa de red tienen un uso amplio, uno orientado hacia las conexiones (X.25, que es favorecido por los operadores de las redes públicas) y otro sin conexión (IP). Un paquete IP se puede enviar sin configuración alguna.

La capa de transporte

Los paquetes se pueden perder en el camino del emisor al receptor. Aunque ciertas aplicaciones pueden controlar su propia recuperación de los errores, otras prefieren una conexión confiable. La tarea de la capa de transporte es proporcionar este servicio. La idea es que la capa de sesión pueda enviar un mensaje a la capa de transporte, con la esperanza de que sea entregado sin pérdida alguna.

Al recibir un mensaje de la capa de sesión, la capa de transporte lo divide en pequeñas partes, de forma que cada una se ajuste a un paquete, asigna a cada uno un número secuencial y después los envía. El protocolo de transporte se llama TCP (Transmission Control Protocol)

La capa de sesión

La capa de sesión es en esencia, una versión mejorada de la capa de transporte. Proporciona el control de diálogo con el fin de mantener un registro de la parte que está hablando en cierto momento, y proporciona facilidades en la sincronización. Esto último es útil para permitir a los usuarios que inserten puntos de verificación en las transferencias de gran tamaño, de modo que si ocurre una falla, sólo sea necesario regresar al último punto de verificación, en vez de recorrer todo el camino hasta el principio.

La capa de presentación

A diferencia de las capas inferiores, preocupadas por la obtención de los bits del emisor al receptor en forma confiable y eficaz, la capa de presentación se preocupa por el significado de los bits. La mayoría de los mensajes no constan de una cadena aleatoria de bits, sino de información más estructurada, como nombres de personas, sus direcciones, etc. En la capa de presentación es posible definir registros que contengan campos como éstos y que entonces el emisor notifique al receptor que un mensaje contiene un registro particular en cierto formato. Esto facilita la comunicación entre las máquinas con distintas representaciones internas.

La capa de aplicación

La capa de aplicación es en realidad una colección de varios protocolos para actividades comunes, como el correo electrónico, la transferencia de archivos y la conexión entre terminales remotas a las computadoras en una red.

Modelo Cliente-Servidor

Los protocolos con capas a lo largo de las líneas OSI se ven como una buena forma de organizar un sistema distribuido. En efecto, un emisor establece una conexión con el receptor y entonces bombea los bits que llegan sin error, en orden al receptor, pero la existencia de tantos encabezados genera un costo excesivo. En las redes de área amplia, donde le número de bits/segundo que se pueden enviar es por lo general bajo, este costo excesivo no es serio. El factor limitante es la capacidad de las líneas, e incluso con todo el manejo de los encabezados, los CPU son muy rápidos como para mantener las líneas en ejecución a toda velocidad. Así, un sistema distribuido de área amplia podría utilizar le protocolo OSI o el protocolo TCP/IP sin pérdida en el rendimiento

Como consecuencia, la mayoría de los sistemas distribuidos basados en LAN no utilizan de modo alguno los protocolos con capas; o bien, si lo hacen, sólo utilizan un subconjunto de toda una pila de protocolos.

Además, el modelo OSI sólo se enfoca hacia un pequeño aspecto del problema, llevar los bits del emisor hacia el receptor. No dice nada acerca de la forma de estructurar al sistema distribuido. Se necesita algo adicional; ese algo adicional es el modelo cliente/servidor; éste estructura al sistema operativo como un grupo de procesos en cooperación, llamados servidores, que ofrezcan servicios a los usuarios, llamados clientes. Las máquinas de los clientes y servidores ejecutan por lo general el mismo micronúcleo y ambos se ejecutan como procesos del usuario. Una máquina puede ejecutar un proceso o varios clientes, varios servidores o combinaciones de ambas.

Para evitar un gasto excesivo en los protocolos orientados hacia la conexión como OSI o TCP/IP, lo usual es que el modelo cliente-servidor se base en un protocolo solicitud/respuesta sencillo y sin conexión. El cliente envía un mensaje de solicitud al servidor para pedir cierto servicio. El servidor hace el trabajo y regresa los datos solicitados o un código de error para indicar la razón por la cual un trabajo no se pudo llevar a cabo.

La principal ventaja es su sencillez. El cliente envía un mensaje y obtiene una respuesta. No se tiene que establecer una conexión sino hasta que ésta se utilice. El mensaje de respuesta sirve como reconocimiento de la solicitud.

Otra ventaja es la eficiencia. La pila de protocolo es más corta y por tanto más eficiente. Si todas las máquinas fueran idénticas, sólo se necesitarían tres niveles de protocolos. Las capas física y de enlace de datos se encargan de llevar los paquetes del cliente al servidor y viceversa. Esto siempre lo maneja el

hardware. No se necesita un ruteo y tampoco se establecen conexiones, por lo que no se utilizan las capas 3 y 4. La capa 5 es el protocolo solicitud/respuesta. Define el conjunto de solicitudes válidas y el conjunto de respuestas válidas a estas solicitudes. No existe administración de la sesión, puesto que éstas no existen. Tampoco se utilizan las capas superiores.

Debido a esta estructura tan sencilla, se pueden reducir los servicios de comunicación que presta el (micro)núcleo; por ejemplo, reducirse a dos llamadas al sistema, una para el envío de mensajes y otra para la recepción

Aunque la comunicación es importante, no es todo lo que hay que considerar. Íntimamente relacionado con esto está la forma en que los procesos cooperan y se sincronizan entre sí.

2.4.6 Mecanismos de sincronización entre procesos

Sin una adecuada sincronización entre procesos, la actualización de variables compartidas puede inducir a errores de temporización relacionados con la concurrencia, que son difíciles de depurar. Una de las principales causas de este problema, es la posibilidad de que los procesos concurrentes puedan observar valores temporalmente inconsistentes de una variable compartida mientras está siendo actualizada. Un método para resolver este problema es realizar las actualizaciones de las variables compartidas de manera mutuamente exclusiva. Esto se puede lograr permitiendo que, como máximo, entre un proceso cada vez a la sección crítica de código, dentro de la cual se actualiza una variable compartida o una estructura de datos particular.

El soporte hardware para el control de concurrencia forman parte íntegra de una manera u otra de prácticamente todas las modernas arquitecturas de computadoras. Algunos de los mecanismos disponibles, tales como las instrucciones de habilitar/deshabilitar interrupciones y las de cómo probar y fijar (TS), son adecuadas para la implementación de estrategias pesimistas del control de la concurrencia. Otras, tales como la instrucción comparar e intercambiar (CS), son más adecuadas para el control de concurrencia optimista. Las instrucciones TS y CS también pueden funcionar en sistemas multiprocesadores, suponiendo que la memoria compartida soporte un ciclo indivisible de lectura-modificación-escritura.

En los sistemas con un CPU, los problemas relativos a las regiones críticas, la exclusión mutua y la sincronización, se resuelven en general mediante métodos tales como los semáforos y los monitores. Estos métodos no son adecuados para su uso en los sistemas distribuidos, puesto que siempre se basan en la existencia de la memoria compartida

Exclusión mutua

Con frecuencia, los sistemas con varios procesos se programan más fácil, mediante las regiones críticas. Cuando un proceso debe leer o actualizar ciertas estructuras de datos compartidas, primero entra a una región crítica para lograr la exclusión mutua y garantizar que ningún otro proceso utilice las estructuras de datos al mismo tiempo. En los sistemas con un procesador, las regiones críticas se protegen mediante semáforos, monitores y construcciones similares.

Regiones críticas y regiones críticas condicionales

Uno de los problemas principales es que los semáforos, tal como están definidos, no están relacionados sintácticamente con los recursos compartidos que protegen. En particular, no hay declaraciones que pudieran alertar al compilador que una estructura de datos específica está compartida y que su acceso necesita ser controlado. Brinch Hansen (1972) ha propuesto una construcción de lenguaje llamada región crítica que soluciona este problema.

Una región crítica protege a una estructura de datos compartida haciéndola conocida por el compilador, el cual puede entonces generar código que garantice el acceso mutuamente exclusivo a los datos afectados.

Las regiones críticas fuerzan la utilización restringida de las variantes compartidas y evitan errores potenciales debidos al uso inadecuado de los semáforos ordinarios. En el lado negativo, las regiones críticas ayudan a estructurar solamente uno de los usos más frecuentes de los semáforos: la exclusión mutua. El otro uso, la señalización no puede ser efectuada por una región. Brinch Hansen ha propuesto otra construcción, la región crítica condicional, para abordar esta situación.

Una región crítica condicional es sintácticamente similar a una región crítica. La implementación de esta construcción permite a un proceso esperar en una condición dentro de una región crítica quedando suspendido en una cola especial, pendiente de la satisfacción de la condición correspondiente. A diferencia de un semáforo, una región crítica condicional puede admitir a otro proceso dentro de la sección crítica en ese caso. En consecuencia, un proceso que espera a una condición no impide que otros utilicen el recurso. Cuando la condición sea finalmente satisfecha, el proceso suspendido será despertado.

Puesto que es complicado seguir la pista a los cambios dinámicos de las numerosas condiciones individuales posibles, la implementación habitual de la región crítica condicional supone que cada proceso completado puede haber modificado el estado del sistema de modo que haya provocado que alguna de las condiciones por las que hay procesos esperando pueda ser satisfecha. Así, cada vez que un proceso abandona la sección crítica, se evalúan todas las condiciones que han suspendido a procesos anteriores, y si se cumplen, se despierta a uno de los procesos. Cuando ese proceso salga, se activará el siguiente proceso en espera cuya condición haya sido satisfecha. Eventualmente, no quedarán más

procesos en suspendidos, o ninguno de ellos dispondrá de las condiciones necesarias para proseguir. En cualquiera de ambos casos, un proceso externo puede ser admitido a una sección crítica inactiva.

La estrategia descrita concede precedencia a los procesos que espera, asegurando así que las nuevas entradas no puedan retrasar a los procesos en espera indefinidamente. La implementación sugerida también satisface el requisito de exclusión mutua permitiendo que como máximo un proceso acceda al recurso compartido en cada momento. Debido a que su implementación es un tanto complicada, las regiones críticas condicionales son raramente soportadas directamente en sistemas comerciales.

Semáforos

Los semáforos son un potente mecanismo para la sincronización entre procesos, para la exclusión mutua y la señalización en particular. Sus propiedades, unidas a la simplicidad y relativa facilidad de su implementación, han hecho del semáforo una herramienta popular habitualmente utilizada incluso en versiones comerciales de sistemas operativos de multiprogramación. Los semáforos han sido criticados en diferentes aspectos, con frecuencia por quienes han propuesto métodos alternativos, y se han ideado varios mecanismos para mitigar algunas de las desventajas identificadas. La mayoría de las críticas giran alrededor de dos temas principales:

1. Los semáforos no son estructurados. Hacen que la sincronización y, en última instancia, la integridad del sistema, dependan de la estricta adherencia de todos los programadores de sistemas afectados a los protocolos de sincronización específicos ideados para el problema en cuestión. Invertir las operaciones WAIT y SIGNAL, olvidar alguna de ellas o simplemente saltarlas puede corromper o bloquear fácilmente el sistema entero.

2. Los semáforos no soportan abstracción de datos. Incluso cuando se utilizan adecuadamente, los semáforos sólo pueden proteger el acceso a las secciones críticas; no pueden restringir el tipo de operaciones sobre recursos compartidos efectuadas por procesos a los que se ha concedido derechos de acceso. Por una parte, los semáforos favorecen la comunicación entre procesos mediante variables globales que están, por otra parte, sólo protegidas de los peligros de la concurrencia. Tales variables globales son vulnerables a la manipulación ilegal o sin sentido por parte de procesos que han adquirido permiso legal para modificarlas mediante la ejecución correcta de las operaciones de semáforo. Un solo proceso que errónea o maliciosamente corrompa los datos globales puede incapacitar a todos los restantes usuarios de los datos incluso después que el propio infractor haya sido eliminado del sistema

Monitores

Por lo que se refiere a la exclusión mutua, todos los mecanismos distribuidos mencionados hasta ahora tan sólo se ocupan de asegurar que como máximo se permita a un proceso acceder a un recurso compartido cada vez. Esto es sólo una parte del problema, y que un proceso al que se permite acceder al recurso puede, errónea o maliciosamente, corromperlo. Los monitores son un mecanismo de estructuración del sistema operativo que aborda esta cuestión de una manera sistemática y rigurosa.

La idea básica de los monitores es proporcionar una abstracción de datos estructural además de controlar la concurrencia, es decir, controlar no sólo la temporización sino también la naturaleza de las operaciones realizadas sobre los datos globales, para prevenir actuaciones dañinas o sin significado. Un modo de limitar los tipos de actualizaciones permisibles es proporcionar un conjunto de procedimientos de manipulación de datos fiable y bien probado. Dependiendo de los usuarios sólo son animados o realmente obligados a acceder a los datos únicamente por medio de los procedimientos suministrados, la abstracción de datos puede ser considerada como débil o fuerte, respectivamente. Las forma débil de abstracción de datos puede ser soportada en un entorno de semáforos. Sin embargo, los semáforos no proporcionan un modo sencillo de forzar el uso de procedimientos suministrados para manipulación de datos, haciendo así que el sistema sea vulnerable al descuido y la mala voluntad de los usuarios.

Los monitores avanzan un paso significativo más allá que los semáforos haciendo los datos críticos accesibles indirecta y exclusivamente mediante un conjunto de procedimientos públicamente disponibles.

Una colección de procedimientos del monitor puede así, manejar la gestión del búffer y la sincronización de las peticiones concurrentes internamente, por medio de código y variables ocultos a los usuarios.

Los procesos de usuario no tienen modo generalmente de conocer la organización interna de un monitor, como por ejemplo, el número, la identidad o la estructura de las variables. Una vez que los procedimientos públicos del monitor han sido depurados a fondo, no hay modo fácil de que los usuarios puedan corromper los datos globales. En otras palabras los monitores encapsulan los datos utilizados por los procesos concurrentes y permiten su manipulación sólo por medio de operaciones adecuada y significativamente sincronizadas. Y esto nos revela el modo cómo debería manejarse la comunicación y sincronización entre procesos.

En cuanto a su estructura, un monitor es esencialmente una colección de datos y procedimientos para su manipulación, Las variables son generalmente privadas del monitor e inaccesibles fuera de él, mientras que los procedimientos pueden ser privados o públicos.

A diferencia de los procesos, los monitores son estructuras estáticas sin vida propia. Un monitor se convierte en activo cuando uno de sus procedimientos es invocado por un proceso en ejecución. Cuando se completa la ejecución, el monitor permanece inactivo hasta la invocación siguiente. En cierto modo, los monitores pueden ser considerados como extensiones funcionales externas de los procesos de usuario. Eso mismo puede decirse también de los procedimientros externos, pero los monitores se diferencian de tales procedimientos en que proporcionan facilidades adicionales para control de concurrencia. Tales como señalización y ejecución mutuamente exclusiva de los procedimientos del monitor.

Mensajes

Al dividir una sola actividad lógica en una serie de procesos, que puedan ejecutarse concurrentemente, es posible lograr una mejora significativa en el rendimiento. Al llevar a cabo sus funciones colectivas los procesos cooperativos deben intercambiar datos y sincronizarse unos con otros. Puesto que para soportar la ejecución de procesos concurrentes es necesaria tanto la sincronización como la comunicación entre procesos, sería deseable integrar ambas funciones dentro de un solo mecanismo. Una herramienta versátil de tal tipo podría proporcionar una mayor uniformidad y facilidad de uso de los servicios del sistema y también seria probable redujese el recargo.

Los semáforos y las regiones críticas están pensados principalmente para la sincronización entre procesos. Los monitores proporcionan adicionalmente abstracción de datos, pero tienen algunos problemas de implementación y tienden a ser restrictivos en cuanto al rango de interpretaciones permisibles de los datos. Además, la implementación de esos mecanismos tiende a confiar en gran medida en la suposición de acceso común a memoria por parte de todos los procesos que intervienen en la sincronización. Por ejemplo, las variables semáforo son globales, y las estructuras de monitor (datos locales, procedimientos públicos) están generalmente centralizadas. El acceso a tales variables globales puede ocasionar considerables retrasos de comunicación en sistemas distribuidos que no disponen de memoria común. Como resultado la aplicación directa de mecanismo centralizados para control de concurrencia en entornos distribuidos suele ser ineficaz y lenta.

Los mensajes son un mecanismo relativamente sencillo tanto para comunicación como para sincronización entre procesos en entornos centralizados, además de entornos distribuidos. Muchos sistemas operativos de multiprogramación comerciales soportan algún tipo de mensajes entre procesos. El envío y recepción de mensajes es una forma estándar de comunicación entre nodos en redes de computadoras lo que hace que sea muy atractivo aumentar esta facilidad para proporcionar las funciones de comunicación y sincronización entre procesos. Por esta razón, los mensajes son muy populares en sistemas operativos distribuidos. Su importancia y utilidad han sido reconocidas también por los diseñadores de buses de sistemas de microcomputadoras de 32 bits que

proporcionan facilidades hardware especializadas para intercambios de mensajes entre procesadores con bajo recargo y elevado ancho de banda.

En esencia, un mensaje es una colección de información que puede ser intercambiada entre un proceso emisor y un receptor. Un mensaje puede contener datos, órdenes de ejecución o incluso código a transmitir entre dos o más procesos. Los mensajes suelen ser utilizados en sistemas distribuidos para transferir porciones importantes del sistema operativo y/o programas de aplicación a nodos remotos.

En general, el formato del mensaje es flexible y negociable por cada pareja específica emisor-receptor y un campo de datos.

2.5 CARACTERÍSTICAS DE LOS SISTEMAS DISTRIBUIDOS

- Poseer mecanismos de comunicación global entre los procesos, de forma que cualquier proceso pueda comunicarse con cualquier otro.
- Tener un esquema global de protección, para evitar diferentes accesos a las listas de control, bits de protección, etc.
- Poseer una misma administración de procesos. La forma en que se crean, destruyen, inician, detienen los procesos no debe variar de una máquina a otra.
- No sólo debe existir un conjunto de llamadas al sistema disponible en todas las máquinas, sino que estas llamadas deben estar diseñadas de manera que tengan sentido en un ambiente distribuido. La coordinación de actividades globales se ve facilitada como consecuencia lógica de esta situación.
- Cada núcleo debe tener un control considerable sobre sus propios recursos locales.
- El sistema de archivos debe tener la misma apariencia en todas partes. Todo archivo debe ser visible desde cualquier posición

2.6 VENTAJAS DE LOS SISTEMAS DISTRIBUIDOS

En general, los sistemas distribuidos (no solamente los sistemas operativos) exhiben algunas ventajas sobre los sistemas centralizados que se describen enseguida.

Economía

El cociente precio/desempeño de la suma del poder de los procesadores separados contra el poder de uno solo centralizado es mejor cuando están distribuidos.

Velocidad

La veracidad y rápido acceso a la información almacenada puede lograrse manteniendo varias copias de los datos en diferentes servidores.

Los sistemas distribuidos son particularmente atractivos para los usuarios quienes tienen tareas diversas, interactivas, y que requieren de una capacidad de procesamiento significativa. La dedicación del poder de procesamiento dan seguridad a los usuarios individuales y una rápida respuesta en el desempeño de la mayoría de las tareas interactivas.

Confiabilidad

Cuando uno de los componentes de un sistema distribuido falla, la mayoría de los trabajos en progreso en la red no necesita ser interrumpido. Solo el trabajo que estaba usando el componente que falló deberá ser mudado para que ocupe un componente en buen estado.

Crecimiento

El poder total del sistema puede irse incrementando al añadir pequeños sistemas, lo cual es mucho más difícil en un sistema centralizado y caro.

Distribución

El administrador de un sistema distribuido es capaz de extender el sistema dependiendo de la demanda de servicios sin remplazar ningún componente. Las estaciones de trabajo y los servidores pueden ser adicionados como sea requerido por las demandas de operación o para proporcionar nuevos servicios. El parámetro limitante es el ancho de banda de la red, ya que cada estación de trabajo activa en la red ocupa un espacio de ésta para comunicarse. Las actuales redes locales han demostrado que tienen la capacidad para soportar varios cientos de estaciones de trabajo.

Por otro lado, los sistemas distribuidos también exhiben algunas ventajas sobre sistemas aislados. Estas ventajas son:

Compartir datos

Un sistema distribuido permite compartir datos más fácilmente que los sistemas aislados, que tendrían que duplicarlos en cada nodo para lograrlo.

Compartir dispositivos

Un sistema distribuido permite acceder a dispositivos desde cualquier nodo en forma transparente, lo cual es imposible con los sistemas aislados. El sistema distribuido logra un efecto sinenergético.

Si las computadoras son componentes de un solo sistema distribuido, los periféricos como las impresoras y discos de almacenamiento pueden ser compartidos entre todas las computadoras en la red. Si éstas no están conectadas a cada una de las computadoras, deberán ocupar sus propios periféricos.

Comunicaciones

La comunicación persona a persona es factible en los sistemas distribuidos, en los sistemas aislados no.

2.7 DESVENTAJAS DE LOS SISTEMAS DISTRIBUIDOS

Así como los sistemas distribuidos exhiben grandes ventajas, también se pueden identificar algunas desventajas, algunas de ellas tan serias que han frenado la producción comercial de sistemas operativos en la actualidad. El problema más importante en la creación de sistemas distribuidos es el software: los problemas de compartición de datos y recursos son tan complejos que los mecanismos de solución generan mucha sobrecarga al sistema haciéndolo ineficiente.

Concurrencia y Paralelismo

Tradicionalmente las aplicaciones son creadas para computadoras que ejecutan secuencialmente, de manera que el identificar secciones de código "paralelizable" es un trabajo arduo, pero necesario para dividir un proceso grande en sub-procesos y enviarlos a diferentes unidades de procesamiento para lograr la distribución. Con la concurrencia se deben implantar mecanismos para evitar las condiciones de competencia, las postergaciones indefinidas, el ocupar un recurso y estar esperando otro, las condiciones de espera circulares y, finalmente, los "abrazos mortales" (deadlocks). Estos problemas de por sí se presentan en los sistemas operativos multiusuarios o multitareas, y su tratamiento en los sistemas distribuidos es aún más complejo, y por lo tanto, necesitará de algoritmos más complejos con la inherente sobrecarga esperada.

Pérdida de Flexibilidad en la Asignación de Memoria y Recursos de Procesamiento

En un sistema centralizado o en uno bajo el modelo fuertemente acoplado de multiprocesamiento, todos los recursos de procesamiento y de memoria son asignados por el sistema operativo en cualquier forma que los requiera la tarea que en ese momento esté cargada. En un sistema operativo distribuido la

capacidad de procesamiento y de memoria de las estaciones de trabajo determina el tamaño de la tarea más grande que se podrá desempeñar.

Dependencia en el Desempeño de la Red y Rentabilidad

La falla en la red local causa que los servicios a los usuarios sean interrumpidos. La sobrecarga en la red degrada el desempeño y el tiempo de respuesta a los usuarios. Mucho se ha trabajado en lo referente al diseño de redes confiables tolerantes a fallas, por lo que afortunadamente las fallas de la red ocurren muy poco frecuentemente, pero este pequeño margen de error puede considerarse como una desventaja en teoría.

Fallas de Seguridad

Para lograr una gran capacidad de expansión, muchas de las interfaces de software para los sistemas distribuidos son accesibles a los clientes. Cualquier cliente que tenga acceso a un servicio básico de comunicación puede acceder también a los servidores. Este tipo de plataforma abierta es atractiva para los desarrolladores, pero se necesitan medidas de protección de software para proteger los servicios del sistema contra violaciones de acceso intencionales o accidentales, para tener un buen control de acceso y mantener la privacidad de la información individual o de grupo.

Aún cuando existen estos potenciales problemas, las ventajas que los sistemas distribuidos poseen sobre los sistemas centralizados tienen mayor peso que las desventajas y se espera que los sistemas distribuidos adquieran cada vez mayor importancia en el futuro cercano.

3 PROGRAMACIÓN PARALELA EN EL MODELO DE INTERCAMBIO DE MENSAJES

3.1 MECANISMOS DE PROGRAMACIÓN PARA SISTEMAS DISTRIBUIDOS

Los mecanismos de programación para sistemas distribuidos son los siguientes:

- Compiladores paralelizantes.
- Lenguajes concurrentes.
- Sistemas operativos para sistemas con varios procesadores, que oculte el paralelismo a los procesos.
- Bibliotecas concurrentes.

3.1.1 Compiladores paralelizantes

El sistema de uso de los compiladores paralelizantes es bastante sencillo, se hace el programa secuencial, y el propio compilador busca el paralelismo intrínseco al programa y lo explota. Desde el punto de vista del programador, éste es el enfoque óptimo.

La desventaja es que estos compiladores tienen una escasa disponibilidad. Si bien si existen para *Fortran* y uno libre para Linux; son muy difíciles de hacer y, hasta los mejores raramente extraen todo el paralelismo inherente al código. Para Linux, sólo se tiene el *Bert 77*, que es un compilador paralelizante de dominio público.

No existen grandes soluciones conocidas de compiladores que generen automáticamente código para sistemas distribuidos.

Por último, la totalidad de los compiladores paralelizantes acaba generando código con alto grado de acoplamiento, por lo que necesitan una máquina que sea capaz de tener alto rendimiento con programas paralelos de alto acoplamiento para que el resultado sea razonable.

3.1.2 Lenguajes concurrentes

Otra opción es emplear un lenguaje concurrente. Se tienen lenguajes como el *Occam*, basado en el mecanismo de intercambio de mensajes propuesto por Dijkstra, u orientados a objetos, como es el caso de Java.

3.1.3 Sistemas operativos para sistemas con varios procesadores, que oculte el paralelismo a los procesos

Otro mecanismo existente es emplear un sistema operativo que oculte a los procesos empleados que están corriendo en una máquina paralela, de forma que los programas se diseñen de forma secuencial y, cuando varios programas se ejecuten concurrentemente, el sistema operativo asigne a cada proceso concurrente un procesador distinto para su ejecución. Este enfoque tiene el problema de necesitar una máquina paralela.

3.1.4 Bibliotecas concurrentes

Un último planteamiento es emplear sobre un lenguaje no concurrente y una máquina monoprocesador bibliotecas que nos permitan interconectar varias máquinas. Este mecanismo tiene el inconveniente de que el programador acaba haciendo todo el trabajo de concurrencia; a cambio, se da libertad para escoger cómo va a ser la comunicación. Es el planteamiento que va a tener mejor rendimiento, y permite adaptar el grado de acoplamiento del paralelismo según la arquitectura de la máquina y la naturaleza del problema, algo que es más difícil en los lenguajes concurrentes y casi imposible en los compiladores paralelizantes.

Las bibliotecas concurrentes se pueden dividir como bibliotecas por paso de mensaje, o por bibliotecas de memoria compartida. Cada una dispone de un conjunto de abstracciones para modelar tanto la comunicación entre tareas como la sincronización.

Memoria compartida

Una forma sencilla de explicar el funcionamiento de la memoria compartida es verlo como un mecanismo similar al de un pizarrón de anuncios, donde el que tiene permiso para escribir pone una nota con lo que le interesa, y el que tiene permiso para leer, lee el pizarrón. Este mecanismo parece sencillo. De hecho, es el mecanismo más intuitivo que se puede emplear para compartir información. Sin embargo, se tiene un problema: el control de acceso a la información. Dicho de otro modo, se debe tener la seguridad de que la información que leemos es semánticamente válida, y no es basura fruto de que uno de los procesos que está escribiendo está operando exactamente con el mismo dato que se está leyendo. El problema será, pues, saber cuándo poder escribir o cuándo poder leer para que el programa siga siendo correcto.

Este problema tiene unos mecanismos de resolución bastante complejos: semáforos, monitores, por lo que sólo se ha visto práctico si el grado de acoplamiento fuera tan alto que un mecanismo de paso de mensajes fuera excesivamente ineficiente.

Paso de mensajes

El paso de mensajes consiste en que un proceso manda a otro proceso un mensaje que va a contener la información que debe conocer. Por ello, no existe problema de control de acceso a la información, ya que, si a un proceso le llega un mensaje, este mensaje ya es correcto. Esto supone una primera ventaja muy importante a favor de escoger una biblioteca de paso de mensajes. Además, la sincronización puede ir en el propio envío y recepción del mensaje haciéndolos síncronos, lo que facilita la programación.

Las soluciones distribuidas habitualmente hacen uso del modelo de paso de mensajes, que se ha popularizado para las redes. Las soluciones paralelas hacen uso de los dos modelos -paso de mensajes y memoria compartida-, habitualmente en función de la arquitectura de la máquina y del lenguaje escogido; ya que en determinadas máquinas paralelas que cuenta con memoria compartida, puede resultar ventajoso emplear segmentos de memoria compartida.

Las bibliotecas que existen en el mercado sobre C para interconectar entre sí los procesos son:

- **Linda:** Está basado en el modelo *Linda*, desarrollado por la Universidad de Yale. En él, en vez de emplear modelos de memoria compartida tradicional o de paso de mensajes, se emplea un espacio de tuplas. La característica básica del espacio de tuplas, es el ser un modelo de memoria asociativa compartida, que todos los procesos pueden insertar y extraer tuplas (siempre de forma asociativa) del espacio de tuplas. La memoria subyacente puede ser distribuida, mas esto queda oculto por la propia Linda. El sistema, como tal, es un conjunto de rutinas que oculta el hardware y permite realizar las operaciones de inserción y extracción en el espacio de tuplas.

Esta abstracción es muy elegante y fácil de usar, aunque es bastante compleja.

- **Piranha**: Es una evolución del concepto de Linda, en la que los recursos computacionales se comportan como agentes inteligentes, realizando ellos mismos la búsqueda de las tareas.

- **Sistema P4:** Es una librería de macros y subrutinas desarrollada en el Laboratorio Nacional de Argonne para la programación de una gran variedad de máquinas paralelas. El sistema p4 soporta tanto el modelo de almacenamiento de memoria como el modelo de memoria distribuida (usando el paso de mensajes).

  El manejo de los procesos en el sistema p4 está basado en un archivo de configuración que especifica las máquinas, en un archivo objeto, que va a

ser ejecutado en cada máquina, el número de procesos que serán iniciados en cada host y otra información adicional.

- **Sockets**: Son portables, sobradamente documentados y conocidos. Están disponibles libremente para casi todas las computadoras que existen. Sin embargo, a pesar de ser el planteamiento más barato, tiene algunos problemas. El más importante es que la complejidad para ser programado se dispara respecto a otras abstracciones, ya que los mecanismos de comunicación entre máquinas son más complejos usando sockets que con las bibliotecas anteriormente citadas.

- **PVM:** Parallel Virtual Machine. Es un paquete de software (conjunto de librerías) para crear y ejecutar aplicaciones concurrentes o paralelas, basado en el modelo de paso de mensajes. Funciona sobre un conjunto heterogéneo de ordenadores con sistema operativo UNIX conectados por una o más redes. Permite arquitecturas de ordenadores diferentes, así como redes de varios tipos.

- **MPI:** Message Passing Interface. Es el estándar oficial de la industria de máquinas paralelas. Se completó en 1994. Está más orientada a ser un estándar de paso de mensajes para todos los supercomputadoras paralelos que un rival para otras bibliotecas, como la PVM. De hecho, se puede decir que la MPI trabaja una capa de abstracción por debajo de la PVM, por lo que quizás no sea extraño ver en un futuro la PVM corriendo sobre MPI. Ha aparecido en el mercado la *UNIFY*, que es una mezcla de MPI y PVM.

3.2 PARALLEL VIRTUAL MACHINE (PVM)

3.2.1 Definición

Parallel Virtual Machine (PVM), es un conjunto integrado de herramientas de software y bibliotecas, que emulan una estructura concurrente de propósito general, flexible y heterogénea en computadoras con distintas arquitecturas, interconectadas a través de una red de comunicación.

El principal objetivo de PVM es el permitir a tal colección de computadoras, ser usada cooperativamente en cómputos concurrentes o paralelos.

3.2.2 Historia de PVM

El proyecto PVM comenzó en el verano de 1989, en el *Oak Ridge National Laboratory.* El prototipo del sistema PVM 1.0 fué construido por *Vaidy Sunderman*

y Al *Geist*; esta versión fue usada internamente en el laboratorio y no fue liberada al público.

La versión 2 de PVM fué escrita en la Universidad de *Tennessee* y liberada en marzo de 1991. Durante los siguientes años, PVM comenzó a ser usado para muchas aplicaciones científicas. Después de una serie de cambios, y de la retroalimentación de los usuarios, se reestructuró completamente el código dando origen a la versión actual, que fue concluida en febrero de 1993.

3.2.3 Características de PVM

PVM provee una estructura unificada, dentro de la cual los programas en paralelo, pueden ser desarrollados en forma eficiente y directa, utilizando hardware ya adquirido. PVM permite que una colección de sistemas de cómputo heterogéneos, pueda ser vista como una sencilla máquina virtual paralela. PVM maneja transparentemente el ruteo de todos los mensajes, conversión de dados y calendarización de tareas, a través de una red de arquitecturas incompatibles.

PVM está diseñado para conjuntar recursos de cómputo y proveer a los usuarios de una plataforma paralela, ejecutar sus aplicaciones, independientemente del número de computadoras distintas que utilicen y de dónde se encuentren localizadas, proporcionando altos niveles de desempeño y funcionalidad.

El modelo computacional de PVM es simple y además es muy general. La interfaz de programación es deliberadamente directa, por lo que permite que estructuras simples del programa sean implementadas de una manera intuitiva. El usuario escribe su aplicación como una colección de tareas cooperativas. Las tareas tiene acceso a los recursos de PVM a través de una biblioteca de rutinas de interfaz estándar. Estas rutinas permiten la inicialización y terminación de tareas a través de la red, así como la comunicación y sincronización entre tareas.

En PVM las tareas pueden poseer un control arbitrario y estructuras dependientes, En otras palabras, en algún punto en la ejecución de una aplicación concurrente, alguna tarea puede iniciar o detener otras tareas, agregar o eliminar computadoras de la máquina virtual. Algunos procesos pueden comunicarse y/o sincronizarse con cualquier otro.

3.2.4 El Sistema PVM

El modelo computacional de PVM se fundamenta en la noción de que una aplicación está formada por muchas tareas. Cada tarea es responsable de una parte de la carga de trabajo del cálculo de la aplicación. De manera nativa, PVM soporta los siguientes lenguajes: C, C++ y Fortran. La condición para que un

lenguaje o utilerías puedan trabajar con PVM es que ésta sea capaz de enlazarse con C.

Los enlaces de C y C++ para la biblioteca de interfaz de usuario son implementadas como funciones, siguiendo la convención general de la mayoría de los sistemas de C, incluyendo UNIX.

Algunos de los principios en los cuales está fundamentado PVM son los siguientes:

- **Conjunto de máquinas configurado por el usuario:** Las tareas de las aplicaciones se ejecutan en conjunto de máquinas que son seleccionadas por el usuario para una ejecución dada de PVM. Tanto máquinas con un solo CPU, como multiprocesadores pueden ser parte del conjunto de máquinas. El conjunto puede ser alterado ya sea agregando o quitando máquinas durante la ejecución.

- **Acceso transparente al hardware:** Las aplicaciones pueden elegir explotar las capacidades de una máquina en específico en el conjunto, colocando ciertas tareas en las computadoras más apropiadas.

- **Cómputo basado en procesos:** La unidad de paralelismo en PVM es una tarea. El mapeo de proceso a procesador no es implicado o forzado por PVM; en particular, múltiples tareas pueden ejecutarse en un solo procesador.

- **Modelo explícito de envío de mensajes:** Existen colecciones de tareas, cada una realizando una parte de la carga de trabajo de una aplicación, utilizando descomposición de datos, descomposición funcional o descomposición híbrida, cooperando mediante el envío y recepción explícitos de mensajes.

- **Soporte a multiprocesadotes:** PVM utiliza las facilidades nativas de envío de mensajes en multiprocesadores para tomar ventaja del hardware local.

3.2.5 Composición de PVM

PVM está compuesto de dos partes. La primera es un demonio llamado pvmd3, el cual reside en todas las computadoras haciendo que la máquina funcione. Pvmd3 está diseñado de tal forma que cualquier usuario con cuenta válida y sin necesidad de tener privilegios de superusuario pueda instalarlo en una máquina.

La segunda parte del sistema es una biblioteca de rutinas de interfaz de PVM. Contiene un repertorio, funcionalmente completo, de primitivas que son necesarias para la interacción entre tareas de una aplicación. Estas bibliotecas contiene rutinas utilizadas por el usuario para envío de mensajes, generación de procesos, coordinación de tareas y modificación de la máquina virtual

Las aplicaciones escritas en C y C++ tienen acceso a las funciones de la biblioteca de PVM enlazándose con el archivo de biblioteca *libpvm3.a*, el cual es parte de la distribución estándar. La interfaz de Fortran con PVM está implementada como fragmentos de biblioteca que en cambio invocan las rutinas correspondientes de C, después de moldear y/o desreferenciar los argumentos apropiadamente. Entonces, las aplicaciones de Fortran requieren enlazarse con la biblioteca *libfpvm3.a*, así como también con la biblioteca de C.

El paradigma general para la programación de aplicaciones con PVM es: un usuario escribe uno o más programas secuenciales en C, C++ o Fortran que contienen llamadas a la biblioteca de PVM. Cada programa corresponde a una tarea. Estos programas son compilados para cada arquitectura en el conjunto de máquinas, y los archivos objeto resultantes son colocados en un lugar accesible para las máquinas en el conjunto de máquinas. Para ejecutar una aplicación, un usuario inicia una copia de una tarea escribiendo en la línea de órdenes de la máquina donde se encuentra tal tarea. Este proceso subsecuente inicia otras tareas de PVM, resultando eventualmente en una colección de tareas activas que entonces calculan localmente e intercambian mensajes con algunas otras para resolver el problema. Como ya se ha mencionado, las tareas interactúan a través del explícito envío de mensajes, identificándose con cada una de las otras mediante un identificador asignado por el sistema.

3.2.6 Filosofía de trabajo de un programa para PVM

Un programa para PVM va a ser un conjunto de tareas que cooperan entre sí. Las tareas se van a intercambiar información empleando paso de mensajes. La PVM, de forma transparente al programador, nos va a ocultar las transformaciones de tipos asociadas al paso de mensajes entre máquinas heterogéneas. Toda tarea de la PVM puede incluir o eliminar máquinas, arrancar o parar otras tareas, mandar datos a otras tareas o sincronizarse con ellas.

Todas las tareas en PVM son conocidas por un identificador de tarea de tipo entero (TID: *Task Identification Number*), que es el número al que se mandan los mensajes habitualmente.

Los mensajes son enviados y recibidos mediante estos *tids*. Debido a que los *tids* deben ser únicos a través de toda la máquina virtual, son generados por el demonio local y no son elegidos por el usuario. PVM también codifica información dentro de cada *tid*.

Sin embargo, no es el único método de referenciar una tarea en la PVM. Muchas aplicaciones paralelas necesitan hacer el mismo conjunto de acciones sobre un conjunto de tareas. Por ello, la PVM incluye una abstracción nueva, el grupo. Un grupo es un conjunto de tareas a las que nos podemos referir con el mismo código, el identificador de grupo. Cuando una tarea se une a un grupo, se le asigna un número de "instancia", único en ese grupo. Los números de instancia inician en 0 y se van incrementando. Las tareas también pueden emitir mensajes a grupos, a los cuales no pertenecen. Para usar alguna de las funciones de grupo, un programa deberá ser ligado con *libgpvm3.a*.

Para que una tarea entre o salga de un grupo, basta con avisar que entró o salió del grupo. Esto nos va a dotar de un mecanismo muy cómodo y potente para realizar programas empleando modelos SIMD, en el que se dividen los datos en muchos datos pequeños que sean fáciles de tratar, y después se codificará la operación simple replicándola tantas veces como datos unitarios tenga que dividirse el problema. Para trabajar con grupos, además de enlazar la biblioteca de la PVM -libpvm3.a- tenemos que enlazar también la de grupos -libgpvm3.a-.

Habitualmente para arrancar un programa para la PVM, lanzaremos manualmente desde un ordenador contenido en el conjunto de máquinas una tarea madre. La tarea la lanzaremos con el comando *spawn* desde un monitor de la máquina virtual, que arrancaremos a su vez con el comando *pvm*. Esta tarea se encargará de iniciar todas las demás tareas, bien desde su función main (será la primera en ejecutarse), bien desde alguna subrutina invocada por ella. Para lanzar nuevas tareas se emplea la función *pvm_spawn*, que devolverá un código de error, asociado si pudo o no crearla, y el TID de la nueva tarea.

El TID -Task Identifier

Es el identificador de tarea, y se comporta como el PID de un sistema Unix, sólo que de la máquina virtual. Está definido como un entero de 32 bits para aumentar el rendimiento al máximo y, al mismo tiempo, permitir direccionar el mayor número de tareas posibles.

El TID está compuesto de 4 campos. Un primer campo es el bit S, que ocupa la posición del bit más significativo. Un segundo campo es el bit G, que ocupa la siguiente posición. Un tercer campo, de 12 bits, es el campo H; y un cuarto campo, formado por los 18 bits menos significativos es el campo L.

Los campos S, G y H son de representación independiente de la máquina que corra el demonio. Los campos S y G se van a emplear para informar qué significado tienen los campos H y L, según la tabla adjunta. El campo H va a ser un campo función exclusivamente de la máquina que asigna el TID, y el campo L va a ser asignado por la máquina que determine el campo H. Estos datos ya nos permiten determinar que podremos tener hasta un máximo de 4095 máquinas en una PVM y hasta un máximo de 262143 procesos corriendo simultáneamente en cada máquina.

Todos los TIDs se mantienen sincronizados con los valores de una tabla de asignación global. Obsérvese que no hay posibilidad de colisión entre dos máquinas distintas, ya que el campo H es un valor que define unívocamente la máquina dentro de la PVM.

Los significados del TID y los rangos de los campos H y L según el valor de los bits S y G son:

S	G	H	L	Uso
0	0	1..4096	1..262143	Identificador de tarea.
1	0	1..4096	0	Identificador de demonio de PVM.
1	0	0	0	Demonio de PVM local a una tarea, para la tarea.
1	0	0	0	Shadow pvmd para el demonio que genera las tareas.
0	1	1..4096	0..262143	Dirección de multienvio
1	1			Condición de error

En el caso de la condición de error, H y L se unen formando un número entero negativo. Como el número de errores distintos es pequeño, este número es pequeño.

Las clases de arquitectura

Para evitar el problema de realizar transformaciones continuas de datos, la PVM define clases de arquitecturas. Antes de mandar un dato a otra máquina comprueba su clase de arquitectura. Si es la misma, no necesita convertir los datos, con lo que se tiene un gran incremento en el rendimiento. En caso que sean distintas las clases de arquitectura se emplea el protocolo XDR para codificar el mensaje.

Las clases de arquitectura están mapeadas en números de codificación de datos, que son los que realmente se transmiten y, por lo tanto, los que realmente determinan la necesariedad de la conversión.

Los eventos asíncronos

Los eventos asíncronos corresponden al equivalente de las señales del sistema en Linux. Se producen cuando acontece algo inaudito, y se puede pedir al sistema que interrumpa, independientemente de lo que se está haciendo, para notificar que, algo anormal está aconteciendo. Los eventos pueden viajar de una máquina a otra, para informar a todas aquéllas que lo precisen. Estos eventos corresponden como una tarea que acaba, una máquina que es eliminada o, la incorporación de máquinas.

Los distintos descriptores de mensaje de los eventos asíncronos son:

PvmTaskExit	Una tarea acaba o aborta. Una tarea colgada (aquella que no llega a su fin) no puede ser detectada, por lo que, si una tarea se cuelga no se generará un mensaje.
PvmHostDelete	Una máquina es eliminada del conjunto de máquinas o se apaga.
PvmHostAdd	Se incorpora una máquina nueva al conjunto de máquinas.

El demonio de PVM

El demonio de PVM puede ser arrancado bien por un monitor local, bien por un monitor remoto. Lo normal es que en el caso del monitor local sea arrancado como maestro y en el caso del remoto sea arrancado como esclavo, pero esto puede ser modificado en la propia forma de invocación. Después de arrancar, crea los sockets que necesita para hablar con los otros demonios y abre un archivo de históricos de error en /tmp/pvml.uid, donde uid es el identificador de usuario propietario de la PVM. Cuando el demonio de la PVM ve que su máquina es desconectada del conjunto de máquinas, mata todas las tareas conectadas a él mediante una señal SIGTERM.

Después manda un mensaje a la tabla de máquinas para que elimine su entrada. En caso de salidas más salvajes la PVM demorará un poco más en descubrir que un nodo no está funcionando, pero acabará descubriéndolo y eliminando el nodo de la tabla de máquinas.

3.2.7 Obtención de PVM

Los archivos de PVM se pueden obtener de tres diferentes maneras:

- ftp anónimo a ftp.netlib.org y se encuentra en el directorio /pvm3.
- http://www.netlib.org/pvm3/index.html.
- Solicitar mediante un correo electrónico, a netlib@netlib.org con el cuerpo del mensaje siguiente: send index from pvm3.

3.2.8 Modelo de paso de mensajes en PVM

El modelo de paso de mensajes es transparente a la arquitectura para el programador, por la comprobación de las clases de arquitectura y la posterior codificación con XDR de no coincidir las arquitecturas. Los mensajes son etiquetados al ser enviados con un número entero definido por el usuario, y pueden ser seleccionados por el receptor tanto por dirección de origen como por el valor de la etiqueta.

El envío de mensajes no es bloqueante. Esto quiere decir que el que envía el mensaje no tiene que esperar a que el mensaje llegue, sino que solamente espera a que el mensaje sea puesto en la cola de mensajes. La cola de mensajes,

además, asegura que los mensajes de una misma tarea llegarán en orden entre sí. Esto no es trivial, ya que empleando UDP se pueden enviar dos mensajes y que lleguen fuera de orden (UDP es un protocolo no orientado a conexión). TCP, por ser un protocolo orientado a la conexión, realiza una reordenación de los mensajes antes de pasarlos a la capa superior.

La comunicación de las tareas con el demonio sí se hace empleando TCP. Esto se debe a que, al ser comunicaciones locales, la carga derivada de la apertura y cierre de un canal, es muy pequeña. Además, no se tienen tantas conexiones como en el caso de la conexión entre demonios, ya que las tareas no se conectan entre sí ni con nada fuera del nodo, por lo que sólo hablan directamente con su demonio. Esto determina que serán n conexiones TCP.

La recepción de los mensajes se puede hacer mediante primitivas bloqueantes, no bloqueantes o con un tiempo máximo de espera. La PVM dotará de primitivas para realizar los tres tipos de recepción. En principio serán más cómodas las bloqueantes, ya que darán un mecanismo de sincronización bastante cómodo. Las de tiempo máximo de espera serán útiles para trabajar con ellas como si fuesen bloqueantes. Por último, la recepción de mensajes mediante primitivas no bloqueantes hace de la sincronización un problema. El mecanismo más cómodo para sincronizar en estos casos será una sincronización de barrera.

3.2.9 Ambiente Gráfico (Xpvm)

Definición

XPVM es una herramienta gráfica que sirve para visualizar los comandos e información de la consola[1] de PVM. Existen varias gráficas animadas para monitorear los programas de PVM, que proveen información de las interacciones entre las tareas en los programas paralelos de PVM, esto con el objetivo de mejorar el desempeño del programa.

XPVM está escrito completamente en C usando TCL/TK y corre como otra tarea de PVM. XPVM iniciará PVM si no esta corriendo, de estar corriendo usa el demonio ya creado. Al iniciar XPVM lee el archivo *"$HOME/.xpvm_hosts"* para crear la máquina virtual, si no existe, inicia la máquina virtual con únicamente la máquina local. Se agrega a un grupo denominado XPVM, la idea de esto es, que las tareas iniciadas fuera de la interfaz de XPVM puedan obtener el tid de XPVM. Se inicia también un archivo de rastreo en *"/tmp"* llamado *"xpvm.trace.[nombre_usuario]"*.

Ambiente

Se mencionará algunos de los botones con que cuenta XPVM y los menús que se tiene en cada uno de ellos.

[1] Interfaz entre la máquina virtual y el usuario

El primer botón es el botón "*File*", donde se puede salir de XPVM, terminar PVM. El segundo es el botón de "*Hosts*" con el que se agregan o borran máquinas, ya sea que hayan estado definidas en el archivo de configuración y estén esperando ser levantadas o, nuevas máquinas que no estén en el "*xpvm_hosts*". Las opciones son: "*Add all hosts*", "*Other hosts*", "*Sewa*", "*Erandi*".

Con el botón de "*Tasks*", tenemos la opción más utilizada que es spawn pero existen otras opciones que también despliegan ventanas donde se les proporciona los valores necesarios para la ejecución. Las opciones son "*Spawn*", "*On the fly*", "*Kill*", "*Signal*", "*Sys tasks*".

Con el botón de "*Views*" podemos activar o desactivar las ventanas que queramos tener a la vista. Las posibles ventanas son: "*Network*", "*Space time*", "*Utilization*", "*Message Queue*", "*Call Trace*", "*Task Output*", "*Event History*" y "*Done*".

Con el botón de "*Reset*" podemos eliminar tareas, vistas, puntos de rompimiento o todo. Las opciones son "*Reset PVM*", "*Reset views*", "*Reset trace*", "*Reset all*", "*Done*". Finalmente se cuenta con un botón de ayuda.

XPVM se utiliza como un monitor de desempeño en tiempo real para las tareas de PVM. Las tareas generadas por XPVM envían automáticamente la información de los eventos que describen cualquier actividad de PVM. Los programas del usuario no necesitan ser recompilados o agregarles alguna instrucción especial para obtener los archivos que guardan la información de los eventos de PVM.

A continuación se hará una breve descripción de las ventanas por las que está formada XPVM, esto, con el objetivo de poder hacer uso de ellas.

Ventana "Network"

La ventana "Network" despliega actividades de alto nivel de las máquinas dentro de la máquina virtual. Cada máquina está representada por un icono, el cual incluye la arquitectura y el nombre de la máquina. Estos iconos son iluminados en distintos colores para indicar el estado de las tareas ejecutándose en cada uno de las máquinas.

El color "Active" implica que al menos una tarea, que está en esa máquina, se encuentra ocupada ejecutando algún trabajo. El color "System" significa que no hay tareas ocupadas ejecutando cálculos de usuarios, pero que al menos una tarea está ejecutando rutinas de PVM. Cuando no hay tareas en una determinada máquina, su icono no se colorea o se mantiene en blanco. Los colores utilizados en cada caso son puestos por default a verde para máquinas activas y amarillos para actividad de PVM y además se pueden personalizar por el usuario.

Ventana "Space Time"

La ventana "*Space Time*" muestra el estado de tareas individuales que se ejecutan a través de todas las máquinas. En el lado izquierdo de la ventana están los nombres de las tareas ejecutables, precedidas por el nombre de la máquina donde están corriendo. Están ordenados por máquina.

La ventana "*Space Time*" combina tres diferentes capas de despliegue. La primera es como un diagrama de Gantt. Cada tarea es representada por una barra horizontal a lo largo del eje de las equis (x) para el tiempo común, donde el color de la barra en cada momento indica el estado de la tarea. La barra inicia en el momento en que la tarea comienza a ejecutarse y termina cuando la tarea termina normalmente.

El color "*Computing*" muestra aquéllas veces en que la tarea está ocupada, ejecutando cálculos del usuario. El color "*Overhead*" indica cuando la tarea ejecuta rutinas de comunicación de PVM, el control de las tareas, etc. El color "*Waiting*" indica aquellos períodos de tiempo utilizados en espera. Los colores por defecto para esos estados son: verde para los cálculos, amarillo para la sobrecarga, blanco para el estado de espera, rojo para el envío de mensajes.

La segunda capa de despliegue se dibuja sobre la primera, representando la comunicación entre las tareas. Cuando un mensaje es enviado entre dos tareas, se dibuja una línea roja en la barra de la tarea que envía el mensaje cuando el mensaje es enviado y termina en la barra de la tarea receptora cuando el mensaje es recibido.

El tercer despliegue aparece sólo cuando un usuario hace clic en la herramienta interesada de la ventana "*Space Time*" con el botón izquierdo del ratón. Un pequeño menú aparece dando información detallada de las tareas o mensajes.

Al hacer clic en la línea del mensaje, la ventana despliega el tiempo de envío y recepción del mensaje, así como el número de bytes y la etiqueta del mensaje.

Cuando se mueve el ratón sobre la ventana "*Space Time*" se despliega una línea vertical azúl con el tiempo correspondiente a esa línea. El usuario puede acercar cualquier área arrastrando la línea vertical con el botón central del ratón. Regresa al estado normal cuando se hace clic con el botón derecho del ratón.

Información más detallada con respecto al estado de tareas específicas, o de mensajes, se puede extraer de la ventana "*Space Time*", arrastrando con el botón izquierdo del ratón al área de la ventana. Cuando a una barra, que representa una tarea, se le hace clic, la información que se encuentra en la parte inferior cambia para indicar el momento preciso en que la tarea comenzó y finalizó, además

aparece información sobre rutinas de PVM que fueron llamadas al inicio (o en algunos casos, quizás más apropiadamente al final de la ejecución).

Ventana "Utilization"

La ventana "*Utilization*" resume la información de la ventana "*Space Time*" en cada instante de tiempo, mostrando la cantidad de cálculo, sobrecarga o estado de espera de las tareas para un mensaje en un momento determinado. Esta información se representa por tres rectángulos, coloreados, apilados verticalmente en cada instante de tiempo, con el tiempo de cálculo representado en la parte inferior, la sobrecarga en la parte media y el tiempo de espera en la parte superior. Los colores por defecto son: verde para el tiempo de cálculo, amarillo para la sobrecarga y rojo para el estado de espera.

La utilización completa de las tareas puede observarse comparando la longitud relativa de los rectángulos sobe la escala de tiempo, de tal forma que en caso de una buena utilización, las áreas "*Computing*" y "*Waiting*" dominan. Estas gráficas escalan instantáneamente al máximo número de tareas encontradas para obtener una buena representación de la utilización de las tareas.

Las ventanas "*Utilization*" y "*Space Time*" están estrechamente acopladas y comparten la misma escala de tiempo horizontal. Apuntando a ambas áreas con el puntero del ratón se crea una línea azul que representa al tiempo en ambas gráficas. Estas líneas identifican el mismo instante de tiempo en cada uno de los despliegues de las ventanas para llegar a correlacionar las dos representaciones. La ventana "*Utilization*" también es agrandada o reducida automáticamente para emparejarla con la escala de tiempo de la ventana "*Space Time*".

Ventanas "Call Trace" y "Task Output"

La información de los eventos de la ejecución del programa se muestra en la ventana "*Call Trace*", la cual muestra los detalles de los últimos eventos recibidos por cada una de las tareas de PVM que se están ejecutando actualmente. Esta información se renueva en el mismo momento, para proveer siempre la última información generada. Todas las salidas de las tareas se muestran en una ventana de texto con barra de desplazamiento.

3.2.10 Avances

Harness es un esfuerzo colaborativo entre el ORNL, la Universidad de Tennessee y la de Emory, está hecho bajo el concepto de la Maquina Virtual Distribuida que surgió con base en la idea original de la investigación sobre PVM, pero fundamentalmente vuelve a crear esta idea y explora capacidades dinámicas más allá de las que PVM puede soportar. El proyecto Harness está enfocado a tres capacidades clave dentro de la estructura de un ambiente de cómputo distribuido heterogéneo que incluye todo tipo de máquinas.

Algunas de las características sobresalientes de Harness son las siguientes:

- Interfaz paralela plug-in, que permite a los usuarios o a las aplicaciones personalizar dinámicamente, adaptar y extender las características del ambiente de cómputo.

- Control punto a punto distribuido que previene puntos de falla sencillos. Mejora sustancialmente la tolerancia a fallas, la cual está disponible para simulaciones que requieren de varios días para su ejecución.

- Múltiples máquinas virtuales distribuidas, que pueden colaborar, mezclarse o separarse. Estas características proveen una estructura para simulaciones colaborativas.

3.3 MESSAGE PASSING INTERFACE (MPI)

3.3.1 Definición

MPI es el ambiente estándar de programación en el modelo de intercambio de mensajes.

3.3.2 Historia de MPI

MPI es el producto de una serie de encuentros sostenidos entre noviembre de 1992 y enero de 1994. La construcción de este estándar comenzó con el taller sobre estándares para intercambio de mensajes en ambientes de memoria distribuida, patrocinado por el centro para la investigación en cómputo paralelo en Williamsburg, Virginia, donde se formó un comité que está constituido por miembros de aproximadamente 40 instituciones, incluyendo la mayoría de los fabricantes de máquinas paralelas, así como universidades y laboratorios gubernamentales alrededor del mundo involucrados en el cómputo paralelo.

El principal objetivo del desarrollo de MPI fue la creación de un estándar. MPI se ha convertido en uno de los ambientes de intercambio de mensajes más utilizados para la programación de aplicaciones paralelas, tanto en máquinas que contienen muchos procesadores como en conjuntos de estaciones de trabajo conectadas en red.

3.3.3 Características de MPI

- Es posible realizar comunicaciones completamente asíncronas, lo que permite traslapar operaciones de cálculo con operaciones de comunicación.

- Los grupos de procesos son sólidos y eficientes, estos últimos son enviados y recibidos por medio de estructuras de datos definidas por el usuario y no usando estructuras definidas internamente por las bibliotecas de comunicación.

- Permite la programación eficiente tanto de máquinas paralelas como de grupos de estaciones de trabajo conectadas en red.

- Es totalmente portátil. El código de MPI puede ser compilado y ejecutado en cualquier arquitectura sin necesidad de modificación alguna.

- Es un estándar. Un programa escrito en una implementación dada del MPI debe funcionar bajo cualquier otra implementación sin modificación alguna.

- Está formalmente especificado. Existe un documento oficial que contiene todas las características que debe contener una implementación estándar.

En MPI se han incluido los siguientes aspectos importantes, deseables en todo ambiente de cómputo paralelo:

- **Comunicaciones punto a punto**: El envío y recepción de mensajes entre dos procesos representan las operaciones básicas del modelo de intercambio de mensajes. Mediante estas operaciones los procesos que intervienen en un programa paralelo pueden intercambiar información y sincronizarse.

- **Operaciones colectivas**: En las operaciones colectivas interviene un conjunto de procesos. Estas operaciones incluyen recolección y difusión de datos, operaciones lógico-aritméticas usando datos distribuidos entre varios procesos, etc.

- **Grupos de procesos**: La formación de grupos de procesos permite la definición de contextos de comunicación e indica cuáles procesos intervendrán en una operación colectiva.

- **Contextos de comunicación**: Los contextos de comunicación permiten la creación de canales a través de los cuales puede transmitirse información.

- **Formación de topología**: Un grupo de procesos puede asociarse a una topología determinada. Mediante estas topologías pueden programarse esquemas de comunicación más eficientes,

- **Enlaces con los lenguajes Fortran 77 y C:** La biblioteca MPI puede ser utilizada en programas escritos en C o Fortran 77.

3.3.4 Obtención de MPI

La implementación portable de MPI es llamada mpich. Ésta fue hecha en el Laboratorio Nacional de Argonne. La principal característica es la portabilidad, ya que puede instalarse en un gran número de plataformas, tanto de estaciones de trabajo como de supercomputadoras.

Otra característica importante es que permite definir los mecanismos de comunicación e incluso que diferentes mecanismos coexistan en una sola aplicación.

Una desventaja es que, si desean obtener trazos de las operaciones de comunicación realizadas, es necesario introducir en el código llamadas a la biblioteca MPE. Mpich es gratuito.

La obtención del archivo mpich.tar.gz (aprox. 4 MB) se puede realizar de las dos maneras siguientes:

- http//www-unix.mcs.anl.gov/mpi/mpich/download o
- ftp anónimo de ftp.mcs.anl.gov en el directorio pub/mpi

4 CONSTRUCCIÓN DE LA PLATAFORMA DEL SISTEMA

Para la construcción de la plataforma del sistema, es decir, del cluster no dedicado, primero se debe tener en cuenta, que también debe dar servicio a usuarios, esto es, además de poder realizar y ejecutar programas distribuidos, también se puede manejar la máquina como un nodo simple, realizando tareas independientes.

Partiendo de esto, decimos que, para la construcción del cluster se necesita tener al menos dos máquinas conectadas físicamente en red, bajo un sistema operativo de red, que soporte procesamiento distribuido. Y una vez que se tienen estas características, se instala el software necesario para poder hacer uso de esta tecnología.

Recordemos que el desarrollo de este proyecto, busca realizar el análisis de un sistema de procesamiento distribuido, por lo que se ha considerado suficiente, realizar las conexiones en sólo dos máquinas.

Teniendo en cuenta que no se harán inversiones de dinero, se usarán las máquinas disponibles con los recursos que tengan. Por esto, las dos máquinas con las que se trabajarán son: Sewa y Erandi, cuyas características físicas son las siguientes:

	SEWA	**ERANDI**
Procesador	486 DX2	Pentium III
Velocidad	66 Mhz	500 MHz
Memoria RAM	16	32
Disco duro	1 Gb	10 Gb
Tarjeta de red	3Com Etherlink	3Com Etherlink Fast Ethernet
Monitor	SVGA	VGA
CD-ROM	No	Si

La construcción de este cluster se puede dividir en dos partes: hardware y software. Dentro del software, se tiene la instalación del sistema operativo de red y la instalación de librerías necesarias para procesamiento distribuido. A continuación se hará una descripción de la forma en que fueron conectadas las máquinas, así como el software instalado y cada uno de los pasos a seguir en dichas instalaciones.

4.1 HARDWARE

A partir de este momento, y por los recursos que tiene cada una de las máquinas; se ha determinado que Erandi será la máquina servidora no dedicada, y Sewa una estación de trabajo o cliente. Más adelante, en la instalación del software, se mencionará las diferencias en la instalación para cada máquina.

Es evidente que la selección de la máquina servidora se debió a las características físicas superiores de una, con respecto a la otra. Como se pudo observar en el cuadro mostrado anteriormente Erandi (servidor) cuenta con dos tarjetas de red, una de las cuales tendrá salida a la red pública y la otra, se usará para la conexión de la red interna, donde estará el cluster

Las principales funciones de la máquina servidora serán: levantar la máquina virtual y tener un control de la asignación de procesos a los nodos. Es necesario mencionar que al ser, un cluster no dedicado, los usuarios pueden hacer uso de la máquina virtual desde cualquier nodo, no únicamente del servidor (a diferencia de los clusters dedicados)

Las tarjetas de red, a través de las cuales se conectarán estas dos máquinas son tarjetas 3Com Etherlink, en las que se usará como medio de comunicación, cable de par trenzado (UTP). Además estarán conectadas bajo una topología punto a punto como se muestra en la figura 17.

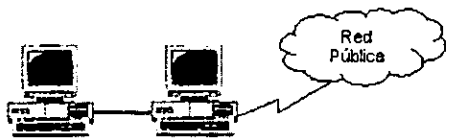

Figura 17 Conexión de Erandi y Sewa

Una vez, que se tienen dichos elementos físicos en cada una de las máquinas y que se han unido mediante el cable de par trenzado, se procede a realizar la instalación del sistema operativo de red.

4.2 SOFTWARE

Partiendo de la plataforma de hardware descrita en la sección anterior, la instalación del software la podemos dividir en las siguientes etapas:

1. Instalación del sistema operativo de red en el servidor.
2. Instalación del sistema operativo de red en el cliente.
3. Configuración de archivos individuales necesarios.
4. Instalación y configuración del software requerido para aplicaciones de procesamiento distribuido.

4.2.1 Instalación del sistema operativo de red en servidor

Primero hablaremos de la instalación en Erandi (servidor), que es una máquina que actualmente cuenta con un sistema operativo instalado (Windows), por lo que, para realizar la instalación del nuevo sistema operativo, Linux, se debe realizar particiones; además, es recomendable hacerlas.

Recuerde, que cada partición se accede como si fuera un disco diferente. Se pueden definir un máximo de cuatro particiones primarias. Como cuatro particiones pueden no ser suficientes, existen **particiones extendidas**.

En Linux, se maneja una partición especial (que a veces puede ser un archivo), llamada ***swap***. El *swap* (intercambio) es un espacio en disco reservado para ser usado como memoria virtual. El tamaño de esta partición dependerá de factores como:

- Cantidad de memoria (RAM).
- Cantidad de usuarios.
- Servicios que preste el sistema.
- Carga del sistema.

Si el sistema tiene 32 Mb o menos de memoria, se recomienda que el *swap* sea el doble de la memoria. Si se poseen más de 32 Mb de memoria, el *swap* puede ser del mismo tamaño que la memoria.

Si el sistema va a ofrecer muchos servicios, si tiene muchos usuarios o si va a tener una gran carga de trabajo, se recomienda tener un *swap* muy grande. El máximo tamaño del *swap* es 128 Mb por partición. Se puede tener más de una partición para el *swap*.

Una vez que se han recordado estos conceptos, se procede a realizar la instalación. Erandi cuenta con unidad de CD-ROM, por lo que la instalación se hará mediante esta unidad lectora.

La distribución de Linux que se va a usar es Red Hat 6.2, aunque se puede usar cualquier distribución se decidió usar ésta, debido a que es una de las mas establecidas y fáciles de obtener.

El programa de instalación será levantado desde la unidad de CD-ROM, basta con insertar el disco en la unidad e iniciar la máquina. Con el reconocimiento automático del hardware, se levantó el **Sistema X Window,** que es una interfaz de usuario gráfica, que permite correr aplicaciones con elementos gráficos, como botones, menús, barras de desplazamiento. Por lo que a partir de este momento la instalación se realizará en un ambiente gráfico.

Para arrancar sin mas parámetros especiales, pulse *enter* en el *prompt* del arranque (boot:).

El siguiente paso es seleccionar el idioma, teclado, y ratón. Es recomendable dejar los que propone el sistema, ya que esto indica, que fueron reconocidos automáticamente por el instalador.

Después, se selecciona el método y tipo de instalación, para posteriormente realizar las particiones necesarias.

Se tienen dos herramientas para realizar las particiones. Estas son: Disk Druid, que está diseñado para facilitar de modo visual el trabajo de partición del disco, la identificación de estas particiones y de la asignación de puntos de montaje de las particiones que ya están hechas y Fdisk, que es más poderosa que la anterior, se puede arrepentir e indicar abandonar la operación de modificación a la tabla de partición del disco.

El número de particiones a realizar, depende de las aplicaciones que se le darán a la máquina; pero se puede decir, que hay tres particiones principales, que son las básicas y las que usaremos en la instalación del sistema operativo en el servidor:

- / (raiz). Lugar donde se colocará toda la estructura de directorios, así como las aplicaciones de los usuarios.
- Boot. Lugar donde estará el kernel de Linux.
- *<swap>*.

Es muy importante asegurarse que el kernel del sistema operativo quede antes del cilindro 1024, esto es implícito al crear la partición de boot. Una vez determinadas las particiones se les da formato y se revisan los sectores dañados.

El siguiente paso, es la instalación de LILO. La mayoría de las distribuciones dan la opción de instalar LILO en el disco duro. LILO es un programa que se instala en el registro maestro de arranque del disco, y está preparado para arrancar varios sistemas operativos, entre los que se incluyen MS-DOS, Windows y Linux, permitiéndole elegir que sistema quiere arrancar en cada momento. Como en este caso, Erandi contará con dos sistemas operativos, será necesario configurarlo.

Una vez que se ha instalado LILO, se deben proporcionar los parámetros de red, la configuración de *firewall*, autenticación y la configuración de la zona horaria.

La configuración de la primera interfaz de red, la que tendrá acceso a la red pública, será configurada en este momento, dándole, los siguientes parámetros:

Dirección IP	132.248.59.52
Broadcast	132.248.59.255
Máscara	255.255.255.0

Una de las actividades finales en la instalación, es asignar la contraseña de *root*. Así como la creación de cuentas de acceso. Para terminar con la configuración de la tarjeta de vídeo.

La configuración de la red interna para intercomunicación entre nodos, eth1, se llevará a cabo explícitamente y después de haber terminado la instalación del sistema operativo.

Ya que esta red está aislada de la red pública, se puede asignar arbitrariamente los datos; como la red a utilizar, el rango de direcciones, etc. Arbitrariamente se seleccionó la subred 192.168.0.0, con máscara de subred de 24 bits (255.255.255.0). Esto proporciona 253 direcciones IP utilizables.

Para establecer esta configuración de red, se escribe siendo root, (usuario especial que tiene acceso libre a todas las partes del sistema) en el prompt lo siguiente:

 # ifconfig eth1 192.168.0.254 netmask:255.255.255.0
 # route add -net 192.168.0.254 netmask:255.255.255.0 dev eth1

4.2.2 Instalación del sistema operativo de red en cliente

Para la instalación del sistema operativo de red en Sewa (cliente), se procedió de manera diferente, debido a que esta máquina no cuenta con una unidad de CD-ROM, por lo que la instalación se realizará mediante la red, vía NFS.

Para realizar este tipo de instalación, se debe exportar el sistema de archivos o directorios en la máquina donde se colocará el CD-ROM.

Para empezar se requiere tener instalados nfs-utils y portmap. Veremos si estos están instalados con la siguiente línea de comando en la máquina servidora:

rpm -q nfs-utils portmap

lo cual debe regresar algo como lo siguiente:

nfs-utils-0.3.1-0.6.x.1
portmap-4.0-19

de lo contrario deben instalarse los RPMS respectivos, que se encuentran en el CD-ROM de instalación.

El siguiente paso es determinar qué directorio se va a compartir agregándose en el arcjivo */etc/exports*. Este archivo determina con qué máquinas y en qué modo lo haremos. Se puede especificar una dirección IP, nombre de alguna máquina, o un patrón común con comodín para definir que máquinas pueden acceder.

/mnt/cd-rom 192.168.0.0/255.255.255.0 (ro)

En este caso, se está definiendo que se compartirá */mnt/cd-rom* a todas las máquinas con dirección IP dentro de la subred 192.168.0.0 y máscara 255.255.255.0. Los parámetros entre paréntesis indican los privilegios con que se exporta el recurso.

Ya que se definieron los volúmenes a compartir, solo resta iniciar o reinciar el servicio *nfs*. Utilice cualquiera de las dos líneas dependiendo el caso:

/etc/rc.d/init.d/nfs start
/etc/rc.d/init.d/nfs restart

Para la máquina donde se instalará Linux, la que no tiene unidad de CD-ROM (Sewa), se necesitará un disco de arranque para red. La imagen del disco de arranque para red es bootnet.img, y se encuentra en el directorio */images* del CD Red Hat Linux/Intel.

A continuación, el programa de instalación comprobará el sistema e intentará identificar la tarjeta de red. En la mayoría de las ocasiones, el controlador puede localizar la tarjeta de red automáticamente. Si es incapaz de identificar la tarjeta de red, le pedirá que elija el controlador que soporta la tarjeta de red y que especifique cualesquiera opciones necesarias para que el controlador la localice y la reconozca.

Una vez que el programa de instalación haya configurado la tarjeta de red, le mostrará varios diálogos para configurar la red TCP/IP del sistema. Los datos que se ocuparán son los siguientes:

Dirección IP	192.168.0.253
Broadcast	192.168.0.255
Mascara	255.255.255.0

NFS Server Name	192.168.0.254
NFS Directory	/mnt/cd-rom

En el primer diálogo se pide la dirección IP y otras direcciones de red. Introduzca la dirección IP que está usando durante la instalación y pulse Enter. El programa de instalación intenta adivinar la máscara de red basándose en la dirección IP; puede cambiar la máscara de red si no es correcta. El programa de instalación supone las direcciones de la pasarela por defecto y el servidor de nombres primario a partir de su dirección IP y máscara de red; puede cambiarlos si no son correctos.

Tras el primer cuadro de diálogo, puede ver un segundo cuadro. Este le solicitará un nombre de dominio, un nombre de máquina y otra información de red.

Finalmente, ya que se han dado los datos de red, comienza la instalación del sistema operativo vía red, repitiéndose los pasos realizados en la instalación del sistema operativo del servidor.

4.2.3 Configuración de archivos individuales necesarios

Existen archivos que son útiles para el manejo de máquinas en la red, por lo que se mencionan a continuación.

/etc/hosts

Este archivo lleva una lista de direcciones IP y nombres de máquinas que les corresponden. En general, */etc/hosts* sólo contiene entradas para su máquina y quizá, alguna otra "importante", como servidores de nombres o pasarelas. Su servidor de nombres local proporciona a otras máquinas traducción automática del nombre de la máquina a su dirección IP.

Por ejemplo, en Sewa, se tiene el siguiente archivo:

```
127.0.0.1       localhost.localdomain   localhost   sewa
192.168.0.254   erandi.fi-b.unam.mx     erandi
132.248.59.2    cronos.fi-b.unam.mx     cronos
```

Si sólo usa el "*loopback*", la única línea necesaria es la que tiene el número 127.0.0.1, añadiendo tras *localhost* el nombre de su máquina.

/etc/host.conf

Este archivo dice al sistema cómo resolver los nombres de las máquinas. Debe contener dos líneas:

```
order hosts,bind
multi on
```

Estas líneas indican a los mecanismos de resolución que empiecen buscando en el archivo */etc/hosts* y luego, pregunten al servidor de nombres, si existe. La entrada *multi* permite que para un nombre de máquina haya varias direcciones IP en */etc/hosts*.

/etc/resolv.conf

En este archivo se configura el mecanismo de resolución, especificando la dirección del servidor de nombres y el nombre del dominio de su máquina. El dominio es como un nombre de máquina "mutilado".

Es un archivo de texto con una palabra clave por línea. Hay tres palabras clave de uso frecuente, que son:

Domain	Palabra clave que especifica el nombre de dominio local.
Search	Especifica una lista de dominios alternativos para completar el nombre de una máquina.
Nameserver	Puede utilizarse varias veces, especifica una dirección IP de un servidor de nombres de dominio para consultarlo cuando se resuelvan nombres.

En nuestro caso se tiene el siguiente archivo, que es el mismo para ambas máquinas:

domain fi-b.unam.mx
search fi-b.unam.mx
nameserver 132.248.10.2

/etc/hosts.equiv

La equivalencia de anfitrión o acceso de anfitrión confiable la configura el administrador y contiene la lista de anfitriones confiables, esto es, le ofrece al usuario un sistema para tener acceso a sus cuentas en sistemas remotos sin tener que utilizar los nombres de registro y contraseñas. Este archivo se usa por r* *(rlogin,rcp rsh)*. El formato es una línea con nombres de máquinas

Cada entrada al archivo *host.equiv* es confiable. Esto es, los usuarios que están en una máquina con nombre tienen acceso a ciertas equivalentes en esta máquina sin necesidad de contraseña. Sin embargo, no es aplicable a *root*. Obviamente, esto supone un gran problema de seguridad, por lo que es recomendable que este archivo este vació o no exista.

El archivo que se colocó en Sewa es el siguiente:

192.168.0.254
localhost
127.0.0.1

$HOME/ .rhosts

El archivo *.rhosts* es un archivo de cada usuario que indica las máquinas a las que permite ejecutar un comando remoto sin pedir *password*. Este archivo debe estar en el directorio HOME, además de ser leído sólo por el usuario. El formato del archivo es el siguiente:

Máquina1 login_en maquina1
Máquina2 login_en maquina2

Este archivo crea un mecanismo de confiabilidad bastante similar al de */etc/hosts.equiv*, por lo que no fue necesario usarlo, ya que se definió el *hosts.equiv*.

$HOME/.netrc

En este archivo se pueden especificar, en texto claro, nombres de máquinas, nombres de usuarios y contraseñas de sistemas remotos, de forma que al conectar a ellos, la transferencia de estos datos se realiza automáticamente, sin ninguna interacción del usuario.

La existencia del archivo *.netrc* en los HOME de los usuarios se puede convertir en un grave problema de seguridad, si un atacante consigue leer el archivo, automáticamente, obtiene nombres y claves de usuario.

La diferencia entre *.rhosts* y *.netrc*, es que en el primer caso, se consigue que la clave no se envíe a través de la red, mientras que en el segundo caso, lo único que se consigue es no teclear el usuario y la contraseña explícitamente, se envía automáticamente.

4.2.4 Instalación de Software

Una vez, que se tienen las dos máquinas conectadas en red, a las que se ha instalado Linux, como sistema operativo, y se han configurado los archivos mencionados anteriormente; se explicará el proceso de instalación y configuración de las dos bibliotecas que usan el paso de mensajes como medio de comunicación entre procesos.

Se ha elegido instalar dos bibliotecas (PVM, MPI), ya que a través de éstas, se puede crear la máquina virtual, concepto que engloba la conexión de más de una máquina trabajando como si fuera una.

A continuación se explicará la instalación y configuración de cada una de ellas. El proceso de instalación de estas bibliotecas es el mismo en ambas máquinas, por esto, no se ha hecho distinción en la instalación.

Instalación de PVM

PVM es simple para instalar y utilizar, además no requiere de privilegios especiales para su instalación. Los pasos para instalar PVM son los siguientes, considerando que tenemos el archivo pvm3_4_3.tgz.

1. Descomprimir el archivo pvm3_4_3.tgz con la instrucción mostrada a continuación; esto generará el archivo pvm3_4_3.tar

   $ gunzip pvm3_4_3.tgz

2. Generar estructura de directorios mediante el comando tar:

   $ tar xvf pvm3_4_3.tar

Algunos de los directorios más importantes que se generarán son:

Bin/$PVM_ARCH	Programas ejecutables de PVM y del usuario.
Conf	Archivos de configuración para todas las arquitecturas.
Console	Fuentes para la consola de PVM.
Doc	Documentación.
Examples	Fuentes de los programas ejemplos.
Include	Librerías para los programas de PVM.
Lib	Ejecutables genéricos del sistema.
src	Fuentes de la librería libpvm y el demonio pvmd.

3. Colocar variables PVM_ROOT, PVM_ARCH y PATH, que se mencionan en el apartado de configuración.

4. Compilar.

   $ cd $HOME/pvm3
   $ make

Configuración de PVM

Para realizar la configuración completa de PVM, se deben realizar los siguientes dos pasos, aunque es necesario mencionar que sólo el primero es obligatorio.

- Configurar las variables de entorno PVM_ROOT, PVM_ARCH y PATH.
- Crear/Configurar el archivo de máquinas.

PVM utiliza dos variables de ambiente cuando arranca y se está utilizando. Cada usuario de PVM necesita dar de alta estas variables para poder utilizar PVM. La primera de ellas es PVM_ROOT, a la cual se le asigna la localización del directorio donde está instalado PVM. La segunda es PVM_ARCH, la cual informa a PVM la arquitectura de la máquina en cuestión y de qué archivos ejecutables tomar del directorio PVM_ROOT. Es importante que a la variable PATH se le agregue la ruta de los directorio *bin* y *lib* de PVM.

El método más sencillo es colocar estas variables en el archivo *.cshrc* (se asume que está utilizando *csh*).

```
$setenv PVM_ROOT $HOME/pvm3
$setenv PVM_ARCH LINUX
set path=($path $PVM_ROOT/lib)
set path=($path $PVM_ROOT/lib/$PVM_ARCH)
```

Esto le indica que los archivos de PVM se encuentran debajo del HOME en un directorio llamado pvm3.

Si el *shell* que estamos manejando es *bash*, debemos colocar en el archivo *.profile*.

```
export PVM_ROOT=$HOME
export PVM_ARCH=LINUX
PATH=$PATH:$PVM_ROOT/bin/$PVM_ARCH:$PVM_ROOT/lib/$PVM_ARCH
```

Para cambiar el *shell* actual se hace con el comando *chsh* y se coloca la ruta completa de la localización de éste*(/bin/[shell])*. Algunos de los posibles *shells* son:

- Sh Bourne Shell
- Csh C Shell
- Bash Bourne Again Shell

Para los usuarios de *csh*: Nótese que el colocar las variables en el *.login* no tiene el mismo efecto, este archivo solo es leído al entrar mientras que el *.cshrc* es leído cada que *csh* inicia. PVM necesita tener las variables de ambiente colocadas cuando inicia un demonio esclavo con "*rsh host pvmd ...*", entonces, deben ser colocadas en *.cshrc*.

Para aquellos que usan un *shell* que no siempre lee el *script* de inicio (por ejemplo, *sh*, *ksh)*, existe otro modo de colocar las variables para PVM. Antes de correr el ejecutable de PVM y el demonio buscan cualquier comando en el archivo $HOME/*.pvmprofile*, si éste existe.

A continuación se describen las variables de ambiente que son leídas por PVM y pueden ser colocadas para personalizar el ambiente de trabajo. Como se mencionó anteriormente se pueden agregar al *.cshrc* o *.profile* o cualquier archivo

de inicio del *shell* y también se debe agregar los directorios de PVM a la ruta de ejecución.

PVM_ROOT	Ruta donde están instaladas las librerías y los programas del sistema. Esta variable debe ser colocada en cada máquina donde se use PVM para que funcione. No hay valor por defecto.
PVM_TMP	Ruta para los archivos temporales de PVM, como el archivo de comunicación de los demonios "pvmd.<uid>" y el archivo de bitácora "pvml.<uid>". Usa esta variable para definir un directorio diferente a /tmp.
PVM_RSH	Ruta del programa *rsh*, si es diferente a la definida en el archivo de configuración $PVM_ROOT/conf/$PVM_ ARCH.def. Esta variable puede ser usada para reemplazarse con *ssh*, agregando seguridad.
PVM_PATH	Ruta donde serán buscados los programas a ejecutar. Por defecto es $HOME/pvm3/bin/$PVM_ARCH y $PVM_ROOT/bin/$PVM_ARCH para las aplicaciones de PVM. Esta variable no sobreescribe la opción *ep* del archivo de máquinas.
PVM_WD	Es el directorio de trabajo del sistema para la creación de procesos con *spawn*, por defecto lo realiza en $HOME, pero por conveniencia en el uso de rutas relativas para el acceso de datos o en los archivos de entrada puede ser definida. Esta variable no sobrescribe la opción *wd* del archivo de máquinas.
PVM_EXPORT	Nombres de las variables que serán exportadas de las tareas padres hacia los hijos a través de *pvm_spawn()*. Se pueden colocar múltiples variables separadas por ":". Si no se coloca la variable, no exporta ninguna variable.
PVM_DEBUGGER	El debugger *script* que usará cuando *pvm_spawn()* se llama con la bandera de PvmTaskDebug. Por defecto es $PVM_ROOT/lib/debugger.
PVM_DPATH	Ruta del *script* de inicio de pvmd, por defecto es $PVM_ROOT/lib/pvmd. Esta variable sobreescribe la opción *dx* del archivo de máquinas.
PVMHOSTFILE	Especifica la ruta del archivo de máquinas que será usado por defecto al iniciar PVM. Esto beneficia en la colocación de la ruta absoluta de este archivo al iniciar la consola o el demonio de PVM.
PVMDLOGMAX	Coloca la longitud máxima del archivo de errores. Por defecto es 1 Mbyte.
PVMTASK	Coloca banderas adicionales para la llamada a la librería *pvm_spawn()*.
PVMBUFSIZE	Coloca el tamaño de los buffers de memoria compartida. Por defecto es 1048576.

El **archivo de máquinas** define la máquina virtual, contiene el nombre de todas las máquinas conectadas. A continuación se coloca un ejemplo de este archivo.

sewa
#La siguiente máquina será agregada dinámicamente después
& estrella
#La siguiente máquina tiene diferente ruta para sus ejecutables
luna ep=/user02/pvm/bin

Si no se desea escribir todas las máquinas que se desean dar de alta cada vez que se ejecuta la consola, se puede utilizar un archivo de máquinas (*hostfile*). Se listan los nombres de máquinas, uno por línea y luego

$ pvm archivo_máquinas

El archivo donde se especifican las máquinas es de extrema importancia, ya que define la arquitectura de nuestra máquina paralela virtual. Se compone de una lista de máquinas, que pueden llevar o no, el símbolo & al principio según sean o no inicializadas al arrancar el monitor, respectivamente.

Las líneas de comentario son aquellas que comienzan con el símbolo #. Además de esto, cada máquina puede llevar asociada una serie de parámetros para determinar su configuración. Los más importantes son:

lo=usuario	Permite identificar un nombre de usuario distinto para la conexión a la máquina indicada. Por defecto, se emplea el nombre del usuario que lanzó el proceso en la máquina actual.
so=clave	Especifica la clave de la máquina remota. No es una buena idea emplear este campo, salvo que la red sea privada. Por defecto intenta una conexión con *rsh*, que sólo funcionará si en la máquina remota está la máquina local como una entrada al archivo *.rhosts* y las RPC están activadas.
dx=ruta	Donde se encuentra el *pvmd3*.
ep=rutas	Conjunto de rutas, separadas por ";", donde se van a encontrar los ejecutables que se van a lanzar en la máquina. Si no se especifica, buscará en el directorio $HOME/pvm3/bin/PVM_ARCH.
sp=valor	Valor de potencia computacional relativa respecto a las otras máquinas. Los rangos posibles están entre 1 y 1000000. El valor por defecto es 1000.
bx=ruta	Localización del depurador. Sólo es activado si también se activa la opción de depuración.
wd=ruta	Directorio de trabajo. Es, donde las tareas lanzadas se van a ejecutar. Por defecto apunta a $HOME.
ip=dirección	Especifica la dirección IP en la que se va a buscar la máquina.

Comprobación de instalación

El programa *pvm* va a corresponder al núcleo de control de la máquina virtual, o, dicho de otra forma, al monitor de la máquina. Se va a comportar como una consola dentro de la máquina virtual; por eso es denominado muchas veces, simplemente como consola. Cuando lo ejecutamos, entramos en el intérprete de comandos de nuestra máquina virtual. Él lanza el *demonio* de la PVM en el caso de que no lo tengamos activo, y además permitirá ejecutar las funciones básicas de control de la PVM.

Al teclear en el *prompt pvm:*

$ pvm

debe aparecer un *prompt* de la consola de PVM, esto significa que PVM está ejecutándose en esa máquina

pvm>

Una vez que PVM inicia, acepta comandos de la entrada estándar. Se puede ver un listado de comandos y sus opciones escribiendo *help* en el *prompt* de la consola. A continuación se mencionan algunos de ellos.

Add máquina	Incorpora la máquina indicada a la máquina paralela virtual.
Delete máquina	Elimina la máquina indicada del conjunto de máquinas asociadas a la máquina paralela virtual. Como es lógico, no podremos eliminar la máquina desde la que se ejecuta el intérprete de comandos.
Conf	Configuración actual de la máquina paralela virtual.
Ps	Listado de procesos de la máquina paralela virtual.
Halt	Apaga la máquina paralela virtual. Esto significa que mata todas las tareas de la PVM, elimina el demonio de forma ordenada y sale del programa *pvm*. Es la forma de salir en condiciones, y mucho mejor que matar el demonio con *kill*.
help	Lista los comandos del programa.
id	Imprime el TID de la consola.
jobs	Genera un listado de los trabajos en ejecución.
kill	Mata un proceso de la PVM.
mstat	Muestra el estado de una máquina de las pertenecientes a la PVM.
pstat	Muestra el estado de un proceso de los pertenecientes a la PVM.
reset	Inicia la máquina. Eso supone matar todos los procesos de la PVM salvo los programas monitores en ejecución, limpiar las colas de mensajes y las tablas internas y pasar a modo de espera todos los servidores.

setenv	Lista todas las variables de entorno del sistema.
sig señal tarea	Manda una señal a una tarea.
spawn	Arranca una aplicación bajo PVM.
trace	Actualiza o visualiza la máscara de eventos rastreados.
alias	Define un alias predefinido, es decir, un atajo para teclear un comando.
unalias	Elimina un alias predefinido.
version	Imprime la versión usada de la PVM.

Cualquiera de estos comandos, o cualquier combinación de ellos puede ser definida en el archivo *.pvmrc*; dicho archivo será leído y ejecutado al arrancar cada monitor.

Los monitores pueden ser arrancados en cualquier momento, en cualquier número, y en cualquier máquina asociada a una máquina paralela virtual determinada.

El monitor tiene una versión gráfica, denominada xpvm, que va a permitir hacer exactamente lo mismo, pero de una forma más visual.

Si se tienen problemas al iniciar PVM o agregar nuevas máquinas a la máquina virtual, verifica el archivo */tmp/pvmd.<uid>* en la máquina donde se está tratando de iniciar. También revisa el archivo local de bitácoras */tmp/pvml.<uid>*"donde estarán los mensajes de error que pueden ayudar a determinar el problema.

Al levantar PVM en el servidor y querer agregar una máquina cliente que no tiene un archivo de equivalencia como es el *host.equiv* o *.rhosts*, marcará el siguiente error.

pvm> add erandi
add erandi
0 successful
HOST DTID
erandi Can't start pvmd

Auto-Diagnosing Failed Hosts...
erandi...
Verifying Local Path to "rsh"...
Rsh found in /usr/bin/rsh - O.K.
Testing Rsh/Rhosts Access to Host "erandi"...

Rsh/Rhosts Access FAILED - "Permission denied."
Make sure host erandi is up and connected to
a network and check its DNS / IP address.
Also verify that sewa is allowed

rsh access on erandi
Add this line to the SHOME/.rhosts on erandi:
sewa tania

Este problema es causado por las restricciones de acceso a máquinas remotas vía *rsh*. Si al ejecutar el siguiente comando: "$ *rsh remote_host 'echo $PVM_ROOT'*", no obtienes el valor correcto de $PVM_ROOT en la máquina remota sin escribir la contraseña, entonces ese es el problema. Necesitas colocar los permisos.

Recordemos que *rsh* copia su entrada estándar en un comando remoto, la salida estándar del comando remoto en su salida estándar y el error estándar del comando remoto en su error estándar.

Para usar *ssh* en lugar de *rsh* en el sistema, puedes optar por alguna de las dos siguientes opciones:

1.-Modificar el archivo *$PVM_ROOT/conf/$PVM_ARCH.def* para cambiarle la ruta absoluta especificada por "RSHCOMMAND" en la parte de ARCHCFLAGS. Reemplaza la ruta de *rsh* con la ruta absoluta de *ssh* y después recompila PVM.

2. Coloca la variable de ambiente "PVM_RSH" apuntando a la ruta absoluta de *ssh*. Esto no requiere una recompilación pero debe ser colocado en cada shell donde se ejecutará PVM.

Una vez que alguna de las dos opciones anteriores se ha hecho, el archivo *$HOME/.rhosts* no es necesario para PVM, por lo que un problema de seguridad ha sido eliminado. Ahora para agregar máquinas mediante *ssh* debes escribir manualmente la contraseña cada vez que se agregue una máquina.

Cuando se tiene un archivo de equivalencia pero la variable PVM_DPATH (variable con la ruta de los demonios en las máquinas clientes) no se encuentra definida, marcará el siguiente error.

pvm>add erandi
0 successful
HOST DTID
erandi Can't start pvmd

Auto-Diagnosing Failed Hosts...
erandi...
Verifying Local Path to "rsh"...
Rsh found in /usr/bin/rsh - O.K.
Testing Rsh/Rhosts Access to Host "erandi"...
Rsh/Rhosts Access is O.K.
Checking O.S. Type (Unix test) on Host "erandi"...

Host erandi is Unix-based.
Checking SPVM_ROOT on Host "erandi"...

The value of the SPVM_ROOT environment
variable on erandi is invalid ("").
Use the absolute path to the pvm3/ directory.

Ejecución de programas en PVM

El primer paso para ejecutar un programa, es compilarlo. Siempre se necesitará la librería *libpvm3.a* o la librería *libfpvm3.a* para los programas elaborados en C y Fortran respectivamente, y la sintaxis es la siguiente:

cc –o myprog myprog.c –ISPVM_ROOT/include
-LSPVM_ROOT/lib/SPVM_ARCH –lpvm3

xlf –o myprog myprog.f –ISPVM_ROOT/include
-LSPVM_ROOT/lib/SPVM_ARCH –lfpvm3 –lpvm3

Para grupos dinámicos, también se necesita agregar la librería libgpvm3.a

cc –o myprog myprog.c –ISPVM_ROOT/include
-LSPVM_ROOT/lib/SPVM_ARCH –lgpvm3 –lpvm3

xlf –o myprog myprog.f –ISPVM_ROOT/include
-LSPVM_ROOT/lib/SPVM_ARCH –lfpvm3 –lgpvm3 –lpvm3

Se debe asegurar que los ejecutables estén en *~/pvm3/bin/$PVM_ARCH*. Para ejecutar la aplicación dentro de PVM, basta con colocar el nombre del programa: *myprog* (en este caso).

Instalación de XPVM

Los requerimientos para instalar son los siguientes:

1. PVM 3.3.0 ó posterior.
2. TCL 7.3 ó posterior.
3. TK 3.6.1 ó posterior.

El proceso de Instalación se puede resumir con los siguientes pasos:

1.Generar la estructura de directorios con el comando *tar*, se generarán los directorios "*trace*" y "*src*"

2. Personalizar el archivo *XPVM/src/Makefile.aimk*
Se necesita colocar el valor de ciertas constantes para XPVM basadas en la versión de PVM instalado en la máquina.

Para PVM 3.3 *PVMVERSION = -DUSE_PVM_33*
Para PVM 3.4. *PVMVERSION = -DUSE_PVM_34*

También será necesario ajustar la definición de TCLLIBDIR, TKLIBDIR hacia la ubicación de las librerías de TCL y TK, así como TCLINCL y TKINCL hacia el lugar donde están los archivos *include* en la máquina, respectivamente. Por ejemplo:

TCLLIBDIR = -LSHOME/tcl/$(PVM_ARCH)
TCLINCL = $HOME/tcl
TKLIBDIR = -LSHOME/tk/$(PVM_ARCH)
TKINCL = $HOME/tk

Se necesita verificar la ubicación de las librerías X11 y los archivos *include* en el sistema para poder colocar el valor correcto a las variables XLIBDIR y XINCL. Normalmente las variables tienen los siguientes valores:

XLIBDIR = -L/usr/lib
XINCL = -I/usr/include/X11

Finalmente se necesitará agregar una librería extra para la definición de SYSLIBS, esto para el correcto uso de las librerías compartidas que usan TCL/TK. Si el sistema usa "*dl* (dynamic linking library)", o "*-lld*", o "*-ldld*", el valor de la variable será:

SYSLIBS = -ldl -lm
SYSLIBS = -lld -lm
SYSLIBS = -ldld -lm

3. Colocar las siguientes variables de ambiente
Se debe colocar la variable XPVM_ROOT apuntando a la ruta absoluta donde se instaló el directorio padre de XPVM. Por ejemplo:

export XPVM_ROOT = $HOME/pvm3/xpvm

Es recomendable usar el archivo "**.stub*" correspondiente al *shell* usado, que se encuentra en *XPVM/src/*.stub* para colocar esta variable.

También se necesita colocar las variables TCL_LIBRARY y TK_LIBRARY apuntando a los directorios que contiene los archivos de inicio (*.tcl) para TCL y TK. Por ejemplo:

export TCL_LIBRARY SHOME/tcl/library
export TK_LIBRARY SHOME/tk/library

4. Una vez que se hayan realizado los cambios antes mencionados, se debe colocar en el directorio de XPVM de mayor nivel ($XPVM_ROOT) y ejecuta el comando "*make*".

Esto creará el archivo ejecutable "*xpvm*" para la arquitectura de la máquina. Estará localizado dentro de "*XPVM/src*", con el mismo nombre que el identificador de arquitectura de PVM.

$XPVM_ROOT/src/$PVM_ARCH/xpvm

Los paquetes TCL y TK están disponibles vía ftp anónimo en ftp.smli.con en el directorio */pub/tcl*.

TCL proporciona una poderosa plataforma para la creación de aplicaciones integrales que reúnen diversas aplicaciones, protocolos, dispositivos. Al trabajar junto con TK, TCL provee la más rápida y poderosa herramienta para crear aplicaciones GUI que corran en PC, Unix. TCL también puede ser usado para una variedad de tareas relacionadas con el Web.

La compilación e instalación de estos paquetes se menciona a continuación:

1. Si se ha compilado TCL/TK alguna vez en un directorio y ahora se desea compilar en el mismo directorio pero para otra plataforma, o si se ha aplicado algún parche, teclea "*make distclean*" para descargar la información computada previamente.

2. Si no hay un *script* llamado "*configure*" en ese directorio es porque se ha trabajado fuera del directorio fuente. En este caso se necesitará usar "*autoconf*" para generar el *script* de configuración. Correrá sin argumentos. Se debe correr en el directorio base y en *dltest*.

3. Teclea *./configure*, Esto ejecuta un *script* creado por GNU para configurar TCL para el sistema y crea el Makefile. Este *script* puede ser modificado para personalizarlo.

4. Teclea *make*, Esto creará una librería llamada "libtcl.a" o "libtcl.so" "libtk.a" o "libtk.so", respectivamente y un intérprete llamado "tclsh", "wish", que permitirá escribir comando de TCL interactivamente o ejecutar *scripts*. Si no termina correctamente la ejecución del *make*, se tendrá que personalizar el *Makefile*.

5. Teclea "make install" para instalar los binarios de TCL/TK y los *scripts* en lugares estándares. Se necesitará permisos de escritura en los directorios de instalación. Los directorios de instalación están determinados por el *script* "configure".

Para este momento, se puede invocar el programa "tclsh" y probar comandos de TCL. Sin embargo, es necesario colocar las variables TCL_LIBRARY y TK_LIBRARY con la ruta absoluta del subdirectorio "library". Nótese que la versión instalada de tclsh, libtcl.a, y libtcl.so tienen el número de versión en su nombre como "tclsh8.2" o "libtcl8.2.so".

En la siguiente figura, se mostrará la apariencia de la versión gráfica de PVM.

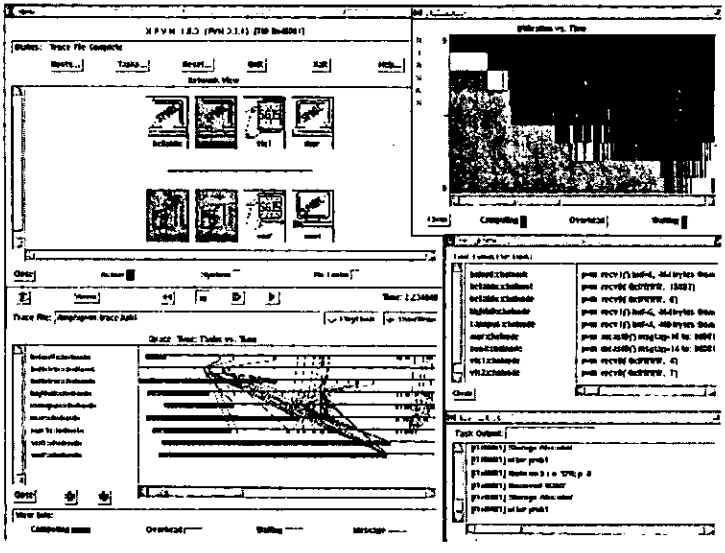

Figura 18 Ambiente gráfico (XPVM)

Instalación de MPI

Si se tiene gunzip, bajar mpich.tar.gz, escribir en la línea de comando lo siguiente, esto descomprimirá y generará la estructura de directorios.

$ gunzip –c mpich.tar.gz | tar xovf –

Si no se tiene gunzip, se puede bajar mpich.tar.Z y usar cualquiera de las siguientes opciones.

$ Zcat mpich.tar.Z | tar xovf – *o* *uncompress mpich.tar.Z*
tar xvf mpich.tar

Se creará un directorio llamado mpich-1.2.0 que contiene varios subdirectorios que incluyen código fuente, algo de documentación, páginas de ayuda, programas de ejemplo. En general, se deben tener los siguientes subdirectorios y archivos:

Makefile.in	Plantilla para el *makefile*, éste será creado cuando se ejecuta el *configure*.
README	Información básica e instrucciones para configurar.
Bin	Directorio donde están los archivos ejecutables como *mpirun* o *mpiman*.
Ccbugs	Directorio para programas que prueban el compilador de C durante la configuración para asegurarse que será capaz de compilar el sistema.
Configure	*Script* que corre para crear el *makefile* en el sistema.
Configure.in	Entrada para autoconfigurar.
Examples	Directorio con ejemplos de programas MPI, tiene otros subdirectorios como *basic*, *test*, *perftest*.
Include	Directorio de librerías de *include*, tanto para el usuario como para el sistema.
Lib	Directorio de librerías para MPI, MPE y las herramientas relacionadas.
Mpid	Código fuente para varios dispositivos que personalizan *mpich* para una máquina en particular, un sistema operativo y su ambiente.
Src	Código Fuente para la parte portable de *mpich*.
Jumpshot	Códigos para los programas de visualización de rendimiento.
Util	Programas y archivos utilitarios.

Después de la creación de directorios, para seleccionar la arquitectura y los dispositivos por defecto, se teclea:

$ ./configure

Como se mencionó anteriormente, la configuración de *mpich* se hace mediante el *script* "configure" que está en el directorio de mayor jerarquía. Este *script* es generado automáticamente por autoconfiguración del archivo *config.in*.

Si se coloca "*$configure –usage*" se tendrá una lista completa de argumentos y sus significados. Todos los argumentos son opcionales.

Por último se realiza la compilación, esto es, mediante el siguiente comando:

$ make > & make.log & (tomará cierto tiempo)

Make, limpiará todo el contenido de los directorios de los archivos objetos previos, compilará las versiones del código fuente, incluyendo Romio y la interfaz de C++, construirá las librerías necesarias y ligará lo necesario.

Comprobación de Funcionamiento

MPI provee un *script* llamado *tstmachines* que verifica el funcionamiento correcto de todos los nodos descritos en el archivo *machines.LINUX*. Este *script* realiza pruebas completas de ejecución remota con *rsh* en todos los nodos, además de compilar y ejecutar un programa de prueba para verificar que todos los ejecutables estén accesibles.

$ /usr/share/mpi/tstmachines

El *script* realiza su prueba nodo a nodo, y para los nodos donde el funcionamiento es correcto, se tiene el siguiente despliegue:

Trying true on sewa ...
Trying ls on sewa ...
Trying user program on sewa ...

Ejecución de programas MPI

El estándar MPI no define los mecanismos mediante los cuales un programa MPI debe ser ejecutado, por lo que éstos son dependientes de la implementación. La implementación que utilizaremos en las estaciones de trabajo es *mpich*.

Para compilar un programa en esta implementación se utilizan los comandos *mpicc* y *mpif77*. Para ejecutar un programa se utiliza el comando llamado *mpirun*. La opción más común es *–np*, la cual especifica el número de procesos que serán ejecutados. Las máquinas en las cuales se ejecutarán estos procesos están especificadas en el archivo *$MPICH/util/machines/machines.XXX*, donde MPICH es el directorio en el cual está instalado el *mpich* y XXX especifica la arquitectura a utilizar. Es posible ejecutar programas en diferentes arquitecturas.

MPI no especifica cómo se arrancarán los procesos MPI. Las implementaciones se encargarán de definir los mecanismos de arranque de procesos; para tales fines, algunas versiones coinciden en el uso de un programa especial denominado *mpirun*. Dicho programa no es parte del estándar MPI.

La opción *–t* muestra los comandos que *mpirun* ejecuta y puede ser usada para determinar cómo *mpirun* arranca los programas en un sistema determinado.

$ mpirun –np 2 –machinefile máquinas holamundo

El parámetro *np* le indica al programa *mpirun* que debe crear dos instancias de "holamundo"; mientras que *machinefile* indica el nombre del archivo (máquinas) que contiene la descripción de las máquinas que serán utilizadas para correr los procesos. Adicionalmente, *mpirun* tiene un conjunto de opciones que permiten al usuario ordenar la ejecución de un programa bajo determinadas condiciones. Estas opciones pueden verse con el comando:

$ man mpirun o mpirun -help

Compilando y enlazando programas MPI

Para proyectos grandes es mejor un *Makefile* estándar. Algunas implementaciones, como MPICH y LAM, proveen los comandos *mpicc* y mpif77 para compilar programas en C y Fortran respectivamente.

Para compilar, usualmente, se usan los comandos *mpicc* y *mpif77* cuyo funcionamiento y parámetros son muy similares a los de los compiladores *cc* y *f77*.

$ mpicc –o holamundo holamundo.c

Además de los comandos de compilación, MPICH incluye una plantilla de ejemplo para la construcción de archivos *Makefile* denominada *Makefile.in*. Dicho archivo genera, a través de traducción el archivo *Makefile*. Para lograr esta traducción debe utilizarse el programa (*script*) *mpireconfig* que construirá el archivo *Makefile* para un sistema particular. Esto permite usar el mismo *Makefile* para una red de estaciones de trabajo o para una computadora masivamente paralela aunque utilicen distintos compiladores, bibliotecas y opciones de enlace

$ mpireconfig Makefile

5 ANÁLISIS Y DISEÑO DEL SISTEMA DE PROCESAMIENTO DISTRIBUIDO

5.1 PLANTEAMIENTO DEL PROBLEMA

Al conocer las tendencias actuales en los sistemas y el avance tecnológico que se ha tendido en los últimos años en este ámbito, se sabe que la creación de poderosos microprocesadores y el gran desarrollo en las redes locales de alta velocidad tienden hacia el procesamiento paralelo o distribuido de los sistemas.

Apoyándose en estas tendencias, y en las exigencias de mejoras en el ámbito costo-eficiencia de las máquinas, se observa que la repartición de carga de trabajo en los procesadores es un área importante de investigación.

Esta repartición o división de tareas, se puede realizar de manera paralela o distribuida. La diferencia principal es la localización de los procesadores, así como la forma en que el usuario verá al sistema. En un sistema paralelo es necesario tener más de un procesador en una misma máquina mientras que en un sistema distribuido los procesadores se encuentran en diferentes máquinas conectadas mediante una red; en ambos casos se busca rapidez y eficiencia en la entrega de resultados.

Definiremos entonces, para nuestro caso, que un sistema de procesamiento distribuido es un conjunto de computadoras independientes, de arquitectura de bajo costo, conectadas en red y que aparentan ser una sola máquina. Es necesario mencionar como característica primordial, el aprovechamiento de recursos de hardware que pueden ser recursos inutilizados en determinado momento.

En la Facultad de Ingeniería, existen muchas áreas donde se dá servicios a usuarios (alumnos, profesores, investigadores, etc.); esto es, una serie de computadoras conectadas en red que realizan trabajos de manera particular hacia un grupo de personas y que comparten algunos recursos de hardware o software. Es necesario mencionar que muchas veces se realizan las tareas de los usuarios de manera ineficiente, debido a que no se cuenta con un planificador de procesos a través de la red; es por eso que se ha elegido hacer el análisis y diseño de un sistema de procesamiento distribuido, buscando obtener las ventajas antes mencionadas.

Como se mencionó anteriormente, al construir un sistema de procesamiento distribuido todos estos recursos que no son utilizados de una manera adecuada, lo serán; y la entrega de resultados se realizará rápidamente.

La construcción de un sistema de procesamiento distribuido es muy factible ya que es un sistema de arquitectura de bajo costo, esto quiere decir que no es necesario realizar inversiones de dinero, porque se construye a partir de las máquinas que se tienen. Solo es necesario tener máquinas con recursos de hardware y software mínimos y que no sean utilizadas de manera adecuada, para conectarlas en red y buscar que el servidor de procesos distribuya el procesamiento de tareas eficientemente.

5.2 ANÁLISIS DEL SISTEMA DE PROCESAMIENTO DISTRIBUIDO

A continuación se mencionará paralelamente la justificación teórica usada en el análisis del sistema, esto, con el objetivo de tenerla presente.

5.2.1 Unified Modeling Language

El lenguaje unificado de modelado o UML (*Unified Modeling Language)* es el sucesor de la oleada de métodos de análisis y diseño orientados a objetos, que surgió a finales de la década de los 80's y principios de los 90's. El UML unifica, sobre todo, los métodos de Booch, Rumbaugh (OMT) y Jacobson, pero su alcance llegará a ser mucho más amplio.

Decimos pues, que el UML es un lenguaje de modelado, y no un método. La mayor parte de los métodos consisten, al menos en principio, de un lenguaje y un proceso para modelar. El lenguaje de modelado es la notación (principalmente gráfica) de que se valen los métodos para expresar los diseños. El proceso es la orientación que nos da el lineamiento de los pasos a seguir para hacer el diseño

UML es usado para modelar un amplio rango sistemas como son los sistemas distribuidos, así como en las diferentes fases del desarrollo de un sistema, desde la especificación de requerimientos hasta las pruebas del sistema.

Existen 5 fases de desarrollo de un Sistema de Cómputo y se irán explicando al mismo tiempo del modelado del sistema distribuido.

Vistas

El modelado de un sistema suele ser una tarea difícil y elaborada, y generalmente no es posible representar en una sola imagen toda la funcionalidad y requerimientos del sistema, por lo tanto se recurren a vistas, donde cada vista representa una proyección de la descripción completa del sistema, mostrando un aspecto particular del mismo.

Cada vista se describe en un número de diagramas que contienen información que enfatiza un aspecto particular del sistema. Un diagrama contiene símbolos gráficos que representan los elementos del modelo del sistema.

Diagramas

Los diagramas son las gráficas que muestran elementos de modelado organizados, para ilustrar un aspecto particular del sistema. Un modelo de sistema típicamente tiene varios diagramas de cada tipo. Un diagrama es parte de una vista específica; y cuando se dibuja, se ubica usualmente en una vista. Algunos tipos de diagramas pueden ser parte de varias vistas, dependiendo de los contenidos del diagrama.

Elementos de modelado

Los conceptos usados en los diagramas son llamados elementos de modelado. Un elemento de modelado es definido con semántica, una definición formal del elemento, o el significado exacto de lo que representa en enunciados no ambiguos. Un elemento de modelado tiene también su correspondiente elemento de vista, que es la representación gráfica del elemento o el símbolo gráfico usado para representar al elemento en diagramas. Un elemento puede existir en varios tipos diferentes de diagramas, pero bajo ciertas reglas. Ejemplos de elementos de modelado son:

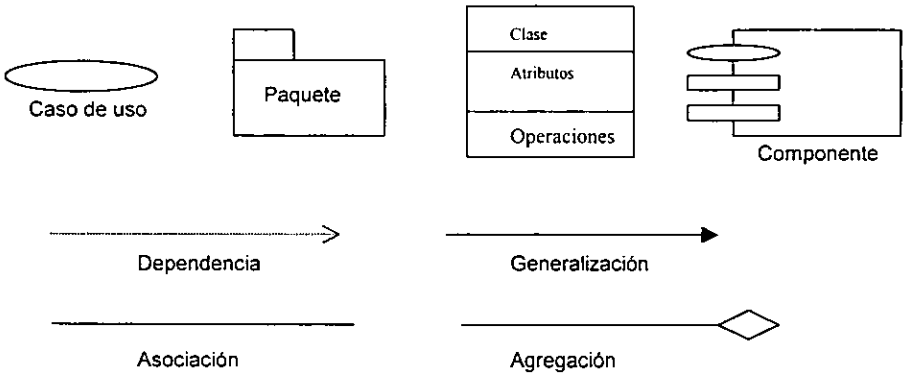

Figura 19 Elementos de modelado

5.2.2 Análisis de requerimientos

UML tiene casos de uso para capturar los requerimientos del cliente. A través del modelado con casos de uso, los actores externos que tienen interés en el sistema, son modelados junto con la funcionalidad que ellos (los casos de uso) requieren del sistema.

Los actores y casos de uso son modelados con relaciones y tienen asociaciones de comunicación con otros, o son separados en jerarquías. Los actores y casos de uso son descritos en un diagrama UML de casos de uso.

Cada caso de uso se describe en texto, y éste, especifica los requerimientos del cliente, es decir, lo que él espera del sistema sin importar como se implante. Un análisis de requerimientos puede también ser realizado con procesos de negocio, no solamente para sistemas de software.

Vista de casos de uso

Describe la funcionalidad que se espera del sistema, según la percepción de actores externos. Un actor interactúa con el sistema y puede ser un usuario u otro sistema. La vista de casos de uso es para clientes, diseñadores, desarrolladores, y probadores. Se describe a través de diagramas de casos de uso y ocasionalmente diagramas de actividad.

La vista de casos de uso es central, dado que su contenido lleva al desarrollo de otras vistas. El objetivo final del sistema es proporcionar la funcionalidad descrita en dichas vistas, por lo tanto esta vista afecta al resto, es usada también para validar y verificar el sistema.

Diagramas de casos de uso

Un diagrama de caso de uso muestra un número de actores externos y su conexión a los casos de uso del sistema. Un caso de uso es una descripción de una funcionalidad (un uso específico del sistema) del sistema.

Los elementos de un diagrama de casos de uso se mencionan a continuación:

Sistema

Como parte del modelado de casos de uso, los límites del sistema desarrollado están definidos. Definir las fronteras y la responsabilidad total del sistema no siempre es fácil, porque no siempre es obvio cuales tareas deben ser automatizadas por el sistema y cuáles deben ser llevadas a cabo manualmente o por otros sistemas. Una mejor idea es identificar la funcionalidad básica y concentrarse en definir un sistema estable y bien planeado en su arquitectura de modo que en el futuro puedan agregarse nuevos módulos fácilmente.

Un sistema es descrito en un diagrama de casos de uso con una caja; el nombre del sistema aparece sobre o dentro de la caja. La caja contiene también para los casos de uso del sistema, como se mostró en la figura anterior.

Actores

Un actor es alguien o algo que interactua con el sistema; es quien o que usa el sistema. Con "interactuar con el sistema" decimos que el actor envía o recibe mensajes del sistema, o intercambia información con éste. En concreto, los actores ejecutan los casos de uso.

Un actor es un tipo (una clase), no una instancia. El actor representa un role, no un usuario individual del sistema. De hecho, una misma persona puede ser a la vez varios actores, dependiendo del role en el sistema. Un actor tiene un nombre, y el nombre debe reflejar el role del actor. El nombre no debe reflejar una instancia especifica del actor, ni la funcionalidad del actor.

Un caso de uso es siempre iniciado por un actor que envía un mensaje. Esto es llamado estímulo. Cuando un caso de uso es realizado, el caso de uso puede enviar mensaje a uno o más actores, los cuales pueden ser diferentes al actor que inició el caso de uso.

Teniendo en mente que los actores en UML son clases con el estereotipo <<actor>>, que el nombre de la clase es el nombre del actor (reflejando el role del actor) un actor puede tener asociados atributos y comportamientos, así como documentación propia describiendo al actor. El icono característico de un actor es un "*stickman*", con el nombre del actor bajo la figura.

Casos de uso

Un caso de uso representa una funcionalidad completa tal cual es percibida por el actor. Un caso de uso en UML es definido como un conjunto de secuencias de acciones que desarrolla un sistema que deriva en un resultado observable de algún valor para un actor en particular. Las acciones pueden involucrar comunicación con un número de actores (usuarios y otros sistemas) así como desarrollar cálculos y trabajo dentro del sistema. Las características de un caso de uso son:

- Los casos de uso se ligan a los actores a través de comunicaciones, generalmente bidireccionales, de modo que puedan intercambiar información en ambas direcciones.

- Un caso de uso es una clase y no una instancia. Describen la funcionalidad como un todo, incluyendo alternativas posibles, errores y excepciones que pueden darse durante el caso de uso. Una instancia de un caso de uso se conoce como escenario, y representa un uso específico del sistema.

Un caso de uso es representado en UML por una elipse, conteniendo el nombre del caso de uso o con el nombre del caso de uso debajo de la elipse. Un caso de uso normalmente se coloca dentro de las fronteras del sistema y puede ser ligado a un actor mediante asociaciones que muestren como interactúan el actor y el caso de uso.

Existen tres tipos de relaciones entre casos de uso: Extiende (extends), usa (uses) y agrupa (grouping). Las relaciones extiende y usa, son diferentes tipos de

herencia, Agrupa es un modo de colocar casos de uso relacionados en un paquete. Las definiciones de cada relación son las siguientes:

- **Relación "Extiende":** Una relación de generalización donde un caso de uso extiende de otro, agregando acciones a un caso de uso general. El caso de uso que extiende puede incluir comportamiento del caso de uso que es extendido, dependiendo de las condiciones de la extensión.

- **Relación " Usa":** Una relación de generalización donde cada caso de uso utiliza otro caso de uso, indicando que es parte del caso de uso especializado, el comportamiento del caso de uso general deberá ser incluido por completo. Si el caso de uso siendo usado, nunca se usa explícitamente, se denomina caso de uso abstracto.

- **Relación "Agrupa":** Cuando un número de casos de uso maneja una funcionalidad similar o están de algún modo relacionados entre sí, pueden ser agrupados en un paquete UML.

5.2.3 Análisis de requerimientos en el sistema distribuido

Para el sistema de procesamiento distribuido a desarrollar, se ha hecho el análisis de requerimientos que se presenta a continuación, basándose en la identificación de las necesidades de éste.

- Creación del ambiente distribuido.

- Interacción del usuario con el sistema a través de las peticiones o trabajos a solicitar, de forma amigable.

- Manejo de la conexión o desconexión de nodos a la máquina virtual, así como la identificación de valores iniciales de los nodos.

- Recepción de las peticiones de los usuarios.

- Identificación del tipo de trabajos a realizar y manejo de éstos.

- Asignación del procesador, basado en sus características y en la carga de trabajo en ese momento.

- Manejo de colas de procesos.

- Creación y destrucción de procesos alternos para la realización del trabajo asignado al procesador.

- Ejecución de las tareas necesarias en el procesador asignado para la obtención de resultados.
- Envío de resultados finales al nodo solicitado.

Actores

Los actores que se han identificado en el sistema son:

- Nodos, llámese cada una de las máquinas conectadas en la máquina virtual.
- Petición, los trabajos solicitados por los usuarios para su ejecución.
- Procesos, cada una de las diferentes tares que se deben realizar para llevar a cabo la petición solicitada.
- Servidor, aquel que iniciará el ambiente distribuido.
- Usuario, es el encargado de hacer uso del sistema distribuido.

Diagramas de casos de uso

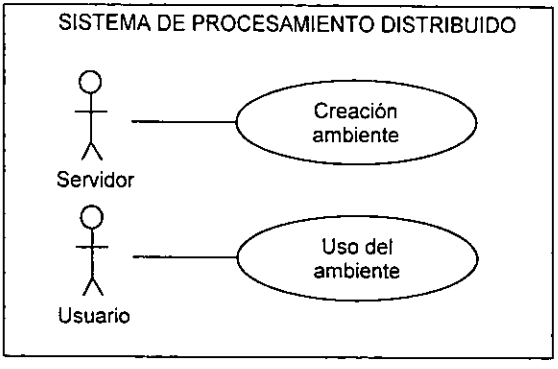

Figura 20 Diagrama de caso de uso principal

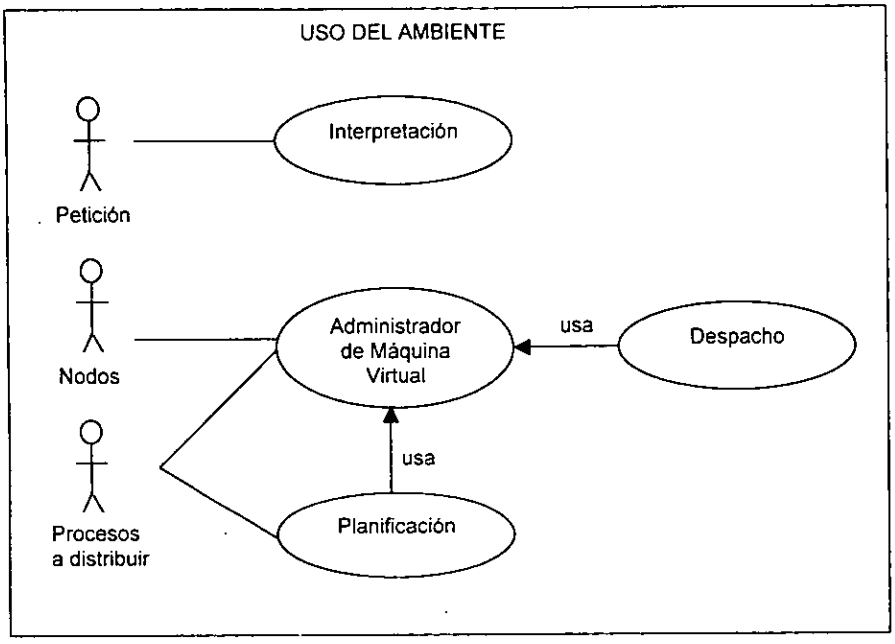

Figura 21 Diagramas de caso de uso

Descripción de casos de uso

En el sistema de procesamiento distribuido se han identificado dos casos de uso principales, éstos son, la creación del ambiente y el uso del sistema.

Al mismo tiempo, el uso del sistema se ha dividido en cuatro casos de uso que se explicarán en el siguiente apartado, además de que servirán como base para la elaboración de los siguientes diagramas y el desarrollo del sistema.

Creación del ambiente

En la creación del ambiente tenemos como actor primario al servidor, que será el encargado de iniciar el ambiente distribuido.

Como iniciación o creación del ambiente distribuido se hace referencia al levantamiento de la máquina virtual, teniendo como nodo inicial, solamente al servidor. Así como también, el levantamiento de servicios (demonios) necesarios.

Uso del ambiente

En el uso del ambiente, el actor primario será el usuario. A continuación se ha hecho una explosión de este caso de uso para poder entender a detalle, cuáles son las tareas por las que pasará una petición hecha por el usuario.

Interpretación

Se tiene como actor primario a la petición o trabajo hecha por el usuario, ésta será la entrada para la identificación de tareas y reconocimiento de valores iniciales para el siguiente paso en el sistema.

Recordemos que se ha planteado un sistema de tipo "Cliente-Servidor", por lo que a partir de este momento, se manejará el siguiente concepto: "Las peticiones son hechas por nodos clientes y recibidas por el servidor, donde residirá el sistema de procesamiento distribuido". Además es necesario mencionar que será una comunicación bidireccional porque así como envía peticiones, también recibirán los resultados de éstas.

Podemos resumir en los siguientes puntos.

1. El nodo cliente hace la petición al sistema distribuido.
2. El shell o intérprete de comandos del sistema distribuido, reconocerá la petición y sabrá qué hacer con ella. Tiene dos opciones; una, continuar con el proceso de distribución o, mandar mensaje de error al cliente.

Administrador de Máquina virtual

La máquina virtual estará encargada de las tres siguientes tareas: construcción de máquina virtual, asignación de procesos para el despacho y entrega de resultados al nodo cliente que haya enviado la petición. En la construcción de la máquina virtual se tendrá como actor primario a los nodos, mientras que en las dos últimas los actores serán los procesos a distribuir.

La construcción de la máquina virtual se refiere a dos aspectos básicos: la conexión y desconexión de nodos clientes. La conexión se referirá a la unión de nuevos nodos, así como sus características. Es necesario mencionar que en un principio se parte con una máquina virtual compuesta sólo por el servidor.

La desconexión hace referencia a la culminación de servicios proporcionados por un nodo cliente.

Resumiendo:

1. Considerar el tipo de petición del nodo cliente.
2. Si el nodo cliente envió mensaje de unión, se debe reconocer los recursos con que cuenta este nodo; esto para la posterior adecuada asignación de procesos.
3. Si es un mensaje de desconexión, hacer las actividades necesarias para dejar de ser considerado.

La asignación de procesos se basa en el monitoreo del estado de los procesadores, con esto se quiere decir que debe conocer los recursos con los que cuentan y la carga de trabajo que tenga; con el objetivo de buscar realizar la mejor asignación.

1. Considerar el estado de los procesadores.

2. Hacer la asignación al procesador adecuado.

También tendrá que recibir los resultados de la ejecución de los procesos en el despachador y enviarlos al nodo cliente.

1. Recibir resultados.
2. Reunir la información necesaria para enviar.
3. Enviar resultados al nodo cliente que realizó la petición.

Planificación

La planificación tendrá como actor a los procesos o tareas a distribuir. Debe seleccionar el siguiente proceso a distribuir usando las colas de procesos; además de agregar, en caso de ser necesario, los valores o parámetros necesarios para su asignación y despacho.

1. Seleccionar en la cola de procesos a distribuir, la siguiente tarea que espera ser asignada.
2. Agregar los valores o parámetros necesarios.

Despacho

El despacho de las tareas se llevará a cabo en los procesadores de cada nodo cliente y tendrá como entrada a los procesos distribuidos.

Es necesario mencionar que se tendrá una interacción permanente con la máquina virtual de manera bidireccional, ya que ésta es la que le envía el proceso asignado, así como también será quién reciba los resultados proporcionados por el despachador.

1. Una vez asignados los procesos distribuidos a los procesadores; éstos (procesadores) reciben los valores y/o datos necesarios para su ejecución.
2. Se ejecuta el proceso obteniendo los resultados finales.
3. Los resultados que son generados por el procesador son enviados a la máquina virtual.

5.2.4 Análisis

La fase de análisis se relaciona con las abstracciones principales (clases y objetos), y mecanismos que están presentes en el dominio del problema. En esta etapa se identifican las clases que modelan lo anterior y se establecen las relaciones entre ellas, describiéndolas finalmente en un diagrama de clases UML. La colaboración entre clases para desarrollar los casos de uso se describen también con modelos dinámicos en UML. En el análisis sólo se modelan las clases en el dominio del problema (conceptos del mundo real), no se incluyen clases técnicas que definan detalles y soluciones en el sistema de software, tales como clases de interfaz de usuario, bases de datos, comunicaciones, etc.

Vista Lógica

La vista lógica describe cómo se proporciona la funcionalidad al sistema. Esta vista está orientada principalmente para los diseñadores y desarrolladores. En contraste con la vista de casos de uso, la vista lógica se involucra dentro del sistema, describiendo tanto la estructura estática (clases, objetos y relaciones) y la colaboración dinámica que ocurre cuando los objetos se envían mensajes entre sí para desempeñar determinada función.

Se definen también propiedades tales como persistencia y concurrencia, así como interfaces y estructura interna de las clases. La estructura estática es descrita en diagramas de clases y de objetos. El modelado dinámico se describe en diagramas de estado, secuencia, colaboración y actividad.

Diagrama de clases

Un diagrama de clases es un tipo de modelo, específicamente, un modelo estático. Un diagrama estático describe la vista estática de un sistema en términos de clases y relaciones entre ellas, mostrando atributos y comportamientos de dichas clases. Uno de los propósitos del diagrama de clases, es definir la base para otros diagramas donde se muestran otros aspectos del sistema, tal como el estado de los objetos, o las colaboraciones entre ellos.

Para crear un diagrama de clases, las clases deben identificarse y describirse; y cuando ya existe un determinado número de clases, éstas pueden relacionarse entre sí. Una clase se representa con una caja rectangular, dividida en tres compartimentos, el del nombre, el de los atributos y el de las operaciones. La sintaxis usada para los compartimentos es independiente de los lenguajes de programación, aunque también pueden usarse.

Un sistema típicamente tiene un número de diagramas de clases – no todas las clases se insertan en un solo diagrama – y una clase puede participar en varios diagramas de clase.

Las clases son los bloques de construcción más importantes en cualquier sistema orientado a objetos. Una clase es una descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones y significado. Una clase implementa una o más interfaces. Podemos decir también que una clase es una abstracción de las cosas que son parte del vocabulario tradicional. Una clase no es un objeto individual, sino que representa un conjunto de objetos.

Se usan las clases para capturar el lenguaje del sistema que se esta desarrollando. Las clases pueden incluir abstracciones que son parte del dominio del problema así como las clases que realizan la implementación. Se usan clases para representar elementos de software, hardware, e incluso cosas puramente conceptuales.

Un conjunto de clases bien estructurado posee límites precisos y forma parte de una distribución balanceada de responsabilidades a través del sistema.

El UML proporciona una representación gráfica de una clase, la cual permite visualizar una abstracción aparte de cualquier lenguaje de programación específico y de algún modo permite enfatizar lo más importante en una abstracción: su nombre, atributos y operaciones.

Nombres

Cada clase debe tener un nombre distintivo, lo menos ambiguo posible, el cual es una cadena textual que sola, se conoce como nombre simple, un nombre con ruta, es precedido por el nombre del paquete en el cual vive la clase. Una clase puede dibujarse como un rectángulo incluyendo solo el nombre. El nombre no debería contener sufijos, ni prefijos.

El nombre de una clase en la práctica, suele ser un sustantivo o una frase corta, tomados del vocabulario del sistema a modelar. Típicamente se capitaliza la primera letra de cada palabra en el nombre de la clase.

Atributos

Los atributos son las propiedades de una clase que describen un rango de valores que las instancias de dicha clase pueden contener. Una clase puede contener cualquier número de atributos, o ningún atributo en absoluto. Los atributos capturan la información que describe e identifica una instancia específica de la clase. Sin embargo, sólo deben incluirse aquellos atributos que sean de interés para el sistema que se está modelando, por lo tanto, el propósito del sistema influye directamente sobre la decisión de que atributos incluir. Los atributos deberán especificar también una visibilidad, que definirá el nivel de acceso de otras clases diferentes a la clase que los contiene. En un momento dado, un objeto de una clase tendrá valores específicos para cada atributo de su clase. Gráficamente los atributos se listan en un compartimento debajo del nombre de la clase, los atributos pueden dibujarse mostrando solo el nombre.

Un atributo generalmente se identifica con un sustantivo o una frase corta que representa alguna propiedad de la clase que lo incluye. Típicamente se capitalizan las primeras letras de cada palabra del nombre del atributo, excepto la primera letra.

Es posible especificar un atributo estableciendo el nombre de la clase y posiblemente un valor inicial.

Operaciones

Una operación es la implementación de un servicio que puede ser solicitado desde un objeto de la clase para llevar a cabo un comportamiento. Una operación es una abstracción de algo que puede hacerse con el objeto y que es compartido por todos los objetos de la clase. Una clase puede tener cualquier número de operaciones o ninguna. Gráficamente, las operaciones son listadas en

un compartimento debajo de los atributos. En la práctica, los nombres de las operaciones son verbos o frases verbales cortas. Típicamente, se capitalizan las primeras letras de cada palabra, excepto la primera.

Los atributos y operaciones pueden no listarse todos, indicando con tres puntos suspensivos que hay más de los que se muestran. También, por motivos de organización, se pueden incluir estereotipos clasificadores para definir categorías.

Relaciones

En el modelado orientado a objetos, existen tres tipos de relaciones que son especialmente importantes:

- **Dependencias**, que representan el uso de relaciones entre clases (incluyendo refinamiento, rastreo y ligado) Se trata exactamente de dependencia, de tal modo que una clase sin la otra no puede llevar a cabo una función específica.
- **Generalización**, las cuales ligan clases generalizadas a sus especializaciones; estas relaciones se conocen como subclase/superclase o hijo/padre.

- **Asociaciones,** que representan relaciones estructurales entre objetos, tales como composición.

Gráficamente, una relación es dibujada como una ruta, con diferentes tipos de líneas usadas para distinguir los tipos de relaciones.

Una dependencia es una conexión que establece que un cambio en la especificación de una cosa, puede afectar a otra que la usa, pero no necesariamente a la inversa. Gráficamente, una relación es dibujada como una línea punteada dirigida a la cosa de la que se depende. Se usan las dependencias cuando se quiere mostrar que una cosa usa a otra. A menudo las dependencias se usan en el contexto de clases para especificar que una clase usa a otra clase como argumento, de tal modo que si la clase usada como argumento cambia, puede afectar a la clase que la usa. Pueden establecerse dependencias con otro tipo de cosas, como notas y paquetes. Una dependencia puede tener un nombre, pero rara vez es necesario, a menos que se requiera distinguir entre todas las existentes en un modelo amplio. Generalmente se usan estereotipo para distinguirlas entre sí.

Una generalización es una relación entre algo general (llamado el padre o la superclase) y un tipo más específico de cosa (llamado también el hijo, o la subclase). Esta relación se conoce también con el nombre " es-un-tipo-de (is-a-kind-of)". La generalización indica que el hijo puede ser usado en cualquier lugar en el que aparezca el padre, pero no viceversa. Un hijo hereda las propiedades de los padres de sus padres, especialmente sus atributos y sus operaciones. Adicionalmente, el hijo puede tener sus propios atributos y operaciones. Una

operación de un hijo que tenga la misma firma que en el padre, borra la operación del padre; esto es conocido como polimorfismo. Gráficamente, la generalización se representa como una línea sólida dirigida con una gran punta de flecha sin rellenar, apuntando al padre. Una clase sin padres, y con uno o más hijos, es una clase base. Una clase sin hijos, y un padre o más, es una clase hoja o final. Un padre, implica herencia simple, más de un padre implica herencia múltiple.

Una asociación es una relación estructural que especifica que objetos de una cosa están conectados a objetos de otra. Permite la navegación de un objeto a otro, en ambos sentidos. Incluso pueden haber asociaciones recursivas a la misma clase. Si la asociación es entre dos clases exactamente, se trata de una asociación binaria, si hay más clases, es una relación n-aria. Gráficamente una asociación se representa como una línea sólida conectando a la misma o diferente clase. Una asociación puede enriquecer su representación con elementos como:

- ***Nombre***, indicando con una flecha sólida la dirección en que debe leerse la asociación.

- ***Role,*** a las clases que participan en la asociación, para definir en que forma se da tal asociación.

- ***Multiplicidad***: Se indica cuantos objetos pueden participar en la asociación por parte de cada clase. Exactamente uno (1), cero o uno, (0..1), muchos (0..*), uno o muchos(1..*) o un numero exacto 7,9 .

- ***Agregación***: Muchas veces, cuando se necesita modelar una relación de (parte/todo), en la cual una cosa representa algo grande (todo) que consiste en parte más pequeñas (partes), se utiliza una relación de agregación, que puede enunciarse como una relación "tiene-un (has-a)". En realidad es sólo un tipo especial de asociación y se representa con un diamante no sólido en el extremo final de la asociación, del lado del todo.

Vista de Concurrencia

La vista de concurrencia trata con la división del sistema en procesos y procesadores. Este aspecto, que no es propiedad funcional del sistema, permite un uso eficiente de recursos, ejecución paralela, y el manejo de eventos asíncronos del ambiente. Además dividiendo el sistema hilos de control concurrentes, la vista debe tratar también con la comunicación y sincronización de dichos hilos.

La vista de concurrencia es para desarrolladores e integradores del sistema, y consiste en diagramas dinámicos (estado, secuencia, colaboración, y actividad).

Diagrama de secuencia

Un diagrama de secuencia muestra una colaboración dinámica entre un número de objetos. El aspecto importante de este diagrama es mostrar una secuencia de mensajes enviados entre los objetos. Muestra también una interacción entre objetos, algo que sucede en un punto específico de ejecución del sistema. El diagrama consiste en un número de objetos mostrados con líneas verticales. El tiempo transcurre hacia abajo en el diagrama, y este muestra el intercambio de mensajes entre los objetos conforme pasa el tiempo en la secuencia o la función. Los mensajes se muestran como líneas con flechas de mensaje entre las líneas verticales de objetos. Las especificaciones de tiempo y otros comentarios, se agregan en un script al margen del diagrama.

5.2.5 Análisis en el sistema distribuido

Como se mencionó en la justificación teórica, en esta parte se hace un análisis del sistema, basándose en la identificación de las clases.

Clases

Las clases identificadas en el sistema de procesamiento distribuido son:

- Petición, clase referente a las solicitudes hechas por los nodos clientes.
- Intérprete, clase encargada de clasificar cada una de las solicitudes hechas por los nodos clientes.
- Procesos a distribuir, clase referente a cada uno de los procesos en que fueron divididos las solicitudes hechas.
- Despacho, clase donde se manejará la ejecución de las tareas necesarias para la obtención de resultados.
- Planificador, clase que hace referencia al encargado de controlar los procesos a distribuir.
- Máquina virtual, clase referente al manejo de nodos.
- Nodos; podemos interpretar como una subclase de la máquina virtual.

Diagrama de clase

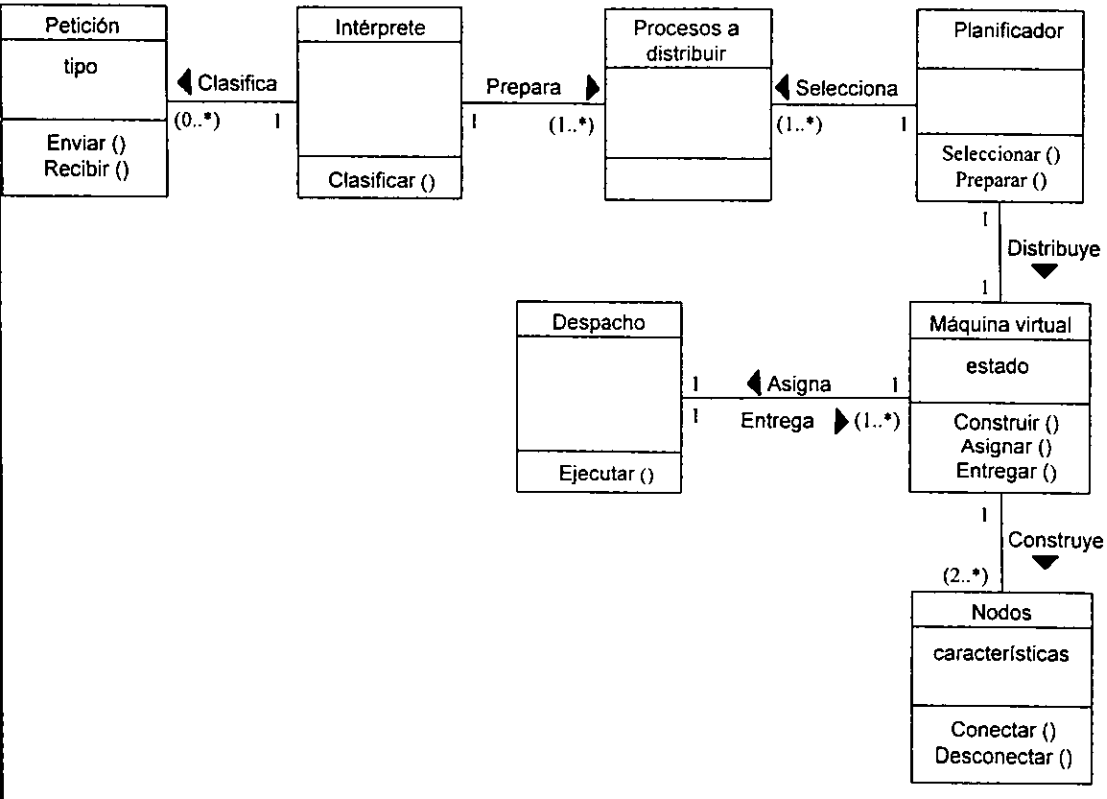

Figura 22 Diagrama de clases

En el diagrama de clases del sistema de procesamiento distribuido se ha dibujado cada una de las clases identificadas, además de las propiedades de los objetos que forman a dicha clase. Por ejemplo, la clase petición tiene como propiedad al tipo de petición, esta identificación de propiedades es básica para que el intérprete (clase con la que se relaciona) pueda realizar la clasificación que éste debe hacer.

En este diagrama también podemos ver los métodos o funciones de cada clase. Por ejemplo, la máquina virtual tiene como funciones el construir la máquina virtual, el asignar el proceso al despachador y entregar los resultados al nodo cliente.

Otra cosa que podemos obtener de este diagrama es la relación entre cada una de las clases. Por ejemplo, la clase planificador seleccionará 1 o muchos de los procesos a distribuir.

Diagramas de secuencia

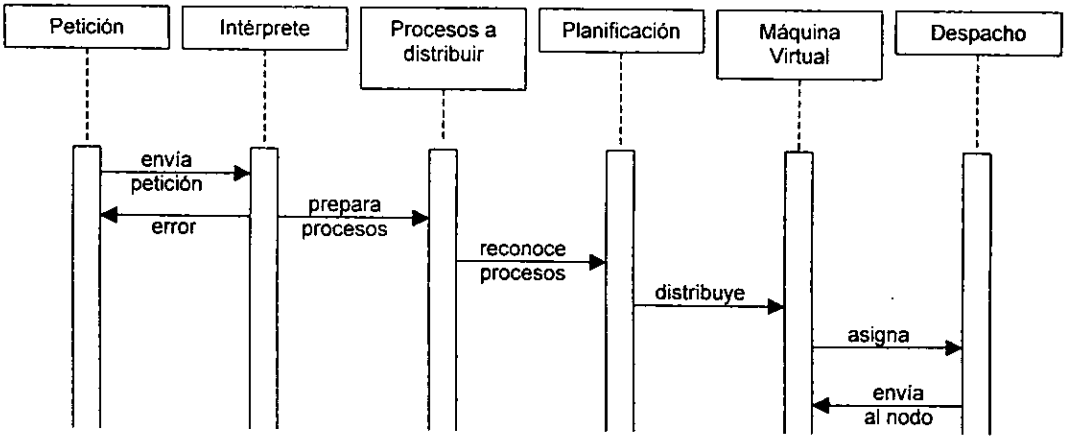

Figura 23 Diagrama de secuencia

En el diagrama de secuencia del sistema de procesamiento distribuido se busca reflejar, a través del tiempo, la comunicación que existirá entre los objetos principales del sistema.

5.3 DISEÑO DEL SISTEMA DE PROCESAMIENTO DISTRIBUIDO

Al desarrollar un sistema, se sabe que este desarrollo está dividido en una serie de pasos a seguir para llegar a la culminación de éste. El primero de ellos, es el análisis, que en este momento ya ha quedado asentado, ahora comenzaremos la fase de diseño del sistema de procesamiento distribuido.

5.3.1 Transición del análisis al diseño

Es necesario mencionar que la serie de pasos a seguir en el desarrollo de un sistema no es excluyente uno del otro, esto es, no se debe caer en el error de considerarlos como unidades separadas e independientes, cada uno de ellos esta relacionado y se debe trabajar con ellos de manera conjunta, es por eso que el diseño que se presenta a continuación está ligado al análisis hecho previamente.

En esta parte se hace una revisión de los diagramas hechos en el análisis y se pueden realizar ajustes con el objetivo de que las representaciones gráficas tengan un mejor entendimiento o percepción del sistema. Es una parte básica. Ya que aquí se hace una clara división de los módulos a programar.

5.3.2 Diseño

En la fase de diseño, el resultado del análisis se expande a una solución técnica. Se agregan nuevas clases para proporcionar la infraestructura técnica: la interfaz de usuario, manejo de bases de datos, comunicaciones, y otros. Las clases de dominio del problema del análisis se agregan a esta infraestructura técnica, haciendo posible cambiar el dominio del problema y la infraestructura. El diseño se convierte en una especificación detallada para la fase de construcción.

Vista de Emplazamiento (Deployment)

Finalmente, la vista de emplazamiento muestra el emplazamiento físico del sistema, abarcando computadoras y dispositivos (nodos) y como se conectan entre sí. La vista de emplazamiento es para desarrolladores, integradores y probadores y es representada por los diagramas de despliegue. Esta vista incluye también un mapeo que muestra como son emplazados los componentes en la arquitectura física; por ejemplo, que programas u objetos se ejecutan en cual computadora.

Diagrama de emplazamiento (deployment)

El diagrama de emplazamiento, muestra la arquitectura física del hardware y software en el sistema. Es posible mostrar las computadoras y dispositivos (nodos), junto con las conexiones que tienen entre sí; también es posible mostrar el tipo de conexiones. Dentro de los nodos, los componentes ejecutables y los objetos son asignados para mostrar que unidades de software se ejecutan en cuales nodos. También pueden mostrarse dependencias entre componentes.

Como ya se ha establecido, el diagrama de emplazamiento, muestra la vista de emplazamiento, describiendo la arquitectura física del sistema. Esto esta muy lejos de la descripción funcional de la vista de casos de uso. Sin embargo, con un modelo bien definido, es posible navegar todo el camino desde un nodo en la arquitectura física hacia sus componentes, hacia las clase que lo implementa, hacia las interacciones de los objetos de la clase, y finalmente hacia el caso de uso en el que se dan dichas interacciones.

Diagrama de paquetes

La idea principal de este diagrama es agrupar las clases en unidades de nivel más alto. En UML, a este mecanismo de agrupamiento se le llama paquete.

La idea de un paquete se puede aplicar a cualquier elemento de un modelo, no sólo a las clases, el agrupamiento se vuelve arbitrario.

En el diagrama de paquetes se busca mostrar los paquetes de clases y las dependencia entre ellos. Los paquetes y las dependencias son elementos de un diagrama de clases, por lo cual un diagrama de paquetes es sólo una forma de un diagrama de clases.

Los paquetes no dan respuestas sobre la manera de reducir las dependencias en el sistema, pero sí ayudan a saber cuáles son las dependencias, y sólo se puede efectuar el trabajo para reducirlas, cuando es posible verlas. Los diagramas de paquetes son, una herramienta clave para mantener el control sobre la estructura global de un sistema.

5.3.3 Diseño en el sistema de procesamiento distribuido

Diagrama de emplazamiento

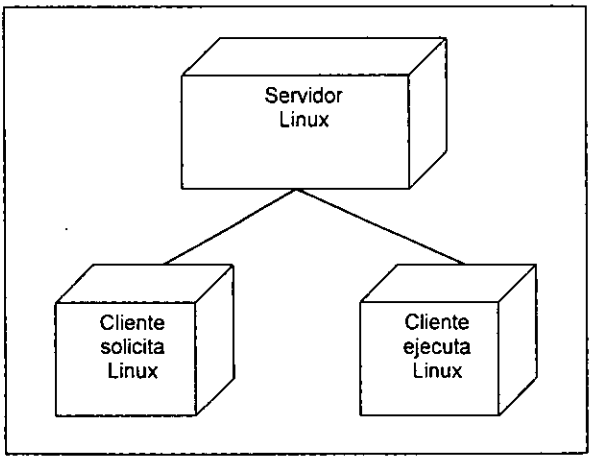

Figura 24 Diagrama de emplazamiento

Diagrama de paquetes de uso del ambiente

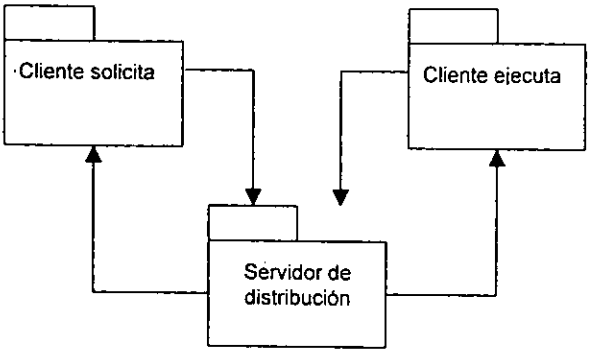

Figura 25 Diagrama de paquetes

En este diagrama podemos ver la agrupación de clases en tres paquetes, cada uno de estos paquetes tiene las siguientes clases.

- Cliente solicita: Petición, Intérprete.
- Servidor de distribución: Procesos a distribuir, Planificador, Máquina Virtual.
- Cliente ejecuta: Despacho.

Como se puede ver también en el diagrama, existe una comunicación en ambos sentidos de los tres paquetes, puesto que el cliente que solicita va a enviar la petición y también recibirá los resultados de esta petición, mientras que el cliente que ejecuta, recibe el proceso a ejecutar y entrega el resultado de este proceso ejecutado.

Tarjetas CRC

Las tarjetas de clase-responsabilidad-colaboracion (CRC) son otra forma de desarrollar modelos. La usada hasta ahora es, la basada en UML (vistas y diagramas), pero además se hará uso de ésta que muestra los objetos en forma de tarjetas con la finalidad de dejar más claros los métodos de cada clase.

En las tarjetas CRC surgen las responsabilidades, que sustituyen a los atributos y métodos. Una responsabilidad es una descripción de alto nivel del propósito de una clase. La idea es tratar de eliminar la descripción en pedazos de datos y procesos y, en cambio, captar el propósito de la clase en unas cuantas frases.

Con cada responsabilidad se indica cuáles son las otras clases con las que se tiene que trabajar para cumplirla. Esto da cierta idea sobre los vínculos entre las clases, siempre a alto nivel.

Uno de los principales beneficios de las tarjetas CRC es que alientan la disertación animada entre los desarrolladores. Son especialmente eficaces cuando se está en medio de un caso de uso para ver cómo lo van a implementar las clases. Los desarrolladores escogen las tarjetas a medida que cada clase colabora en el caso de uso. Conforme se van formando ideas sobre las responsabilidades, se puede escribir en las tarjetas. Es importante pensar en las responsabilidades, ya que evita pensar en las clases como simples depositarias de datos, y ayuda a que el equipo se centre en comprender el comportamiento de alto nivel de cada clase.

A continuación se presentará las partes de una tarjeta CRC.

Nombre de la clase	
Tipo de clase	
Descripción	
Responsabilidad	Clase vinculada
Datos de intercambio	

Un error común es generar largas listas de responsabilidades de bajo nivel. Las responsabilidades deben caber en una tarjeta, es decir, no deben tener mas de tres y si es así debe plantearse la posibilidad de dividir la clase o integrarlas en enunciados de un mayor nivel.

Se presentan las siguientes tarjetas CRC para representar el problema analizado.

Creación del ambiente	
La creación de ambiente es la superclase que tiene como subclase a la Máquina Virtual.	
Ésta, es una de las dos clases principales cuyo objetivo es levantar la máquina virtual teniendo como único nodo al servidor, así como la inicialización del ambiente distribuido.	
Iniciar la máquina virtual.	
Iniciar el ambiente distribuido	Máquina virtual
Inicialización de ambiente (comandos y/o servicios)	

Uso del ambiente	
El uso del ambiente es la superclase que agrupa Petición, Intérprete, Procesos a distribuir, Planificador, Máquina virtual, Nodo y Despacho	
Esta encargada de recibir la petición hecha por el usuario, llevar a cabo el proceso de distribución y terminar con la entrega final de resultados al usuario que realizó la petición.	
Recibir la petición hecha por el nodo cliente.	Usuario
Entregar los resultados finales al nodo cliente.	Usuario
Petición (cadena). Resultados (cadena).	

Petición	
Es una subclase de uso del ambiente.	
Es la encargada de llevar la petición del nodo cliente para su interpretación.	
Enviar la el solicitud de distribución desde nodo cliente.	Intérprete
Recibir los resultados de la solicitud hecha, desde la Máquina Virtual	Máquina Virtual
Petición (cadena).	

Intérprete	
Es una subclase de uso del ambiente.	
La función del intérprete es identificar la petición hecha, para saber si se procesará o se enviará un mensaje de error.	
Recibir la solicitud hecha por el nodo cliente.	Petición
Clasificar la solicitud.	Procesos a distribuir
Envía mensaje de error en caso de ser necesario	Petición
Proceso a distribuir (mensaje). Error (cadena).	

Procesos a distribuir	
Es una subclase del ambiente.	
Los procesos a distribuir so cada una de las tareas que serán coordinadas por el planificador.	
Recibir la clasificación del intérprete.	Intérprete
Organizar procesos para uso del planificador	Planificador
Procesos (mensaje)	

Planificador	
Es una subclase del ambiente.	
El planificador seleccionará el siguiente proceso a distribuir usando las colas de procesos y agregará los valores necesarios para su asignación y despacho.	
Prepara el proceso a distribuir agregándole información necesaria.	Procesos a distribuir
Reúne los procesos a distribuir asignado los valores necesarios.	
Seleccionar el siguiente proceso a distribuir.	Máquina Virtual
Procesos (mensaje)	

Máquina virtual	
Es una subclase de uso del ambiente.	
Esta subclase como parte del uso del ambiente, está encargada de asignar los procesos para el despacho y entregar los resultados al nodo cliente, así como de construir la máquina virtual mediante las solicitudes recibidas por cada uno de los nodos.	
Recibir solicitud del nodo cliente.	Nodo cliente
Clasificar el tipo de solicitud.	

Realizar la conexión o desconexión del nodo según sea la solicitud hecha actualizando el estado de la máquina virtual.	
Asignar el proceso a ejecutar en el procesador adecuado.	Despacho
Recibir los resultados obtenidos de la ejecución de procesos.	Despacho
Enviar los resultados al nodo cliente.	Nodo
Solicitud del nodo cliente (cadena). Procesos (mensaje). Resultados (mensaje).	

Nodo	
Es una subclase de uso del ambiente.	
Es manejado como una unidad que solicita la conexión o desconexión en la máquina virtual.	
Enviar solicitud a máquina virtual.	Máquina virtual
Recibir los resultados de la máquina virtual.	Máquina virtual
Solicitud hacia la máquina virtual (cadena).	

Despacho	
Es una subclase de uso del ambiente.	
La principal tareas de ejecutar los procesos recibidos y enviar los resultados obtenidos.	
Recibir asignación de la máquina virtual.	Máquina Virtual
Ejecutar los procesos recibidos.	
Entregar los resultados a la máquina virtual.	Máquina Virtual
Resultados (mensaje).	

5.4 PROGRAMAS

Una vez, que se tienen instaladas las herramientas necesarias, y que se ha hecho el análisis y diseño del sistema de procesamiento distribuido, se puede pasar a la fase de construcción. A continuación se muestran algunos programas que fomarían parte del sistema.

El primer programa realiza el levantamiento de la consola de *pvm* y muestra la configuración de la máquina virtual. Este programa es la ejemplificación de la parte que elaboraría el servidor dentro del análisis hecho, es decir, levanta el ambiente.

```
#include <stdio.h>
#include "pvm3.h"

/*Declaración de variables globales*/
char opcion[1];

main ()
{
   do
   {
      menu();
     opciones();
     }
   while (*opcion!='3');
}
/********* Funciones *********/
menu()
{
   printf ("\n\nSeleccione una opción\n\n");
   printf ("1.-Iniciar Máquina Virtual\n");
   printf ("2.-Configuración de Máquina Virtual\n");
   printf ("3.-Para terminar\n\n");
   printf ("Opción -->  ");
   gets (opcion);
}

opciones ()
{
   if (*opcion =='1')
      iniciar ();
   else if (*opcion=='2')
      configuracion ();
   else if (*opcion=='3')
      exit (1);
   else
      printf ("\nERROR : Opción Invalida !!!!\n\n");
}

/***** Iniciar Maquina Virtual *****/
iniciar ()
{
```

```
    /*Declaracion de variables locales*/
    int info;
    static char*argv[]={"sewa"};
    printf("\n\n ***** LEVANTAMIENTO DEL SD *****\n\n");
      info=pvm_start_pvmd(1,argv,0);
    if (info==-28)
       printf("\nERROR: Ya está el demonio corriendo !!!\n");
    else if (info==-14)
       printf("\nERROR: No responde el demonio ...\n");
    else if (info!=0)
       printf("\nERROR: %d", info);
}

/***** Configuracion Maquina Virtual *****/
configuracion()
{
    /*Declaracion de variables locales          */
    int info;
    int i=0;
    int nhost, narch;
    struct pvmhostinfo *hostp;
    printf("\n\n ***** CONFIGURACION DE SD *****\n\n");
    info=pvm_config(&nhost, &narch, &hostp);
    if (info<0)
    {
       printf("\nError en la configuración de la máquina virtual\n");
       printf("Revisar si está levantado el demonio de pvm\n");
     }
else
{
    printf("\nLa MV actual esta formada por %d maquinas:\n\n",nhost);
    for (i=0; i<nhost; i++)
    {
       printf("%d\t", hostp[i].hi_tid);
       printf("%s\t", hostp[i].hi_name);
       printf("%s\t", hostp[i].hi_arch);
       printf("%d\n", hostp[i].hi_speed);
    }
 }
}
```

A continuación se muestran dos programas que se comunican entre ellos, éstos pueden estar en diferentes máquinas del cluster, esto es una ventaja dentro de la máquina virtual, ya que cualquier nodo forma parte de una máquina central, funciona como uno sólo.

```
#include <stdio.h>
#include "pvm3.h"

main()
{
  int cc, tid, ptid:
  char buf[100];

  printf("Yo soy t%x\n", pvm_mytid());
```

```
    ptid=pvm_parent();
    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("Desde t%x: %s\n", tid, buf);
    }
    else
        printf("No se puede iniciar hello_other\n");
    pvm_initsend(PvmDataDefault);
    pvm_pkstr("terminaras de ejecutar hello");
    pvm_send(ptid,1);
    pvm_exit();
    exit(0);
}

#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);
}
```

Como se mencionó anteriormente, el concepto de máquina virtual es de suma importancia en el cluster, Es en éste, alrededor del cual giran la administración de procesos y la inserción o eliminación de nodos de la máquina virtual. En el análisis hecho previamente, se mencionó el caso de uso encargado de construir la máquina virtual, refiriéndose a la conexión y desconexión de nodos. Los siguientes dos programas ejemplifican esta funcionalidad; sirven para agregar o eliminar nodos de la máquina virtual.

Es necesario hacer uso de algunas utilidades de Linux, que se describirán brevemente. Netcat, es una de ellas, es una herramienta usada por el programa que se encuentra en el cliente, para mandar la petición de unión o eliminación de la máquina virtual.

Por otra parte el servidor, necesita saber escuchar esta petición por algún puerto y realizar lo correspondiente; para esto se ha levantado un demonio, que estará escuchando las peticiones.

Netcat lee y escribe datos a través de conexiones de red, utilizando el protocolo TCP o UDP. Está diseñada para ser una herramienta de respaldo fiable que puede usarse directamente o fácilmente guiada desde otros programas y *scripts*. Al mismo tiempo, es una herramienta rica en características de depuración y exploración de la red, puesto que puede crear casi cualquier clase de conexión que pueda necesitar, y tiene varias capacidades integradas interesantes.

La herramienta nc (o netcat) se utiliza para cualquier cosa que implica el TCP o el UDP. Puede abrir conexiones TCP, enviar paquetes del UDP, escucha cualquier puerto TCP y UDP. A diferencia de telnet, nc es agradable, y separa mensajes de error en vez de enviarlos a la salida estándar, como telnet lo hace.

Los puertos de destino pueden ser simples números, nombres como los colocados en */services* o rangos. Algunas opciones son las siguientes:

-p port	Especifica el puerto fuente que nc usará
-u	Usa UDP en lugar de TCP.
-w timeout	Especifica el número de segundos que nc esperará antes de que el intento de conexión sea terminado. También es usado para especificar el tiempo que esperará datos después de que la entrada estándar sea cerrada.

Por ejemplo, *"nc -w 5 example.host 42"*, abre una conexión TCP al puerto 42 de la máquina ejemplo, y espera 5 segundos para conectarse

Por otro lado tenemos, a los servidores de red y los servicios, que son aquellos programas que permiten a un usuario remoto hacer uso de la máquina Linux. Los programas servidores escuchan en los puertos de red. Los puertos de red son el medio de llegar a un servicio en particular en una máquina en particular.

Podemos decir, que se usa el concepto de puertos para la diferenciación de servicios. Este concepto no se refiere a puertos físicos (conexiones en una máquina), sino a puertos lógicos, desde o hacia los cuales se establece una conexión. Así, toda conexión TCP/IP está únicamente identificada por cuatro números:

- Dirección de origen (32 bits)
- Puerto de origen (16 bits)
- Dirección de destino (32 bits)
- Puerto de destino (16 bits)

Los servicios bien conocidos de TCP/IP tienen asignados puertos estándares. Así, por ejemplo, el servicio de *telnet* tiene asignado el puerto 23. Cuando un usuario ejecuta el comando *telnet* para comunicarse a otra máquina, dicho programa (el cliente) establece una conexión al puerto 23 del servidor.

En Linux, los puertos numerados del 0 al 1023 son conocidos como "puertos confiables", debido a que solamente los programas ejecutados por el superusuario pueden establecer conexiones a través de ellos. Esto pretende evitar que un usuario normal pueda obtener información a través de un servicio falso.

Frecuentemente, los servicios son llevados a cabo por los llamados **demonios**. Un demonio es un programa que abre un determinado puerto, y espera a recibir peticiones de conexión. Si se recibe una petición de conexión, lanza un proceso hijo que aceptará la conexión, mientras el padre continúa escuchando a la espera de más peticiones. Este concepto tiene el inconveniente de que por cada servicio ofrecido, se necesita ejecutar un demonio que escuche por su puerto a que ocurra una conexión, lo que generalmente significa un desperdicio de recursos de sistema.

Por ello, casi todas las instalaciones corren un ``super-servidor" que crea *sockets* para varios servicios, y escucha en todos ellos simultáneamente. Cuando un nodo remoto requiere uno de los servicios, el super-servidor lo recibe y llama al servidor especificado para ese puerto.

El super-servidor usado comúnmente es *inetd*, el demonio Internet. Es iniciado en tiempo de arranque del sistema, y toma la lista de servicios que debe tratar de un archivo de arranque denominado *"/etc/inetd.conf"*. Aparte de esos servidores invocados por *inetd*, hay varios servicios triviales que el propio *inetd* se encarga de llevar a cabo denominados servicios internos.

En conclusión, existen dos modos de operación para los demonios de red. Ambos se usan por igual en la práctica. Las dos maneras son:

- Autónomo (*standalone*). El programa demonio de red escucha en el puerto de red asignado y, cuando llega una conexión, se ocupa él mismo de dar el servicio de red.
- Esclavo del servidor *inetd*. El servidor *inetd* es un demonio de red especial que se especializa en controlar las conexiones entrantes. Tiene un archivo de configuración que le dice qué programa debe ser ejecutado cuando se reciba una conexión.

El archivo *"/etc/inetd.conf"* es el archivo de configuración para el demonio servidor *inetd*. Su función es la de almacenar la información relativa a lo que *inetd* debe hacer cuando recibe una petición de conexión a un servicio en particular. Para cada servicio que desee que acepte conexiones, deberá decirle a *inetd* qué demonio servidor de red ejecutar, y cómo ha de hacerlo.

El formato del archivo es relativamente sencillo. Es un archivo de texto en el que cada línea describe un servicio que desee proporcionar. Cualquier texto en una línea que siga a # es ignorado y se considera un comentario. Cada línea

contiene siete campos separados por cualquier número de espacios en blanco (espacio o tabulador). El formato general es el siguiente:

```
servicio tipo protocolo espera usuario servidor linea_de_comando
```

Servicio	Servicio correspondiente a esta configuración, tomado del archivo /etc/services.
Tipo	Describe el tipo de socket que esta entrada considerará relevante. Los valores permitidos son: *stream*, *dgram*, *raw*, *rdm* o *seqpacket*. Es un poco técnico por naturaleza, pero por regla general casi todos los servicios basados en *tcp* usan *stream*, y casi todos los basados en *udp* usan *dgram*.
Protocolo	Protocolo considerado válido para este servicio. Debería corresponder con la entrada apropiada en el archivo /etc/services y suele ser *tcp* o *udp*. Los servidores basados en Sun RPC (*Remote Procedure Call*) usarán rpc/tcp o rpc/udp.
Espera	Sólo hay dos valores posibles. Este campo le dice a *inetd* si el programa servidor de red libera el socket después de comenzar la ejecución, y si por tanto *inetd* podrá ejecutar otro servidor para la siguiente petición de conexión, o si *inetd* deberá esperar y asumir que el demonio servidor que esté ejecutándose controlará las nuevas peticiones de conexión. Por norma general todos los servidores *tcp* deberían tener esta entrada con el valor *nowait* y la mayoría de servidores *udp* deberían tener *wait*.
Usuario	Campo indicador de qué cuenta de usuario de /etc/passwd será asignada como dueña del demonio de red cuando se ejecute. Esto es a menudo útil si quiere protegerse ante riesgos de seguridad. Puede asignar el usuario nobody a una entrada, por lo que si la seguridad del servidor de red es traspasada el posible daño queda minimizado. Habitualmente, sin embargo, este campo está asignado a root, porque muchos servidores requieren privilegios de administrador para funcionar correctamente.
Servidor	Es el camino completo hasta el programa servidor a ejecutar para esta entrada.
Línea de comando	Es opcional. Es en donde se pone cualquier argumento de linea de órdenes que desee pasar al programa demonio servidor cuando es ejecutado.

Un ejemplo de este archivo, que fué colocado en Erandi es:

```
# /etc/inetd.conf:  see inetd(8) for further informations.
# Internet server configuration database
# Internal services
#echo          stream    tcp          nowait    root    internal
#echo          dgram     udp          wait      root    internal
daytime        stream    tcp          nowait    root    internal
daytime        dgram     udp          wait      root    internal
```

```
#chargen    stream   tcp          nowait    root   internal
#chargen    dgram    udp          wait      root   internal
time        dgram    udp          wait      root   internal
#
# These are standard services.
#
telnet      stream   tcp    nowait    root   /usr/sbin/tcpd /usr/sbin/in.telnetd
ftp         stream   tcp    nowait    root   /usr/sbin/tcpd /usr/sbin/in.ftpd
#fsp        dgram    udp    wait      root   /usr/sbin/tcpd /usr/sbin/in.fspd
#
# Shell, login, exec and talk are BSD protocols.
#
shell       stream   tcp    nowait    root   /usr/sbin/tcpd /usr/sbin/in.rshd
login       stream   tcp    nowait    root   /usr/sbin/tcpd /usr/sbin/in.rlogind
#exec       stream   tcp    nowait    root   /usr/sbin/tcpd /usr/sbin/in.rexecd
#
escucha     stream   tcp    nowait    tania  /usr/sbin/tcpd pvmmaqvirtual
```

Después de realizar cualquier cambio en el archivo "inetd.conf", es necesario indicarlo que lo vuelva a leer, para que visualice los cambios, esto se hace con el siguiente comando "killall -HUP inetd".

Otro archivo importante es *"/etc/services"* que asigna nombres a los números de puerto. Este archivo es una base de datos sencilla, que asocia un nombre entendible, con un puerto de servicio de la máquina. Su formato es bastante simple. Es un archivo de texto en el que cada línea representa una entrada a la base de datos. Cada entrada comprende tres campos separados por cualquier número de espacios en blanco (espacio o tabulador). Los campos son:

nombre puerto/protocolo sobrenombres # comentario

Nombre	Una sola palabra que representa el servicio descrito.
Puerto/protocolo	Campo está dividido en dos subcampos.
Puerto	Número que especifica el número de puerto del servicio que estará disponible.
Protocolo	Debe tener como valor *tcp* o *udp*.
Sobrenombres	Otros nombres que pueden usarse para referirse a esta entrada de servicio.

El siguiente archivo se colocó en Erandi, para poder escuchar por el puerto 1200.

```
# /etc/services:
tcpmux      1/tcp        # TCP port service multiplexer
echo        7/udp
systat      11/tcp       users
netstat     15/tcp
msp         18/udp       # message send protocol
ftp-data    20/tcp
ftp         21/tcp
ssh         22/tcp       # SSH Remote Login Protocol
```

```
telnet          23/tcp
# 24 - private
smtp            25/tcp          mail
# 26 - unassigned
time            37/tcp          timserver
whois           43/tcp          nicname
domain          53/tcp          nameserver      # name-domain server
escucha         1200/tcp
#
```

Ya que se ha hecho la explicación de las herramientas y archivos utilizados para el correcto funcionamiento de estos programas, se muestran a continuación:

```
# include <stdio.h>
# include <stdlib.h>

/*Declaración de variables globales*/
char opcion[1];
char *apuntador;
const char *cadena;
char maquina[50];
char usuario[50];

main ()
{
    do
    {
        menu();
        /*Recopilacion de informacion*/
        apuntador=getenv("USER");
        strncpy(usuario,apuntador,49);
        apuntador=getenv("HOSTNAME");
        strncpy(maquina,apuntador,49);
        strcat(usuario,"_");
        strcat(usuario,maquina);
        opciones ();
    }
    while (*opcion !='4');
    return (0);
}

/********* Funciones ********/
menu()
{
    printf ("Seleccione una opción\n\n");
    printf ("1.-Agregar a Máquina Virtual\n");
    printf ("2.-Eliminar de Máquina Virtual\n");
    printf ("3.-Para terminar\n\n");
    printf ("Opción -->  ");
    gets (opcion);
}

opciones ()
{
    if (*opcion=='1')
        agregar ();
```

```
    else if (*opcion=='2')
        eliminar ();
    else if (*opcion=='3')
        exit (1);
    else
        printf ("\nOpción Invalida, oprima enter para continuar\n");
}

/******** Agregar Nodo a Maquina Virtual ********/
agregar ()
{
  int error;
  printf ("\n\nINSERCIÓN DE NODO A LA MÁQUINA VIRTUAL\n\n");
  strcat(usuario,"_");
  strcat(usuario,"1");
  printf("\nMomento de agregar y envio a ....\n");
  error=setenv("cadena",usuario,1);
  if (error!=0)
    printf("%d\n",error);
  system("export cadena");
  system("echo Scadena     | nc 192.168.0.54 1200");
  printf("\n\n ***************************** \n\n");
}

/******** Eliminar Nodo de Maquina Virtual ********/
eliminar ()
{
  int error;
  printf ("ELIMINACIÓN DE NODO DE MÁQUINA VIRTUAL");
  strcat(usuario,"_");
  strcat(usuario,"2");
  printf("\nMomento de eliminar y envio a ....\n");
  error=setenv("cadena",usuario,1);
  if (error!=0)
    printf("%d\n",error);
  apuntador=getenv("cadena");
  system("echo Scaena | nc 192.168.0.54 1200");
  printf("\n\n ***************************** \n\n");
}

#include <stdio.h>
#include <string.h>
#include "pvm3.h"

char leido[150];    /*Variable leida en el puerto 1200*/
char usuario[50];
char maquina[50];
char accion[1];
char *hosts[2];
int infos;
int info;

main()
{
  scanf("%s",leido);
  separacion();
```

```
  printf("%s",accion);
  if (*accion=='1')
    {
    printf("\nAGREGAR\n");
    agregar();
    }
  else if (*accion=='2')
    {
    printf ("\nELIMINAR\n");
    eliminar();
    }
  else
    printf("Opcion no existente\n");
}

/********** FUNCIONES **********/
separacion()
{
  int largo, j, k=0, l=0;
  largo=strlen(leido);
  for (j=0; leido[j]!='_'; j++)
  {
    usuario[j]=leido[j];
  }
  usuario[j]='\0';
  for (j=j+1; leido[j]!='_'; j++)
  {
    maquina[k]=leido[j];
    k++;
  }
  maquina[k]='\0';
  hosts[0]=maquina;
  for (j=j+1; j<largo; j++)
  {
    accion[l]=leido[j];
    l++;
  }
  accion[l]='\0';
}

agregar()
{
  printf ("\nSe agregará la máquina %s .....\n", maquina);
  info=pvm_addhosts(hosts, 1,&infos);
  if (info<1)
  {
    printf("No se agregó la máquina \n");
    exit(0);
  }
}

eliminar()
{
  printf ("Se eliminará la máquina %s", maquina);
  info=pvm_delhosts(hosts,1,&infos);
  if (info<1 )
```

```
    {
      printf("No se eliminó la máquina \n");
      exit(0);
    }
  }
```

5.5 MOSIX Y CONDOR

Mosix y Condor son dos herramientas disponibles en la red, que tienen muchas de las características mencionadas en el análisis y diseño del sistema anteriormente realizado. Es por esto, que se hace una breve mención de ellas a continuación.

Mosix es un paquete de software que mejora el núcleo de Linux con capacidades computacionales de cluster. El núcleo mejorado permite que cualquier cluster de estaciones de trabajo y de servidores trabaje como si fuera parte de un solo sistema. Es aquí, donde encontramos lo denominado como máquina virtual.

Al ejecutarse en un cluster de Mosix, no hay necesidad de modificar las aplicaciones o de conectarse a ninguna biblioteca, o aún de asignar procesos a diversos nodos. Mosix lo hace automáticamente. Por ejemplo, es posible crear muchos procesos en el nodo de conexión y dejar a Mosix asignar estos procesos a otros nodos.

La base de Mosix son los algoritmos que vigilan y responden a la distribución desigual de recursos entre nodos. Estos algoritmos utilizan la migración de procesos para asignar y reasignar procesos entre nodos, aprovechando continuamente los mejores recursos disponibles.

Debido a que Mosix se ejecuta en el núcleo de Linux, sus operaciones son totalmente transparentes a las aplicaciones. Puede ser utilizado para definir diversos tipos de cluster, incluso un cluster con diversos CPU o velocidades de la red.

Mosix tiene también herramientas para observar lo que está pasando en cada nodo del *cluster*.

Por otro lado tenemos a **Condor** es un ambiente de cómputo de alto rendimiento, que puede manejar colecciones muy grandes de estaciones de trabajo distribuidos. Su desarrollo ha sido motivado por la necesidad de científicos e ingenieros, de aumentar la capacidad de tales colecciones. El ambiente se basa en una arquitectura de capas que proporcione alcance y flexibilidad a las aplicaciones secuenciales y paralelas.

Condor trata que los propietarios de los recursos lleven a cabo de manera exitosa el cómputo de alto rendimiento. Por lo tanto presta especial atención a los derechos y sensibilidades de los propietarios de las estaciones. Es el propietario de la estación quien define las condiciones bajo las cuales la estación de trabajo puede ser usada por un usuario externo. Condor preserva en gran medida el ambiente de ejecución de la máquina original. El trabajo de Condor consiste en que, un solo proceso será automáticamente inspeccionado y emigrado entre las estaciones de trabajo como necesidad, para asegurar la terminación eventual.

Para alcanzar el rendimiento de procesamiento más alto, Condor proporciona dos funciones importantes. Primero, los recursos disponibles los hace más eficientes poniendo las máquinas ociosas a trabajar. En segundo lugar, amplía los recursos disponibles para los usuarios, funcionando bien en un ambiente distribuido.

Condor aprovecha los recursos de cómputo que serían perdidos de otra manera y les dá buen uso. Condor organiza las tareas del científico, permitiendo la ejecución de muchos trabajos al mismo tiempo. De esta manera, las enormes cantidades de cómputo se pueden hacer con la mínima intervención del usuario. Por otra parte, Condor permite que los usuarios aprovechen las máquinas ociosas a las cuales no tendrían acceso de otra manera.

Condor proporciona otras características importantes a sus usuarios. El código fuente no tiene que ser modificado de ninguna manera para aprovechar estas ventajas. Al religar con las bibliotecas de Condor se obtienen dos capacidades más: los trabajos pueden producir puntos de verificación y pueden realizar llamadas al sistema.

Un punto de verificación es la información completa del estado de un programa. Dado un punto de verificación, un programa puede utilizar este punto para reasumir la ejecución. Para los cómputos duraderos, la capacidad de reproducir y tener puntos de verificación pueden salvar días, o aún semanas de tiempo acumulado de cómputo. Si una máquina truena, o se debe reiniciar para una tarea administrativa, el punto de verificación conserva el cómputo ya realizado. Condor hace revisión de trabajos periódicamente, para que en caso de tener problemas, el trabajo se puede continuar en otra máquina (de la misma plataforma); esto se conoce como migración de proceso.

Usando las bibliotecas de Condor, las llamadas del sistema son tomadas y realizadas por Condor, en vez del sistema operativo de la máquina remota. Condor envía la llamada del sistema de la máquina remota a la máquina donde los trabajos fueron proporcionados. La función de la llamada del sistema se ejecuta, y Condor envía el resultado de nuevo a la máquina remota.

Esta puesta en práctica tiene la ventaja que el usuario que le proporciona el trabajo a Condor no necesita una cuenta en la máquina remota.

Al unirse más máquinas al conjunto de máquinas de Condor, la cantidad de recursos de cómputo disponibles para los usuarios crece. Mientras que Condor puede manejar eficientemente la cola de trabajos donde el cluster consiste de una sola máquina, Condor trabaja extremadamente bien cuando el cluster contiene centenares de máquinas.

Los trabajos en Condor buscan las máquinas sobre las cuales pueden ejecutarse. Un trabajo requerirá una plataforma específica en la cual ejecutarse. Las máquinas tienen recursos específicos disponibles, por ejemplo la plataforma y la cantidad de memoria disponible.

Hasta ahora, no ha quedado muy claro cuales son las diferencias entre estas dos herramientas, es por eso, que se presentan algunas comparaciones.

Condor y Mosix parecen similares, debido a que han implementado una migración de procesos. Sin embargo, cada uno está intentando solucionar un problema diferente. El diseño de éstos, son necesariamente diferentes, ya que fueron basados en diferentes hipótesis

Consecuentemente, la puesta en práctica de Mosix toma típicamente la forma de un conjunto de procesadores, que actúan como blancos de la migración para cómputo científico de alto-rendimiento. Aunque Mosix se puede utilizar para pedir prestados ciclos ociosos de estaciones de trabajo, no es el foco primario.

En contraste, la motivación primaria de Condor para la migración de procesos es proporcionar un camino elegante para los procesos que utilizan ciclos ociosos del procesador en una máquina no nativa, migrando de esa máquina cuando el funcionamiento sea lento. La estrategia de la migración no proporciona un modelo completamente transparente; saben que se están ejecutando en una máquina no nativa, y la máquina originaria no tiene ningún expediente de la existencia de los procesos.

Los diseñadores de Condor fueron capaces de implementar la migración de procesos sin la modificación del núcleo.

Mosix implica grandes modificaciones al núcleo para soportar la migración de procesos, mientras que Condor es un esquema de migración de procesos que es implementado en el espacio del usuario. Aunque no se realizan cambios al código fuente, los usuarios necesitan ligar sus programas a la biblioteca de migración de procesos de Condor. La biblioteca intercepta ciertas llamadas al sistema en casos donde necesita registrar el estado sobre la llamada al sistema. La biblioteca también instala un programa piloto para que pueda responder a las señales de los demonios que se ejecutan en las máquinas, diciéndoles el punto de verificación y la culminación.

Condor confía en un regulador centralizado, que limita la escalabilidad de los sistemas e introduce un punto de ruptura para el sistema entero. En contraste, los

nodos de Mosix son todos autónomos, y cada uno utiliza un algoritmo probabilístico de la distribución para recopilar la información de un subconjunto (pequeño), aleatoriamente seleccionado de los otros nodos en el sistema. Esto hace Mosix mucho más escalable, y su control totalmente descentralizado hace más robusto en el área de rupturas.

Condor no asume que los *filesystems* disponibles en la máquina nativa, están también disponibles en la máquina receptora cuando emigra el proceso. En lugar, remite todas las peticiones del *filesystem* de nuevo a la máquina nativa, satisfaciendo la petición y remitiendo los resultados. En contraste, Mosix asume que el mismo filesystem estará disponible globalmente, pero Mosix utiliza el NFS estándar.

En Mosix, los procesos parecen ejecutarse en el nodo nativo sin importar su localización real de ejecución. El proceso en sí mismo piensa siempre que se está ejecutando en su nodo nativo, incluso si emigró y se está ejecutando realmente en otro nodo. En contraste, Condor no utiliza un modelo tan fuerte de transparencia.

Mosix procura balancear dinámicamente la carga continuamente a través del curso de la vida de todos los procesos, intentando maximizar la utilización total del procesador en el cluster. Condor asigna un proceso a un nodo ocioso al momento de crear los trabajos. Condor no procura reequilibrar dinámicamente la carga como Mosix.

Los mecánicos de migración de procesos de Condor son tales, que el estado completo del proceso está escrito en el disco cuando ocurre una migración. Después de que el estado de proceso se escribe a disco, se termina el proceso, el estado se transfiere a una máquina nueva, y el proceso es reconstruido. Esta puesta en práctica tiene efectos secundarios útiles. Por ejemplo, el archivo del estado del proceso puede ser salvado. Las puestas en práctica de MOSIX son generalmente memoria a memoria e imposibilitan estas posibilidades interesantes.

Este trabajo es el resultado de la inquietud en la investigación sobre el cómputo distribuido y todos aquellos factores que giran alrededor de éste; como son: redes, administración de procesos, administración de sistemas operativos, seguridad, entre otros.

Tomando en cuenta el objetivo de este trabajo y recordando que el procesamiento distribuido es un avance importante en el ambiente computacional, que busca las mejoras en el ámbito costo-eficiencia, potencia y rapidez del procesador, independencia de una sóla máquina central, se puede decir lo siguiente:

1. En el ámbito costo-eficiencia se ha corroborado, que sin necesidad de hacer una inversión de equipo, podemos mejorar la eficiencia en la ejecución de tareas, ya que se puede aprovechar cada uno de los recursos individuales disponibles.

2. La ejecución de tareas pesadas se lleva a cabo de una manera rápida al poder usar más de un procesador en su ejecución, es decir, se puede realizar lo que se conoce como migración de procesos.

3. Permite el manejo de la máquina virtual (unión de más de una máquina para manejar en conjunto los recursos disponibles); en dicha máquina podemos ejecutar tareas, dividiéndolas o asignando máquinas específicas.

Todos estos avances no podrían llevarse a la práctica, si no fuera por el poder de manipulación de un sistema operativo que lo permita y además teniendo en cuenta que la instalación de este sistema operativo, tampoco necesitó hacer una inversión económica en esta área.

También se elaboró el análisis y diseño del sistema de procesamiento distribuido, objetivo principal de esta tesis, sustentado en el lenguaje unificado de modelado (UML). Este análisis se fortaleció con la justificación teórica como antecedente al sustento práctico.

Podemos decir, que dicho análisis se ha englobado en dos casos de uso principales, creación del ambiente distribuido y el uso de éste. El uso del ambiente, a su vez, se ha dividido en cuatro casos de uso; que son: interpretación, administración de la máquina virtual, despacho y planificación.

Asimismo se hizo un esbozo de los requerimientos y de los actores que intervienen en los casos de uso y se ejemplificó a través de diferentes diagramas las relaciones entre éstos, así como las propiedades y características de cada uno de ellos.

Finalmente, podemos decir, el tema de tesis elegido es un tema amplio y de gran relevancia, del que pueden salir muchas cosas nuevas, que con el transcurrir el tiempo se tendrá más información, así como también van a surgir nuevas inquietudes, nuevos puntos que aclarar. Es un tema de gran proyección, del que seguramente saldrán muchos proyectos de tesis a desarrollar.

Con este trabajo se pretende dejar una base asentada para la investigación en el procesamiento distribuido. Está claro que es una pequeña base, debido a la difícil limitación de alcances, sin embargo, contribuye en gran medida como una parte importante de ese gran mundo.

En lo personal, el desarrollo de este proyecto, implicó un nuevo y un doble esfuerzo, que me permitió investigar y aprender muchas cosas nuevas y profundizar otras tantas.

Siempre un trabajo de tesis implica mucho esfuerzo y dedicación. Por lo que se culmina este trabajo dejando en él un gran apasionamiento, esfuerzo y dedicación y esperando que sea un aporte para próximos proyectos

ÍNDICE DE FIGURAS

BIBLIOGRAFÍA

Lawrence S. Orilia
Las computadoras y La Información
Mc Graw Hill

Tanenbaum, Andrews S.
Sistemas Operativos: Diseño e Implementación
Prentice-Hall

Milan & Milenkovic
Sistema Operativos: Conceptos y Diseño
Mc Graw Hill

Ida M. Flynn & Ann McIver McHoes
Understanding Operating Systems
Publishing Company

Tanenbaum, Andrews S.
Sistemas Operativos Distribuidos
Prentice-Hall

Thierauf Robert J.
Distribuided Processing Systems
Prentice Hall

Zand, John Van
Parallel Processing in Information Systems
John Wiley & Sons

Tames Mohr
Linux, Recursos para el Usuario
Prentice Hall

Jack Tackett, David Junter && Lance Brown
Linux
Prentice Hall Hispanoamericana

Hans Erik Eriksson & magnus Penker
UML Toolkit
Wiley Computer Publishing

Martin Fowler & Kendall Scott
UML Gota a Gota
Addison Wesley Longman

REFERENCIAS ELECTRÓNICAS

Tema: Sistemas Distribuidos
http://laurel.datsi.fi.upm.es/~fgarcia/sd/descripcion.html
http://golum.riv.csu.edu.au/~ialtas/module3/index.html#1
http://fciencias.ens.uabc.mx/notas_cursos/so2/expo2.htm
http://www.biocristalografia.df.ibilce.unesp.br/irbis/disertacion/node225.html
http://habitantes.elsitio.com/avilavit/index2.html
http://laurel.datsi.fi.upm.es/~fgarcia/sd/descripcion.html

Tema: Clusters
http://www.clusters.unam.mx/Clusters/Conceptos
http://www.xtreme-machines.com/x-cluster-qs.html
http://www.eafit.edu.co/revista/110/trefftz.pdf
http://yan.act.uji.es/sop/ssdditig/teo/teo.html

Tema: PVM
http://www.epm.ornl.gov/pvm/pvm_home.html
http://www.epm.ornl.gov/pvm/EuroPVM97/
http://nacphy.physics.orst.edu/PVM/pvm.html
http://www.mac.cie.uva.es/pvm.html

Tema: MPI
http://www-unix.mcs.anl.gov/mpi/index.html

Tema:Mosix
http://www.mosix.org

Tema:Condor
http://www.cs.wisc.edu/condor/

A. FUNDAMENTACIÓN TEÓRICA DE PVM

Aunque en principio, como esté codificado un programa que emplee PVM es algo muy personal del programador, la estructura básica de un programa para PVM puede ser organizada en un conjunto de bloques estructurales que realizan determinadas funciones básicas.

A.1 ESQUEMA DE LA APLICACIÓN PARA EL MAESTRO

En un programa común de PVM, el maestro se encarga tanto de un arranque para que la aplicación se ejecute correctamente, como de un cierre ordenado de la aplicación. Es útil también para grabar los resultados finales de la aplicación en un archivo y para leer los datos de entrada, si éstos tienen que ser difundidos a varios nodos o leídos con una estructura determinada. Es necesario destacar que, si una aplicación sale de PVM, se sigue ejecutando normalmente, solamente pierde su vínculo con PVM, tanto en el caso del servidor como el de los clientes. Esto es interesante, porque en caso de que por una condición de error uno de los procesos salga de la máquina paralela virtual, dicho nodo seguirá optimizando, solamente que no intercambiará información en el exterior y sus resultados quedarán en los históricos locales.

Un esquema organizado del arranque del maestro puede ser:

- Solicita su TID.
- Se inscribe en un grupo.
- Lanza los esclavos a los nodos que considere oportuno.
- Hace un bloqueo de barrera, esperando a que todos los esclavos se den de alta en el grupo.

A.1.1 Solicitud de TID

Existen algunas estructuras que son comunes en todos los programas de PVM escritos en C, como es la inserción del archivo de encabezado, que contiene toda la información necesaria acerca de la interfaz de programación.

#include "pvm3.h" al comienzo de un programa elaborado en C ó,
#include "fpvm3.h" en un programa de Fortran.

Antes de ejecutar cualquier instrucción relacionada con PVM, se debe solicitar un TID. Además de ser importante el TID para gran cantidad de operaciones, la operación de solicitud de TID automáticamente dá de alta nuestro proceso para el demonio de PVM local. En la PVM para Linux no es obligatorio

solicitar el TID, ya que toda llamada a PVM, da de alta automáticamente. Sin embargo, es recomendable, ya que determinadas implementaciones de PVM exigen que se solicite el TID antes de comenzar a trabajar. La sentencia es:

int tid=pvm_mytid()

Es importante destacar que cumple la función de solicitar un TID sólo si es la primera función de la PVM que invocamos. En otro caso, todo lo que hará será devolver el TID que ya teníamos asignado.

A.1.2 Inscripción en un grupo de tareas

Para inscribir una tarea en un grupo de tareas, basta con ejecutar desde la tarea que desea inscribirse la instrucción ***pvm_joingroup***. Esta instrucción tiene la sintaxis:

*int inum = pvm_joingroup(char *grupo)*

inum será el identificador de la tarea dentro de un grupo. En caso de que el grupo no exista, lo crea. El número de identificación del grupo es el valor que regresa esta función (inum). Es importante mencionar que una tarea puede pertenecer a más de un grupo.

A su vez, siempre se puede abandonar un grupo con:

*int info = pvm_lvgroup(char *grupo)*

A.1.3 Lanzamiento de tareas esclavas

Para lanzar tareas esclavas, basta con hacer un spawn de las tareas. El esquema será:

*int ntareas=pvm_spawn(char *tarea,char **argumentos, int flag,char *host,int ntareas, int *TIDs);*

repitiendo la línea tantas veces como tareas distintas se quieran lanzar, o tareas iguales en máquinas distintas, donde:

char *tarea	Nombre del archivo a ejecutar.
char **argumentos	Matriz con los argumentos.
int flag	Opciones para lanzar una tarea.
char *host	Nombre de la máquina en la que se va a ejecutar la tarea, si procede.

int ntareas	Número de tareas iguales que serán lanzadas en el nodo indicado.
int *TIDs	Matriz con los TIDs de las tareas creadas.

El valor devuelto por la función será el número de tareas lanzadas con éxito.

Opciones del parámetro flag para lanzar tareas

PvmTaskDefault	PVM decide en qué máquina va a lanzar las tareas.
PvmTaskHost	El parámetro host indica en qué máquina se van a lanzar las tareas.
PvmTaskArch	El parámetro host indica en que arquitectura (conjunto de máquinas con la misma variable $PVM_ARCH) se van a lanzar las tareas.
PvmTaskDebug	La tarea será lanzada junto con el depurador.
PvmTaskTrace	Se mantendrá un histórico para las llamadas a las rutinas de PVM para esa tarea.
PvmMppFront	Las tareas serán lanzadas con un front-end de MPP.
PvmHostCompl	Las tareas serán lanzadas en aquellos nodos que no cumplan las características indicadas.

Otras funciones de mantenimiento de tareas

PVM permite desde el código controlar las distintas tareas con un conjunto de funciones. Estas son:

int rc = pvm_kill(int tid)	Mata la tarea con el TID indicado.
int tid = pvm_parent()	Devuelve el TID del proceso que lanzó la tarea, o PvmNoParent si fue lanzada directamente por el usuario.
int status = pvm_stat(int tid)	Devuelve información sobre una tarea

A.2 ESQUEMA DE LA APLICACIÓN PARA EL ESCLAVO

El arranque de un esclavo es bastante más sencillo. Las funciones básicas que un esclavo va a realizar para arrancar son:

- Solicita su TID.
- Se da de alta en el grupo de su maestro, y realiza un bloqueo de barrera, esperando que todos los otros esclavos se den de alta.

A.3 ENVÍO DE DATOS

Para enviar datos, se emplean tres pasos:
1. Inicialización del buffer.
2. Empaquetado del dato.
3. Enviar el dato.

Los datos son codificados con XDR, y, por ello, es preciso empaquetar el dato para asegurar la transmisión de forma correcta.

Cuando comienza a ejecutarse un programa para PVM, tiene ya un buffer activo de emisión y otro de recepción que serán empleados para comunicarse con otros nodos. Si se quiere crear algún buffer adicional, se emplea la función:

int bufid = pvm_mkbuf(int decod)

Donde bufid será el identificador de buffer y decod la forma de codificación. Los buffers pueden ser eliminados con la instrucción:

int info = pvm_freebuf(int bufid)

Donde bufid es el identificador del buffer que se eliminará.

Otra forma bastante más cómoda que se dispone para mandar datos es con la instrucción ***pvm_psend***, que permite mandar una matriz contigua de datos del mismo tipo.

A.3.1 Inicialización del buffer

Siempre que se empaquete un mensaje, se tiene que ejecutar la instrucción ***pvm_initsend,*** que inicializará el buffer activo. En caso de no hacerlo, los datos que se manden serán los empaquetados ahora, más los que se tenían ya empaquetados previamente en el buffer. La sintaxis es:

int bufid = pvm_initsend(int encod)

El identificador de buffer devuelto por la función será el del buffer inicializado. El parámetro que se pasa a la función es la codificación, de definición similar a la del mismo parámetro en la instrucción de creación de un buffer nuevo.

Opciones de codificación del buffer

El buffer admite varias formas distintas de codificación, seleccionables empleando banderas. Estos son:

PvmDataRaw	Los mensajes serán mandados sin codificar en XDR. Si al desempaquetar no es entendido el formato, generará un error.
PvmDataDefault	Los mensajes serán mandados codificados con XDR.
PvmDataInPlace	En el buffer se tiene punteros a datos y tamaños de datos, en lugar de datos.

Funciones para operaciones con buffers

Se pueden realizar operaciones con los buffers desde el programa, que permiten emplear más de los buffers que tenemos por defecto. Para ello se emplean las funciones:

int bufid=pvm_getrbuf()	Devuelve el identificador del buffer activo de recepción.
int bufid=pvm_getsbuf()	Devuelve el identificador del buffer activo de emisión.
int bufid=pvm_setsbuf()	Activa un buffer para emisión.
int bufid=pvm_setrbuf()	Activa un buffer para recepción.

A.3.2 Empaquetado de datos

En PVM es preciso empaquetar los datos en un buffer y después mandar el buffer. Aquí tenemos una gran diferencia entre C y Fortran. Fortran tiene una instrucción para empaquetar, mientras que C tiene catorce distintas. El formato es:

*int info = pvm_pkXXX(YYY *que, int cuantos, int desde_donde)*

donde XXX es el tipo en PVM de lo que se va a mandar e YYY el tipo en C relacionado. Para empaquetar una cadena no se indican ni cuántos ni el desplazamiento, siendo la instrucción de la forma:

*int info = pvm_pkstr(char *cadena)*

Otra manera de hacerlo es la siguiente, que se invoca de la misma forma y con los mismos parámetros que el printf de C.

*int info=pvm_packf(const char *formato, ...)*

Tipos de datos para empaquetado y desempaquetado

Tipo	Tipo C	Tipo Fortran
Byte	char *	BYTE1
Entero 1 byte	short *	BYTE1
Entero 2 bytes	int *	INTEGER2
Entero 4 bytes	long int *	INTEGER4

Flotante 4 bytes	float *	REAL4
Flotante 8 bytes	Double *	REAL8
Complejo 8 bytes	Float *	COMPLEX8
Complejo 16 bytes	Double *	COMPLEX16
Cadena	Char *	STRING

Funciones para empaquetado y desempaquetado

Tipo	Empaquetado	Desempaquetado
Byte	pvm_pkbyte	pvm_upkbyte
Entero 1 byte	pvm_pkshort	pvm_upkshort
Entero 2 bytes	pvm_pkint	pvm_upkint
Entero 4 bytes	pvm_pklong	pvm_upklong
Flotante 4 bytes	pvm_pkfloat	pvm_upkfloat
Flotante 8 bytes	pvm_pkdouble	pvm_upkdouble
Complejo 8 bytes	pvm_pkcplx	pvm_upkcplx
Complejo 16 bytes	pvm_pkdcplx	pvm_upkdcplx
Cadena	pvm_pkstr	pvm_upkstr

A.3.3 Enviar datos

El envío del dato se realiza mediante la función:

int info= pvm_send(int TID, int canal)

La función anterior manda a la tarea con el TID indicado por el canal indicado.

Para mandar el buffer por defecto al mismo canal de conjunto de tareas, se puede emplear la función:

*int info = pvm_mcast(int *TIDs, int número, int canal)*

donde el primer parámetro será una matriz de TIDs y el segundo, el número de TIDs contenidos en la matriz.

Es importante tener en cuenta que las comunicaciones se realizan mandando el mensaje solamente al canal de una tarea, o de un conjunto de tareas. Si una tarea está bloqueada escuchando otro canal, no escuchará el mensaje.

La emisión no es bloqueante para el emisor, respecto a la tarea del receptor, mas sí respecto al demonio del emisor. Esto significa que la tarea cliente quedará bloqueada en la emisión hasta que todo el mensaje sea transmitido vía TCP para

el demonio. Después, la tarea continuará ejecutándose, y el servidor intentará contactar con el demonio de la tarea receptora.

A.4 RECEPCIÓN DE DATOS

Para recibir datos, se debe realizar los siguientes dos pasos:

1. Recibir el mensaje.
2. Desempaquetar el dato.

Exactamente igual que el envío, se emplea un buffer, sólo que ahora no hay que limpiarlo antes de recibir cada dato.

Es importante recordar que el buffer de envío y de recepción, al arrancar una tarea en PVM, son distintos; y tienen distintas instrucciones para ser activados.

A.4.1 Recibir el mensaje

Al recibir el mensaje, la tarea receptora queda bloqueada contactando con su servidor para un canal determinado. Según el servidor y el modo de recepción, la tarea quedará bloqueada o no, esperando un mensaje. De cualquier forma, mientras el mensaje es transmitido entre el demonio receptor y la tarea receptora, la tarea receptora es bloqueada. El mensaje es transmitido entre demonio receptor y tarea receptora vía TCP. Además de todo esto, la espera entre emisión del mensaje y recepción puede ser bloqueante o no según la función de recepción. Estas son:

int bufid=pvm_recv(int TID, int canal) : bloqueante.

El receptor esperará a que la tarea TID le mande un mensaje por el canal indicado, entonces transferirá el mensaje al buffer de recepción por defecto -bufid-.

int bufid=pvm_nrecv(int TID, int canal): no bloqueante.

Si se encuentra en el servidor un mensaje de la tarea TID para el canal apropiado ya lista, se transfiere el mensaje de forma bloqueante al buffer de recepción por defecto. Si no, la función devuelve como bufid el valor 0, y no queda bloqueada.

Una función interesante es ***pvm_probe***, que verifica si ha llegado un mensaje determinado. Su comportamiento es exactamente igual que el de ***pvm_nrecv***, sólo que no recibe el paquete; por lo que para recibirlo sigue siendo precisa una instrucción de recepción de paquete. Su sintaxis es:

int bufid = pvm_probe(int TID, int canal)

aún sin bajarlo, podemos obtener información detallada del mensaje con la función ***pvm_bufinfo***, que da información sobre el mensaje recibido antes de descargarlo; sólo se necesita combinar su uso con ***pvm_probe.***

Cualquiera de las funciones de recepción antes mencionadas pueden tener como parámetro TID el valor -1, lo que significa cualquier tarea. Del mismo modo, un canal con valor -1 significa cualquier canal.

A.4.2 Desempaquetado de datos

El desempaquetado de datos es exactamente igual que el empaquetado de datos, con sus catorce funciones y con todos sus parámetros iguales, solo que cambiando ***pvm_pk*** por ***pvm_upk***. Es decir, el formato es:

*int info = pvm_upkXXX(YYY *que, int cuantos, int desde_donde)*

donde XXX es el tipo en PVM de lo que se va a mandar e YYY el tipo en C relacionado, y para la cadena tampoco indicamos ni cuántos ni el desplazamiento, siendo la instrucción de la forma:

*int info = pvm_upkstr(char *cadena)*

Exactamente igual que con el empaquetado de datos, tenemos la función:

*int info=pvm_upackf(const char *formato, ...)*

A.5 SINCRONIZACIÓN DE BARRERA

Este mecanismo de sincronización tiene como idea básica no dejar a ninguna tarea pasar hasta que un número suficiente de tareas estén esperando en la barrera. El punto de espera se define con la función ***pvm_barrier***. El problema básico es que precisa saber cuántos procesos han de esperar en la barrera antes de dejarlos pasar a todos, información que el maestro conoce, mas los esclavos no; o, al menos, no basta con saber cuántos están en un grupo, ya que puede que algunos elementos todavía no hayan entrado. De ahí que se forme un grupo, y el maestro mande esta información a todo el grupo con ***pvm_mcast***. El formato de la instrucción es:

*int info=pvm_barrier(char *grupo, int cuantos)*

donde el primer parámetro es el grupo sobre el que se realiza la función, y el segundo, cuántos tienen que esperar para pasar. En caso de que se emplee el grupo sólo para sincronía, se puede indicar en el segundo parámetro -e, lo que

significa que tomará todos los miembros del grupo en ese momento y lo empleará como parámetro, por lo que no será preciso que el maestro mande la información.

A. 6 SALIDA DE LA PVM

La salida es bastante fácil, basta con ejecutar la función ***pvm_exit.*** Se puede desde el maestro ir matando los esclavos con ***pvm_kill,*** pero es un procedimiento bastante poco recomendable.

A.7 FUNCIONES PARA OPERAR SOBRE EL CONJUNTO DE MÁQUINAS

Así como se puede operar sobre el conjunto de máquinas desde un archivo de configuración, también se puede hacer desde el propio programa. Para operar sobre el conjunto de máquinas desde el programa las funciones son:

pvm_addhosts	Añade un conjunto de máquinas de PVM.
pvm_delhosts	Elimina un conjunto de máquinas de PVM.
int pvm_mstat	Devuelve PvmOk si la máquina indicada está en PVM, PvmNoHost en cualquier otro caso.
int pvm_config	Devuelve información acerca del estado de la máquina paralela virtual.
int rc = pvm_halt	Para la máquina paralela virtual.
int dtid = pvm_start_pvmd	Arranca el demonio para la máquina donde se ejecuta la tarea.
int dtid = pvm_tidtohost	Devuelve el identificador de máquina de un proceso. Sólamente extrae el dato del TID sin comprobar ni que exista ni que esté activo.

A. 8 FUNCIONES CON GRUPOS DE LA PVM

Además de las operaciones con grupos básicas ya explicadas, disponemos de otras operaciones adicionales bastante interesantes. Estas son:

pvm_gather	Rrecoge los datos de todo un grupo para almacenarlos en una matriz. Lo opuesto a pvm_scatter.
pvm_gettid	Devuelve el TID de una tarea para un número de instancia.
pvm_reduce	Aplica un operador de reducción sumatoria, producto, máximo, mínimo y otros definidos por el usuario a los miembros de un grupo, generando un solo escalar.
pvm_scatter	Distribuye datos de una tarea a las restantes de un grupo. Los datos están originariamente en una matriz, y se mandan los count primeros a la primera, los count siguientes a la siguiente y asi los

	restantes.
pvm_getinst	Devuelve el número de instancia de una tarea dentro de un grupo.
pvm_mcast	Permite mandar un mismo mensaje a muchos procesos.
pvm_bcast	Permite a un proceso enviar un mensaje a todo un grupo.
pvm_getinst	Regresa el grupo de procesos.
pvm_gett	Regresa el identificador del grupo.
pvm_gsize()	Regresa el tamaño del grupo.

B. FUNDAMENTACIÓN TEÓRICA DE MPI

B.1 DEFINICIÓN DE TÉRMINOS

B.1.1 Tipos de argumentos

Los argumentos en las llamadas a funciones MPI se clasifican en tres tipos: IN, OUT ó INOUT, cuya semántica es la siguiente:

- **IN**: Un argumento IN no es alterado por la función, ésta solo toma su valor y lo utiliza internamente.
- **OUT**: Los argumentos OUT son utilizados por las funciones para colocar en ellos valores que son resultado de la operación de dichas funciones. Por lo tanto, el valor de este tipo de argumento es alterado después de la ejecución de la función.
- **INOUT**: Estos argumentos son utilizados tanto para proporcionar valores a las funciones (IN) como para que éstas modifiquen su valor de acuerdo al resultado de alguna operación (OUT).

Esta clasificación está basada en la manera en que una función particular utiliza un argumento dado y no en el tipo de dato de la variable usada como argumento.

B.1.2 Rangos, Grupos, Etiquetas, Contextos y Comunicadores

El modo en que MPI identifica a los diversos procesos consiste en asignar a cada uno de éstos un número entero diferente. Dicho número se conoce como rango. Debido a que el rango permite la distinción entre procesos diferentes, es utilizado en las operaciones de comunicación, para determinar el proceso que envía un mensaje y el proceso que debe recibirlo.

Por otra parte, debido a que un proceso dado puede mandar dos o más mensajes diferentes a un mismo proceso receptor, es necesario utilizar un mecanismo que permita a este último distinguir los distintos mensajes que recibe. Esto se logra mediante el uso de etiquetas en los mensajes. Las etiquetas son números enteros escogidos arbitrariamente pero cuidadosamente por el usuario, con el objeto de distinguir los diversos mensajes de un programa.

En la operación de recepción de MPI, pueden especificarse valores comodines tanto para el rango del proceso origen del mensaje como para la etiqueta de éste. Debido a esta característica, un proceso podría interceptar mensajes interfiriendo con la lógica del programa; esta situación puede causar

errores, principalmente cuando se usan bibliotecas paralelas, en las cuales grupos de procesos intercambian información para realizar cierta tarea. Para evitar estos errores, es posible crear canales de comunicación específicos, llamados comunicadores, usando dos conceptos: grupo y contexto. Un grupo es un conjunto de procesos que trabajan concurrentemente para alcanzar un determinado objetivo. El concepto de contexto es usado para separar los objetivos para los cuales los diversos procesos realizan las tareas encomendadas. Los comunicadores combinan estos dos conceptos y son usados por MPI para, a petición explícita del usuario, crear grupos de procesos independientes de acuerdo a los diferentes contextos. Estas condiciones permiten que únicamente los procesos pertenecientes al grupo del comunicador puedan recibir los mensajes enviados dentro de ese contexto.

El grupo especial ***MPI_Group_empty*** sirve para denotar aquel grupo que no tiene miembros. Mientras, la constante ***MPI_Group_null*** es un valor usado para referirse a un grupo no válido.

El comunicador especifica un dominio de comunicación, puede ser usado para las comunicaciones entre los procesos de un mismo grupo en cuyo caso se habla de un intracomunicador (intracommunicator), o para comunicaciones punto a punto (las únicas permitidas). Entre procesos de diferentes grupos, se denominan intercomunicadores (intercommunicator).

B.1.3 Tipos de funciones y operaciones

Las funciones de MPI se clasifican dado su comportamiento en funciones de bloqueo y de no bloqueo. Se dice que una función es de bloqueo si no permite la continuación del flujo del programa hasta que se haya cumplido el objetivo de dicha función. Similarmente, una función es de no bloqueo si el flujo del programa puede continuar aún cuando la función no ha cumplido completamente su objetivo.

Por otra parte, se dice que una operación es local si no requiere que otro proceso realice alguna operación para poder ser complementada, mientras que una operación es no local si requiere de la intervención de otro proceso para completarse

B.1.4 Estructuras Internas

Objetos opacos y manijas

Un programa MPI maneja 2 tipos de memoria: la memoria asignada al usuario y la memoria asignada al sistema. La memoria del sistema es usada como buffer, para el envío de mensajes y para el almacenamiento de diversos objetos

que representan distintas entidades tales como grupos, contextos, tipos de datos, etc. Ya que la memoria del sistema no es directamente accesible al usuario se dice que los objetos almacenados en ésta, son opacos. Los objetos opacos son accesibles a través del uso de parámetros llamados manijas, las cuales son definidas en la memoria del usuario. Por lo tanto, las funciones de MPI que operan sobre objetos opacos requieren argumentos de tipo manija para poder acceder a tales objetos. Las manijas son variables declaradas por el usuario.

Constantes

La definición de constantes, los prototipos de las funciones, y la definición de nuevos tipos deben suministrarse mediante la inclusión de un archivo denominado "mpi.h". La mayoría de las funciones retornan un código que indica si la función finalizó exitosamente o no.

Tipo de dato choice

En la descripción de algunas funciones de MPI se especifican argumentos de un tipo de dato denominado choice; esta descripción indica que distintas llamadas a una de estas funciones pueden tener argumentos de distintos tipos.

Tipo de dato derivados

MPI tiene una gran potencialidad al permitir al usuario construir nuevos tipos de datos en tiempo de ejecución; estos nuevos tipos son llamados tipos de datos derivados. A través de los tipos de datos derivados, MPI soporta la comunicación de estructuras de datos complejas tales como secciones de arreglos y estructuras que contienen combinaciones de tipos de datos primitivos. Los tipos derivados se construyen a partir de datos básicos usando los constructos ***MPI_Type_contiguous***, que es el constructo para tipos más simple. Define un arreglo de tipos de datos iguales que se agrupan de forma continua.

Un tipo derivado es un objeto opaco que especifica dos cosas; una secuencia de datos primitivos y una secuencia de desplazamientos.

La extensión de un tipo de dato se define como el espacio que hay del primer byte al ultimo byte que ocupa el tipo de dato, redondeado hacia arriba para satisfacer requerimientos de alineación. Si por ejemplo tipo={(double,0),(char,8)}, un doble con desplazamiento 0 y un carácter con desplazamiento 8, y los dobles deben ser alineados en direcciones que son múltiplos de 8, entonces el tamaño de este tipo es 9 mientras que su extensión es 16 (9 redondeado al próximo múltiplo de 8). La extensión y tamaño de un tipo de dato se puede obtener con ***MPI_Type_extent*** y ***MPI_Type_size*** respectivamente.

MPI_Type_vector define un tipo de dato derivado que consiste de un arreglo de bloques separados por una distancia fija. Cada bloque se obtiene por la concatenación de un mismo número de copias del tipo de dato.

MPI_Type_indexed permite especificar un diseño de datos no continuos donde el desplazamiento entre bloques sucesivos no necesariamente es el mismo. Esto permite recolectar entradas arbitrarias de un arreglo y enviarlas en un mensaje, o recibir un mensaje y difundir las entradas recibidas dentro de posiciones arbitrarias de un arreglo.

B.2 FUNCIONES BÁSICAS

B.2.1 Inicialización y Finalización

Antes de utilizar cualquier función de comunicación se debe hacer una llamada a la función ***MPI_Init***, con el fin de inicializar el ambiente MPI. Al finalizar el programa, debe hacerse una llamada a la función ***MPI_Finalize***. Las definiciones de estas funciones son:

*int MPI_Init (int *argc, char ***argv);*
int MPI_Finalize (void);

Los argumentos argc y argv en ***MPI_Init*** solo son requeridos en C. En C, todos los nombres de las funciones MPI comienzan con el prefijo MPI_ seguidas por una letra mayúscula, mientras que todas las constantes definidas son nombres con letras mayúsculas. El encabezado "*#Include <mpi.h>*" provee los tipos y las definiciones básicas para MPI. Esta instrucción es obligatoria en todo programa MPI

B.2.2 Características del ambiente

Dos datos importantes acerca de un programa en paralelo son: el número de procesos que conforman el programa y el identificador de cada proceso. Estos datos pueden ser obtenidos mediante las funciones ***MPI_Comm_size*** y ***MPI_Comm_rank***, cuya sintaxis es:

MPI_Comm_size (comm, size)

Comm	*IN*	*Comunicador(manija)*
Size	*OUT*	*Tamaño del comunicador (entero)*

MPI_Comm_Rank (comm,rank)

Comm	*IN*	*Comunicador (manija)*
Rank	*OUT*	*Rango del proceso dentro del comunicador (entero)*

Las definiciones en C son las siguientes:

*int MPI_Comm_Size (MPI_Comm, int *size)*
*int MPI_Comm_Rank (MPI_Comm, int *rank)*

El identificador o rango es un número entre cero (0) y el total de procesos menos uno (procesos –1)

MPI maneja el sistema de memoria que es usado para almacenar mensajes y las representaciones internas de objetos de MPI como grupos, comunicadores, tipos de datos, etc. El usuario no puede accesar directamente esta memoria y los objetos ahí almacenados se dicen que son opacos: su tamaño y forma no son visibles al usuario. Los objetos son accesados usando asas (handles), que existen en el espacio del usuario.

Un comunicador identifica un grupo de procesos y el contexto en el cual se llevará a cabo una operación. Ellos proveen un mecanismo para identificar subconjuntos de procesos en el desarrollo de programas modulares y para garantizar que mensajes concebidos para diferentes propósitos no sean confundidos. El valor por omisión ***MPI_Comm_world*** identifica todos los procesos involucrados en un cómputo.

B.2.3 Mediciones de tiempos de ejecución

Medir el tiempo de duración de un programa es importante para establecer su eficiencia y para fines de depuración, sobre todo cuando éste es paralelo. Para esto MPI ofrece las funciones ***MPI_Wtime*** y ***MPI_Wtick***. Ambas proveen alta resolución y poco costo comparadas con otras funciones similares existentes en otros lenguajes.

MPI_Wtime devuelve un punto flotante que representa el número de segundos transcurridos a partir de cierto tiempo pasado, el cual se garantiza que no cambia durante la vida del proceso. Es responsabilidad del usuario hacer la conversión de segundos a otras unidades de tiempo.

MPI_Wtick permite saber cuantos segundos hay entre tics sucesivos del reloj. Por ejemplo, si el reloj esta implementado en hardware como un contador que se incrementa en cada milisegundo, entonces ***MPI_Wtick*** debe devolver 10^{-3}

La definición de esta función es:

double MPI_Wtime (void)
double precision MPI_Wtime ()

B.3 COMUNICACIONES PUNTO A PUNTO

El envío y recepción de mensajes (operación send y receive) es el mecanismo básico de comunicación entre los procesos proporcionado por MPI: Estas operaciones están implementadas en dos formas diferentes: de bloqueo y de no bloqueo.

B.3.1 Envío de bloqueo

El envió de mensajes de bloqueo se realiza mediante la función ***MPI_Send***, la cual tiene la siguiente sintaxis:

MPI_Send (buf, count, datatype, dest, tag, comm)

Buf	*IN*	*Dirección del primer dato del mensaje (choice)*
Count	*IN*	*Número de elementos que contiene el mensaje (entero)*
Datatype	*IN*	*Tipo de dato de cada elemento del mensaje (entero)*
Dest	*IN*	*Rango del proceso destino (entero)*
Tag	*IN*	*Etiqueta del mensaje (entero)*
Comm	*IN*	*Comunicador a través del cual circula el mensaje*

La definición en C es la siguiente:

int MPI_Send (void buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);

La descripción del contendido del mensaje (dirección inicial, número de datos y tipo de éstos) es dada por los parámetros buf, count y datatype, mientras que el sobre del mensaje se especifica mediante dest, tag y comm. Los tipos de datos (datatype) que pueden ser especificados corresponden a los tipos de datos básicos del lenguaje utilizado.

B.3.2 Recepción de bloqueo

La recepción de bloqueo se realiza mediante la función ***MPI_Recv***, cuya sintaxis es:

MPI_Recv (buf, count, datatype, source, tag, comm, status)

Buf	*OUT*	*Dirección a partir de la cual se almacenará el mensaje*
Count	*IN*	*Número de elementos a recibir (entero)*
Type	*IN*	*Tipo de dato de cada elemento del mensaje (manija)*
Source	*IN*	*Rango del proceso originado de los datos (entero)*
Tag	*IN*	*Etiqueta del mensaje (entero)*
Comm	*IN*	*Comunicador a través del cual circula el mensaje*
Status	*OUT*	*Estado de la operación (status)*

La definición en C es la siguiente:

Int MPI_Recv (void buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_COMM comm, MPI_Status status);

El proceso receptor puede especificar valores comodines tanto para el origen como para la etiqueta de un mensaje (definidos como ***MPI_Any_source*** y ***MPI_Any_tag***, respectivamente); el origen y la etiqueta del mensaje recibido son desconocidos si se han utilizado estos valores comodines en la función de recepción. La información acerca de estos parámetros, así como el código de error producido por la operación de recepción, son almacenados en el parámetro status. El argumento status también contiene información sobre la longitud del mensaje recibido, sin embargo, ésta no es accesible directamente como un campo más de este argumento. La función ***MPI_Get_count*** proporciona dicha longitud, a partir de los valores de la variable status. La sintaxis de esta función es:

MPI_Get_count (status, datatype, count)

Status	*IN*	*Estado de la operación de recepción (status)*
Datatype	*IN*	*Tipo de datos recibido (manija)*
Count	*OUT*	*Número de elementos recibidos (entero)*

B.3.3 Funciones de No Bloqueo

El rendimiento en un cálculo paralelo puede ser incrementado traslapando las operaciones de cómputo con las operaciones de comunicación. Esto se consigue utilizando operaciones de comunicación de no bloqueo. Una operación de no bloqueo inicia el envío o la recepción de un mensaje permitiendo la continuación del flujo del programa aún antes de que ésta sea completada. Es necesaria la ejecución de otra función que determina si la operación de comunicación ha concluido. De esta manera, las operaciones de no bloqueo se realizan en dos pasos: inicio y finalización.

Las funciones que inician una operación de comunicación de no bloqueo utilizan un objeto opaco del tipo request como identificador de la operación de comunicación que inician. Los objetos request almacenan información como: el modo de comunicación (en el caso de que la operación sea el envío de un mensaje), el espacio de memoria que contiene el mensaje, su contexto, la etiqueta del mensaje, el origen o el destino (ya sea el caso de una operación send o receive), además del estado de la operación. A continuación se describe las funciones de no bloqueo que inician las operaciones send y receive, así como las funciones que finalizan dichas operaciones.

Envío de mensajes

La función de no bloqueo que inicia una operación send es la siguiente:

MPI_Isend (buf, count, datatype, dest, tag, comm, request)

Buf	*IN*	*Dirección del primer dato del mensaje (choice)*
Count	*IN*	*Número de elementos del mensaje (entero)*
Datatype	*IN*	*Tipo de dato de cada elemento del mensaje (manija)*
Dest	*IN*	*Rango del proceso receptor (entero)*
Tag	*IN*	*Etiqueta del mensaje (entero)*
Comm	*IN*	*Comunicador a través del cual circula el mensaje*
Request	*OUT*	*Identificador de la operación de comunicación (manija)*

Los argumentos buf, count, datatype, dest, tag y comm tienen el mismo significado que en la función de bloqueo ***MPI_Send***. La función aquí descrita aloja un objeto de tipo request y lo asocia con la manija del argumento request.

El uso de una función de no bloqueo que inicia la operación send indica al sistema que puede empezar a copiar los datos del mensaje, ya sea en la memoria del proceso receptor o en un buffer. El proceso que envía el mensaje no debe modificar los datos de éste hasta que la operación haya sido completada.

Recepción de mensajes

Para la recepción de no bloqueo se utiliza la siguiente función:

MPI_Ireceive (buf, count, datatype, source, tag, comm, request)

Buf	*OUT*	*Dirección del buffer que recibirá el mensaje (choice)*
Count	*IN*	*Número de elementos a recibir (entero)*
Datatype	*IN*	*Tipo de dato de cada elemento a recibir (manija)*
Source	*IN*	*Identificación del proceso originario de los datos*
Tag	*IN*	*Etiqueta del mensaje (entero)*
Comm	*IN*	*Comunicador a través del cual circula el mensaje*
Request	*OUT*	*Identificador de la operación de comunicación (manija)*

Los argumentos buf, count, datatype, source, tag y comm tienen el significado descrito para la función de bloqueo ***MPI_Recv.*** La función ***MPI_Irecv*** aloja en la memoria del sistema un objeto de tipo request, asociado a la manija usada en el argumento request. El uso de la función ***MPI_Irecv*** indica al sistema que puede iniciar la escritura en el área de memoria señalada por los parámetros buf, count y datatype de la llamada a la función. El proceso receptor no debe modificar dicha memoria mientras la recepción no se haya completado.

Finalización de operaciones de no bloqueo

Las funciones ***MPI_Wait*** y ***MPI_Test*** son utilizadas para determinar si una comunicación iniciada por una función de no bloqueo ha terminado. La finalización de una operación de envío significa que el proceso que envió el mensaje tiene ya la libertad de escribir en el área de memoria en que éste se encuentra almacenado sin comprometer su integridad. La finalización de una operación de recepción

significa que el mensaje ha sido completamente copiado a la memoria del proceso receptor y que, por lo tanto, éste puede tener acceso a dicha área de memoria; además, la finalización de una operación de recepción indica también que el estado de la operación ha sido obtenido. La sintaxis de la función ***MPI_Wait*** es:

MPI_Wait (request, status)

Request	*INOUT*	*Identificador de la operación de comunicación*
Status	*OUT*	*Estado de la operación de comunicación (status)*

Esta función permite el flujo del programa hasta que la operación de comunicación identificada por request haya terminado. El objeto asociado con la manija request es desalojado de la memoria al término de esta función, colocando en dicha manija el valor nulo definido como ***MPI_Request_null***. La variable status contiene la información descrita anteriormente.

Por otra parte, la sintaxis de la función ***MPI_Test*** es:

MPI_Test (request, flag, status)

Request	*INOUT*	*Identificador de la operación de comunicación*
Flag	*OUT*	*Bandera que indica el estado de la operación*
Status	*OUT*	*Estado de la operación de comunicación (status)*

La función ***MPI_Test*** regresa un valor verdadero en el argumento flag si la operación de comunicación asociada al request ha terminado, en tal caso, el argumento status contendrá el estado de dicha operación y el objeto asociado con request será desalojado de la memoria. En el caso de que la operación no haya sido terminada, la función regresará un valor falso en el argumento flag. Esta función es local.

B.3.4 Modos de comunicación

La acción de copiar el mensaje a un buffer desacopla las operaciones de send y receive, permitiendo que la operación send de bloqueo pueda completarse tan pronto como el mensaje haya sido copiado en el buffer, aún cuando la operación de recepción no haya sido ejecutada todavía por el proceso receptor; sin embargo, el uso de buffers en las operaciones send tiene una alto costo, tanto por la sobrecarga adicional que representa la acción de copiar el mensaje al buffer como por los mayores requerimientos de memoria para el sistema. Dada la posibilidad de copiar los datos de un mensaje en un buffer temporal, se tienen varios modos de comunicación posibles para el envío de mensajes. Estos modos permiten la elección explícita de la forma en que la operación send debe realizarse. Los modos de comunicación existentes son: con buffer, síncrono, ready y estándar.

En el modo de comunicación con buffer la operación send, inicia e incluso se completa aún cuando el proceso receptor no haya comenzado la operación de recepción. La función que realiza la operación send en este modo es local, ya que todos los mensajes son copiados a un buffer temporal, independientemente de que el proceso destinatario ejecute o no la operación de recepción. En el caso de que el tamaño del buffer sea insuficiente para copiar un mensaje dado, ocurrirá un error que generalmente produce la finalización anormal del programa.

Una operación send en el modo síncrono, puede iniciarse aún si no existe alguna operación de recepción, sin embargo finaliza hasta que el proceso de recepción se haya completado, es decir, hasta que el mensaje ha sido copiado en la memoria del proceso receptor. Este modo provee una comunicación síncrono, ya que ambas operaciones (envío y recepción) finalizan simultáneamente.

En el modo de comunicación ready, una operación send puede iniciar únicamente si el proceso de recepción ha sido también iniciado de otra manera, la operación es errónea y su comportamiento es indefinido. Este modo de comunicación funciona de la misma forma que el modo síncrono, con la excepción de que evita sobrecargas debidas a la espera del inicio de la operación de recepción.

Las funciones que envían un mensaje en los modos con buffer, síncrono y ready, en comunicación de bloqueo y no bloqueo, son las siguientes:

MPI_Bsend (buf, count, datatype, dest, tag, comm)
MPI_Ssend (buf, count, datatype, dest, tag, comm)
MPI_Rsend (buf, count, datatype, dest, tag, comm)
MPI_Ibsend (buf, count, datatype, dest, tag, comm, status)
MPI_Issend (buf, count, datatype, dest, tag, comm, status)
MPI_Irsend (buf, count, datatype, dest, tag, comm, status)

En el modo estándar de comunicación, el MPI decide automáticamente cuáles mensajes serán enviados en el modo buffer y cuales en el modo síncrono. Esta elección dependerá principalmente de la cantidad de memoria disponible.

B.3.5 Operaciones colectivas

Como se ha definido anteriormente, las operaciones colectivas son aquellas que deben ser realizadas por todos los procesos pertenecientes a un grupo dado para poder ser completadas. Esto es una operación colectiva terminará hasta que todos los procesos miembros del grupo hayan realizado la llamada a la función correspondiente. En general, las operaciones colectivas se pueden dividir en los siguientes tipos:

Sincronización de procesos. Permiten que un grupo de procesos realice las mismas operaciones simultáneamente.

Distribución de datos. Éstas se encargan de la distribución de un conjunto de datos entre los diversos procesos que pertenecen a un grupo.

Cálculos colectivos. Realizan las operaciones de suma, producto, AND, OR, entre otras, con datos distribuidos entre los procesos miembros del grupo.

Obviamente, es posible construir cualquiera de las operaciones colectivas utilizando las operaciones básicas en MPI, ya que además de que simplifican la programación, generalmente estas implementaciones ya han sido optimizadas. A continuación se describe tanto las operaciones colectivas definidas por MPI como las funciones que las realizan.

Barreras

Las barreras son operaciones colectivas que permiten la sincronización de procesos. Cuando un proceso realiza la operación de barrera, el flujo se detiene hasta que el resto de los procesos dentro del grupo ejecuta la operación. La función que realiza esta operación es ***MPI_Barrier***, cuya sintaxis es:

MPI_Barrier (comm)
Comm IN Comunicador (manija)

El comunicador comm indica qué grupo de proceso debe ejecutar la operación de barrera para que ésta pueda concluir.

Difusión de datos

La operación de difusión de datos consiste en la propagación de un conjunto de datos pertenecientes a un proceso particular (al cual se le conoce como proceso raíz de la operación) hacia todos los demás procesos del grupo. La sintaxis de la función ***MPI_Bcast***, que realiza esta operación es:

MPI_Bcast (buf, count, datatype, root, comm)

Buf	*INOUT*	*Dirección de memoria del primer dato (choice)*
Count	*IN*	*Número de datos del buffer (entero)*
Datatype	*IN*	*Tipo de dato (manija)*
Root	*IN*	*Rango del proceso raíz de la operación (entero)*
Comm	*IN*	*Comunicador (manija)*

La dirección de memoria buf indica, para el proceso raíz, la dirección de inicio de los datos que van a ser transmitidos, mientras que para el resto de los procesos indica la dirección de memoria a partir de la cual los datos serán almacenados. Al finalizar la operación, todos los procesos tendrán en la dirección de memoria buf los mismos datos.

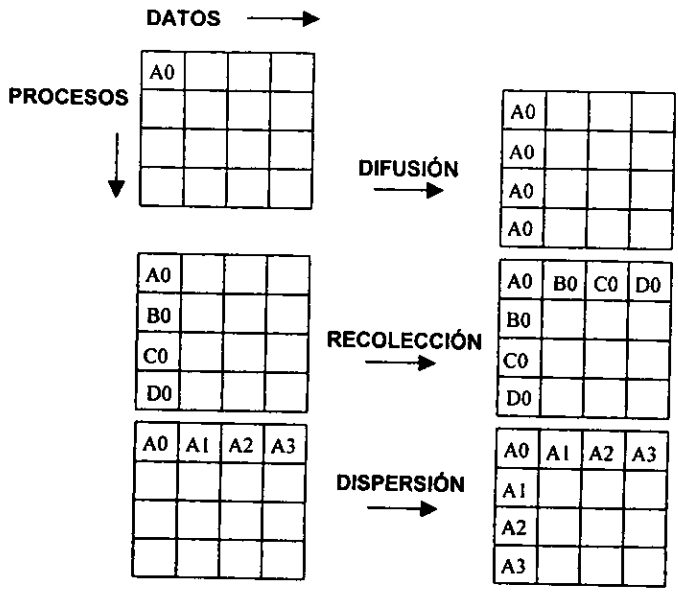

Figura A1. Distintas operaciones de distribución de datos

Recolección de datos

En una operación de recolección todos los procesos envían un conjunto de datos particular al proceso raíz de la operación. Este proceso coloca los datos recibidos en localidades contiguas de memoria, de modo que los datos procedentes del proceso i precedan a los datos procedentes del proceso i+1. Esto implica que el espacio de memoria reservado para recibir los datos debe ser "p" veces la longitud de los datos enviados por cada proceso (en donde p es el número de procesos que intervienen en la operación). La sintaxis de la función ***MPI_Gather***, la cual realiza la operación de recolección, es la siguiente:

MPI_Gather (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

Sendbuf	*IN*	*Dirección inicial de los datos a ser enviados (choice)*
Sendcount	*IN*	*Número de datos a ser enviados (entero)*
Sendtype	*IN*	*Tipo de dato a ser enviado (manija)*
Recvbuf	*OUT*	*Dirección de almacenamiento de los datos (choice)*
Recvcount	*IN*	*Número de datos a ser recibidos (entero)*
Recvtype	*IN*	*Tipo de dato a ser recibido (manija)*
Root	*IN*	*Rango del proceso raíz de la operación (entero)*
Comm	*IN*	*Comunicador de la operación (manija)*

Al finalizar la operación de recolección, el proceso raíz tendrá en la dirección señalada por recvbuf la concatenación del contenido sendbuf de todos los procesos, incluyéndose a sí mismo.

Dispersión de datos

La operación de dispersión de datos es inversa a la operación de recolección, esto es, el proceso raiz distribuye un conjunto de datos entre los diversos procesos que participan en la operación, incluyéndose a sí mismo. A diferencia de la operación de difusión, en donde todos los procesos reciben exactamente los mismos datos, en la operación de dispersión cada proceso recibe una parte diferente del conjunto de datos. La sintaxis de la función ***MPI_Scatter***, es:

MPI_Scatter (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

Sendbuf	*IN*	*Dirección de los datos a ser enviados (choice)*
Sendcount	*IN*	*Número de datos a ser enviados (entero)*
Sendtype	*IN*	*Tipo de dato a ser enviado (manija)*
Recvbuf	*OUT*	*Dirección de almacenamiento de los datos (choice)*
Recvcount	*IN*	*Número de datos a ser recibidos (entero)*
Recvtype	*IN*	*Tipo de dato a ser recibido (manija)*
Root	*IN*	*Rango del proceso raíz de la operación (entero)*
Comm	*IN*	*Comunicador de la operación (manija)*

Empaquetamiento de datos

Con el objetivo de dar mayor flexibilidad y de ser compatible con las ideas de otras bibliotecas como PVM, MPI provee mecanismos para empaquetar datos no continuos para que sean enviados como un buffer continuo y para esparcirlos nuevamente en localidades discontinuas en el proceso receptor. Además, las operaciones de packed y unpacked dan a MPI capacidades que de otra manera no podrían lograrse, como el poder recibir mensaje en partes, donde la operación de recepción dependa del contenido de las partes que forman el mensaje. También estas operaciones facilitan el desarrollo de bibliotecas adicionales de comunicación basadas sobre MPI.

El manejo de packed y unpacked en MPI se logra con las siguientes tres rutinas:

MPI_Pack permite copiar en un buffer (outbuf) continuo de memoria datos almacenados en el buffer (inbuf) de entrada. Cada llamada a la rutina ***MPI_Pack*** modificará el valor del parámetro position incrementándolo según el valor de incount.

MPI_Unpack permite copiar en un (o varios) buffer (outbuf) los datos almacenados en el buffer continuo (inbuf) de entrada. Cada llamada a la rutina ***MPI_Unpack*** modificará el valor del parámetro position incrementándolo según el valor de outcount.

MPI_Pack_size permite encontrar cuanto espacio se necesitará para empaquetar un mensaje y, así, manejar la solicitud de espacio de memoria par el buffer.

Cálculos Colectivos

Las operaciones de cálculos colectivos realizan, entre otras, las operaciones aritméticas suma y producto, además de las operaciones lógicas AND, OR y XOR, entre conjuntos de datos distribuidos a través de los procesos miembros de un grupo. Estas operaciones toman un arreglo de datos de entrada de cada proceso y realizan una operación especifica, operando elemento por elemento de dichos arreglos. La función ***MPI_Reduce*** realiza un cálculo colectivo almacenando los resultados obtenidos en la memoria del proceso raíz de la operación, la sintaxis es:

MPI_Reduce (sendbuf, recvbuf, count, datatype, opt, root, comm)

Sendbuf	*IN*	*Dirección de los datos de entrada (choice)*
Recvbuf	*OUT*	*Dirección de almacenamiento de los resultados (choice)*
Count	*IN*	*Número de datos de la operación (entero)*
Datatype	*IN*	*Tipo de datos de la operación (manija)*
Opt	*IN*	*Operación a ser realizada (manija)*
Root	*IN*	*Rango del proceso raíz de la operación (entero)*
Comm	*IN*	*Comunicador de la operación (manija)*

Esta función toma los count datos del tipo datatype almacenados a partir de la dirección especifica por sendbuf y realiza count operaciones opt, elemento por elemento. Los resultados son almacenados en el proceso root a partir de la dirección recvbuf. Las operaciones posibles para el parámetro opt se especifican a continuación:

Nombre	**Significado**	**Tipos de datos válidos**
MPI_Max	máximo	enteros, reales
MPI_Min	mínimo	enteros, reales
MPI_Sum	suma	enteros, reales y complejos
MPI_Prod	producto	enteros, reales y complejos
MPI_Land	and lógico	enteros y lógicos
MPI_Band	and bit a bit	enteros y byte
MPI_Lor	or lógico	enteros y lógicos
MPI_Bor	or bit a bit	enteros y byte
MPI_Lxor	xor lógico	enteros y lógicos
MPI_Bxor	xor bit a bit	enteros y byte

La función ***MPI_Allreduce*** tiene el mismo objetivo que ***MPI_Reduce***, con la característica adicional de que reproduce en todos los procesos participantes los resultados obtenidos. La sintaxis es la siguiente:

MPI_Allreduce (sendbuf, recvbuf, count, datatype, op, comm)

Sendbuf	*IN*	*Dirección de los datos de entrada (choice)*
Recvbuf	*OUT*	*Dirección de almacenamiento de los resultados (choice)*
Count	*IN*	*Número de datos de la operación (entero)*
Datatype	*IN*	*Tipo de datos de la operación (manija)*
O	*IN*	*Operación a ser realizada (manija)*
Comm	*IN*	*Comunicador de la operación (manija)*