

MODULO: 11

SOFTWARE PARA EL PROCESAMIENTO DISTRIBUIDO

CONTENIDO:

- Conceptos sobre el software del procesamiento distribuido.
- Conceptos sobre Middleware
- Principales características de los S.O. para el procesamiento distribuido.
- Conceptos sobre procesamientos remotos.
- Problemas en el procesamiento distribuido

OBJETIVO DEL MÓDULO: Describir conceptualmente los principios y los fundamentos del software para el procesamiento distribuido y sus problemas en Sistemas de Cómputos.

OBJETIVOS DEL APRENDIZAJE: Después de estudiar este módulo, el alumno deberá estar en condiciones de:

- Explicar y reconocer las características del middleware.
- Comprender y explicar la organización y los métodos utilizados en los Sistemas Distribuidos
- Entender los mecanismos de llamadas a procedimientos remotos y pasaje de mensajes utilizados en los Sistemas Distribuidos.
- Conocer las funciones de un Sistema Operativo Distribuido.
- Comprender los distintos algoritmos utilizados en los Sistemas Distribuidos.
- Comprender los distintos problemas que aparecen en el procesamiento distribuido, como ser sincronización, Deadlocks y los archivos distribuidos.
- Comprender la comunicación entre procesos remotos y la utilización de las transacciones entre ellos.
- Conocer y explicar la terminología específica empleada en éste módulo.

Metas:

En este Módulo presentamos el software que utilizan los sistemas distribuidos, principalmente el S.O. y veremos como se implementan la ejecución de procesos y las transacciones en estos ambientes.

También desarrollaremos conceptos del software en el modelo cliente servidor. En él daremos la definición de **middleware** y su aplicación en los sistemas distribuidos

Luego brindaremos básicamente los conceptos de transacciones entre maquinas, utilizados para permitir el intercambio de datos y los distintos tipos de difusión actualmente implementados.

11.1. El software en los Sistemas Distribuidos:

El software de los sistemas distribuidos constituye la mayor desventaja del uso de estos sistemas. Es un área en proceso de investigación continuo.

El software cumple un rol muy importante cuando hablamos de sistemas distribuidos. Se pueden distinguir dos clases de sistemas operativos para los sistemas compuestos de varias CPUs: *sistemas operativos fuertemente acoplados* y los *débilmente acoplados*.

El *software fuertemente acoplado* es aquel en el que las CPU trabajan de forma tal que la salida de una de ellas es la entrada para la otra, así formando una cadena de entradas y salidas entre las CPUs. De esta forma el acoplamiento es totalmente fuerte.

El *software débilmente acoplado* permite que las máquinas y usuarios de un sistema distribuido sean independientes entre sí en lo fundamental, pero que interactúan en cierto grado, como por ejemplo compartir algunos recursos utilizando las comunicaciones.

Las características de diseño que debe cumplir el software distribuido son:

- Transparencia de:
 - Migración
 - Localización
 - Concurrencia
 - Replicación
 - Paralelismo
- Flexibilidad
- Desempeño
- Escalabilidad

Características de diseño del Software para el Procesamiento Distribuido

Debe existir un mecanismo global de comunicación entre procesos para todo proceso pueda comunicarse con otro. Además debe existir un sistema global de protección. La administración de procesos debe ser igual en todas partes. Entonces es natural que se ejecuten núcleos idénticos en todas las CPUs del sistema. Esto facilita la coordinación de actividades globales. Además se requiere de un sistema global de archivos y que cada núcleo tenga un control considerable de sus propios recursos.

Entonces surgen algunas características necesarias en el diseño del software para el procesamiento distribuido:

Transparencia: El desafío para los diseñadores ha sido ofrecer transparencia en la variedad de procesadores y dispositivos de almacenamiento. La interfase con el usuario de un sistema distribuido transparente no debe generar diferencias al acceder a recursos locales o recursos de la red. Es responsabilidad del sistema operativo distribuido localizar los recursos y generar la interacción adecuada. Una adecuada transparencia debe permitir al usuario movilidad, o sea que pueda utilizar cualquier computadora conectada al sistema, y transportarle su entorno al lugar donde se conecte. Se trata que el sistema sea transparente para los programas, diseñando la interfase de llamadas al sistema de modo que no sea visible la presencia de varios procesadores. El concepto de transparencia se aplica a distintos aspectos al software de un sistema distribuido, principalmente al S.O.:

- **Transparencia de migración:** Los recursos deben poder moverse de ubicación sin cambiar de nombre.
- **Transparencia de localización:** En un verdadero sistema distribuido los usuarios no pueden indicar la localización de los recursos de hardware y software, tales como la CPU, impresora, archivos y bases de datos.
- **Transparencia de réplica:** El sistema operativo puede realizar copias adicionales de archivos y otros recursos sin que lo noten los usuarios.
- **Transparencia con respecto a la concurrencia:** Los usuarios no notarán la presencia de otros usuarios y así compartir recursos de manera automática.

- **Transparencia con respecto al paralelismo:** Un sistema distribuido debe aparecer frente a los usuarios como un sistema tradicional de tiempo compartido con un único procesador, de modo que varias actividades pueden ocurrir en paralelo sin el conocimiento del usuario.

Flexibilidad: Hay dos escuelas diferentes en cuanto a la flexibilidad de los sistemas distribuidos:

- **Modelo con núcleo monolítico:** Ésta escuela sostiene que cada máquina debe ejecutar un núcleo tradicional que brinde la mayoría de los servicios. Es el sistema operativo básico actual, aumentando las capacidades de red y la integración de los servicios remotos. La mayoría de las llamadas al sistema se realizan mediante llamadas al núcleo, que luego regresa el resultado al proceso del usuario. Su ventaja potencial es el rendimiento ya que es más rápido.
- **Modelo micronúcleo:** Sostiene que el núcleo debe brindar lo menos posible y que la mayoría de los servicios del sistema operativo se obtenga de los servidores a nivel de usuario. Es flexible ya que sólo proporciona cuatro servicios mínimos: comunicación entre procesos, cierta administración de memoria, cantidad limitada de planificación y administración de procesos de bajo nivel, y entrada salida de bajo nivel. El micronúcleo no proporciona el sistema de archivos, el sistema de directorios, la administración de procesos ni gran parte del manejo de llamadas al sistema. Todos los demás servicios se implementan como servidores a nivel de usuarios. La ventaja es que este método es altamente modular, existe una interfase bien definida en cada servicio y cada servicio es igual de accesible a todos los clientes. Además es fácil implantar, instalar y depurar nuevos servicios y no requiere el alto total del sistema.

Confiabilidad: La idea es que si alguna máquina falla, alguna otra del sistema se encargue del trabajo. Los datos confiados al sistema no deben perderse o mezclarse. Si hay varias copias de archivos en el sistema, las copias deben ser consistentes. Tiene distintos aspectos:

- **Disponibilidad:** Se refiere a la fracción de tiempo en que se puede utilizar el sistema. La disponibilidad se mejora con un diseño que no exija el uso simultáneo de muchos componentes críticos. Otra herramienta para mejorar la disponibilidad es la redundancia, sea duplicando el software o el hardware, para casos de fallas.
- **Seguridad:** Los archivos y demás recursos deben ser protegidos contra accesos de usuarios no autorizados.
- **Tolerancia a fallas:** Las fallas de comunicación, de la máquina, de los dispositivos de almacenamiento, se consideran fallas que deben tolerarse y así el sistema debe continuar su funcionamiento, pese a las fallas. En general los sistemas distribuidos pueden diseñarse de modo que las fallas se oculten a los usuarios.

Desempeño: La ejecución de una operación en un sistema distribuido no debe ser peor que su ejecución en un único procesador. Se pueden utilizar diversas métricas del desempeño, como tiempo de respuesta, trabajos por hora, uso del sistema y cantidad consumida de la capacidad de la red. El desempeño empeora por ser lenta la comunicación en los sistemas distribuidos en general. Hay que minimizar el número de mensajes o realizar actividades paralelas en diferentes procesadores, que a su vez implica muchos mensajes. Otra solución es hacer remotamente las tareas que implican grandes cálculos y bajas tasas de interacción, y localmente los trabajos con gran número de pequeños cálculos.

Escalabilidad: Es la capacidad de un sistema a adaptarse a un incremento en el servicio. Los sistemas tienen recursos limitados y pueden saturarse al aumentar la carga. En un sistema escalar el rendimiento debe degradarse menos que en un sistema no escalar y sus recursos deben saturarse más tarde que en un no escalar. Un sistema escalar no debe generar problemas al aumentar la cantidad de recursos, dado que es usual aumentar el número de computadoras interconectadas o conectar varias redes.

El problema más importante que se presenta en los sistemas distribuidos es el software: los problemas de compartir datos y recursos es tan complejo que los mecanismos de solución generan mucha **sobrecarga** al sistema haciéndolo ineficiente. El verificar, por ejemplo, quiénes tienen acceso a algunos recursos y quiénes no, el aplicar los mecanismos de protección y registro de permisos consume demasiados recursos. En general, las soluciones que se presentan hoy día para estos problemas están aún lejos de consolidarse como buenas soluciones.

11.2. Middleware

La tendencia más significativa de los sistemas de información en los últimos años ha sido la predominancia de los procesos *cliente/servidor*. Éste tipo de procesamiento está reemplazando a los procesamientos dominados por computadoras centrales, el proceso centralizado y otros tipos de procesamiento distribuido de datos.

El proceso *cliente/servidor* implica dividir una aplicación en tareas y poner cada tarea en la plataforma donde pueda ser manejada más eficazmente. Esto suele significar que la máquina del usuario tenga el proceso necesario para la aplicación y, en el servidor, la gestión y almacenamiento de datos.

La característica central de la arquitectura *cliente/servidor* es la ubicación de las tareas del nivel de aplicación entre clientes y servidores. Para obtener reales beneficios de la estructura *cliente/servidor*, los productores de software y de hardware deben tener herramientas que brinden uniformidad en el acceso a los recursos del sistema en todas las plataformas, permitiendo así la integración y la interoperatividad, ya sea en el interior de una empresa o en el caso de necesitar compartirse datos y programas con otras empresas.

Para ello se utilizan interfases estándares de programación y protocolos que se sitúen entre la aplicación y el software de comunicaciones y sistema operativo. Éstos estándares de interfases y protocolos se denominan *middleware*.

Middleware son interfases estándares de programación que brindan los fabricantes para facilitar que otro proveedor pueda usar el producto en sus aplicaciones. Esto brinda un gran beneficio a los clientes.

El middleware cumple una función muy importante en las arquitecturas cliente/servidor, es el encargado de “dialogar” entre las aplicaciones y el sistema operativo, ocultando la complejidad y las disparidades que tiene los distintos protocolos de red y los distintos sistemas operativos, para ello cuenta con protocolos e interfases de programación estándares las cuales le permiten a los programadores de aplicaciones cliente/servidor poder crear sistemas que permitan el acceso a los datos y a los recursos del sistema en forma transparente. El middleware cumple un rol muy importante especialmente en aquellas arquitecturas three-tier.

Existen muchos paquetes de *middleware*, con la capacidad de ocultar las complejidades y diferencias de los distintos protocolos de red y sistemas operativos. En el *middleware* existen componentes cliente y servidor.

11.2.1. Arquitectura de Middleware

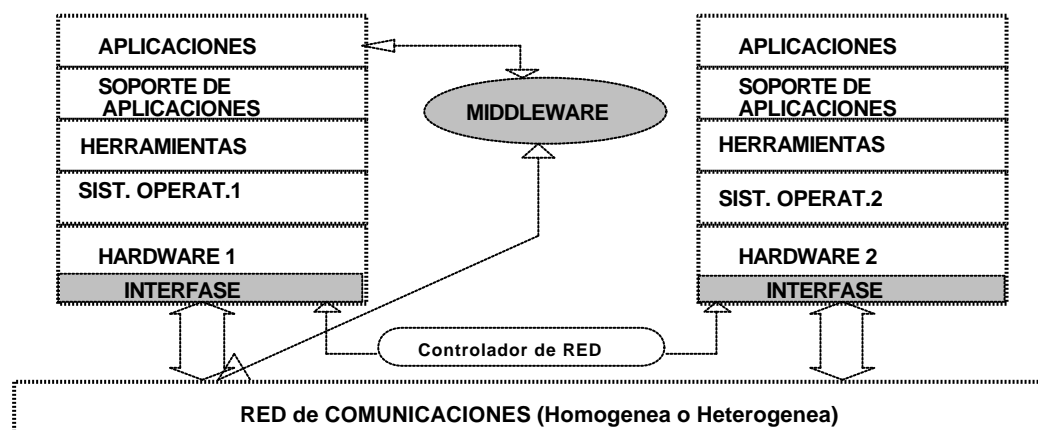


Fig. 11.01 Arquitectura con computadores secuenciales en red usando middleware

La finalidad básica del middleware es hacer que una aplicación o usuario del cliente acceda a una serie de servicios del servidor sin preocuparse de las diferencias entre servidores. Si se considera un área específica de aplicación, se supone que el lenguaje estructurado de consulta (SQL) proporciona una forma estándar de acceder a una base de datos relacional tanto a usuarios o aplicaciones locales como remotos.

Es interesante considerar el papel del middleware desde un punto de vista lógico, más desde la implementación. El middleware permite cumplir todos los objetivos del proceso distribuido. El sistema distribuido entero puede verse como un conjunto de aplicaciones y recursos disponibles para los usuarios.

Aunque hay una amplia variedad de productos de middleware, estos se basan normalmente en uno de estos dos mecanismos básicos: el paso de mensajes o las llamadas a procedimientos remotos (RPC).

Para poder hacerlo interoperativo todo el sistema heterogéneo necesita una interfase que generalmente se llama controlador de comunicaciones o de red.

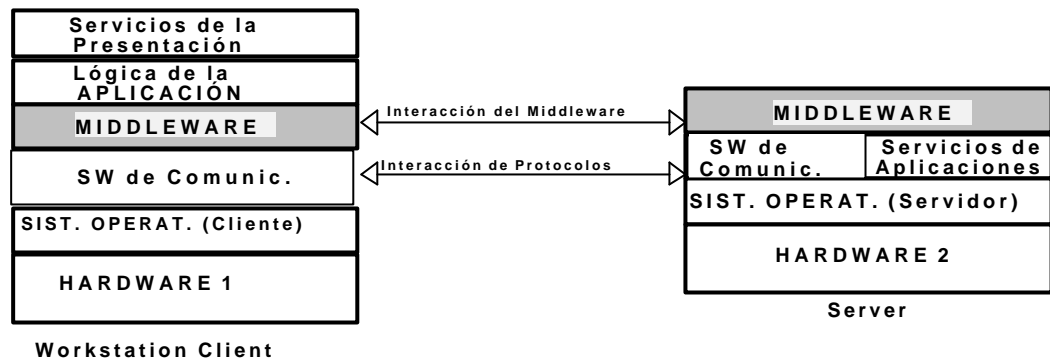


Fig. 11.02 Arquitectura Cliente - Servidor usando middleware

El Controlador debe homogeneizar los Mensajes, las señales, etc.

- **MIDDLEWARE** (en Client-Server Computing): Es una Categoría de software que reside entre una aplicación y la Red cuya principal función es el envío de mensajes, o la organización de sesiones, entre los Nodos de la red para luego ejecutar (entre bastidores) con el fin de proveer datos y conectar las partes.
- Surge como solución a aspectos no cubiertos por los estándares Físico-Lógicos en cuanto al procesamiento distribuido.
- Los se ocupa:
 - Del ruteo apropiado de datos.
 - de la incompatibilidad de las plataformas integrándolas.
 - De ejecuta en cada ambiente.
 - De garantizar la transparencia de accesos a los recursos
 - De utilizar técnicas de Message passing o Remote Procedure Call (RPC) para sus comunicaciones.

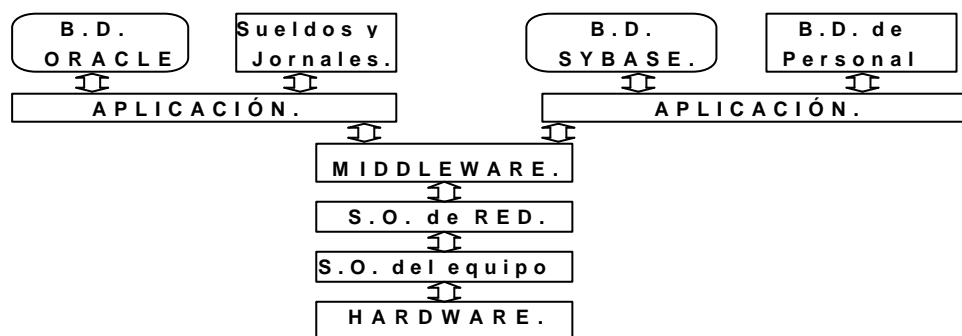


Fig. 11.03. Ejemplo de SQL en dos Bases de Datos propietarias usando middleware

Ejemplo SQL (Structured Query Language) Fig. 11.03:

- Es un standard para acceder a Bases Datos Relacionales.
- Cada Proveedor de Bases de Datos agrega funciones propias como extensiones al SQL.
- Esto crea incompatibilidad entre distintos proveedores.
- Los distintos proveedores generan dificultad en la integración, y en la interoperatividad (ya sea dentro de la empresa o entre empresas que deseen compartir datos y programas).
- El Middleware es una interfase estandarizada de programación que es aceptada por los distintos proveedores (estos facilitan sus interfaces en sus productos) para que otro proveedor pueda usar el producto en sus aplicaciones.
- El Middleware homogeneiza la disparidad de protocolos de redes y los distintos S.O.

11.2.2. Aspectos lógicos del Middleware:

Dado que un sistema cliente/servidor trabaja sobre una red se deben definir protocolos para lograr conectividad que permita a programas o procesos comunicarse en forma transparente. Para ello se deben definir los protocolos que el middleware debe soportar.

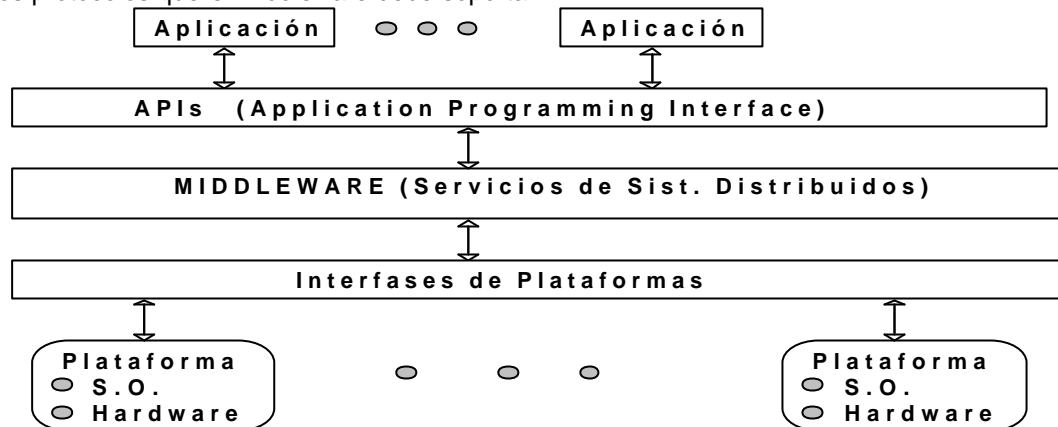


Fig. 11.04 Aspecto lógico del middleware

Los protocolos se dividen en tres grupos: de medios, de transporte y protocolos cliente/servidor.

- Los protocolos de medios determinan el tipo de conexión física usada en la red ej: Ethernet, Token Ring, Fiber Distributed Data Interface (FDDI).
- Los protocolos de transporte proveen los mecanismos para mover paquetes de datos desde el cliente al servidor o viceversa ej: IPX/SPX de Novell, AppleTalk de Apple, Protocolo de Internet TCP/IP.

Una vez que la conexión física se ha establecido y se cuenta con una conexión de transporte, resta por definir el protocolo cliente/servidor para que el usuario pueda acceder a los servicios de la red.

- Un protocolo cliente/servidor dicta la manera en que los clientes requieren la información al servidor y la manera en que el servidor le responde a esos requerimientos. Ej: NetBIOS, RPC, Advanced Program-to-Program Communication (APPC), Named Pipes, Transport Level Interface (TLI) o Sequenced Packet Exchange (SPX)).

11.2.3. Servicios de Middleware

Aquí se presenta un enfoque donde se define tres niveles de funciones para el middleware, básicas, intermedias, avanzadas.

Servicios Básicos

Los servicios básicos son un mínimo nivel de funciones que se deben esperar de una arquitectura middleware. Una característica clave de estos servicios es que deben proveer transparencia, en otras palabras que se invisible al usuario.

- **Diferentes protocolos:** A bajo nivel esto incluye tecnologías como IPX/SPX y TCP/IP. El Middleware debe proveer soporte para una cantidad importante de protocolos para cubrir los actuales y futuros estándares.
- **Diferencias en TCP/Ips:** Solo decir que se soporta TCP/IP no es suficiente porque hay por lo menos 15 variantes de este "estándar". El middleware necesita ser capaz de operar sobre todas o la mayoría de estas implementaciones.
- **Traslación de Protocolos:** Cuando parte de la red de una empresa opera con un protocolo y otra parte lo hace con otros, los mensajes tendrán que pasar por múltiples protocolos sin problema.

Servicios de conectividad y Acceso a datos

- **Conectividad:** Este es generalmente el punto clave del middleware en una arquitectura cliente/servidor. Hay un numero de propiedades estándar de APIs que pueden ser usadas para establecer conectividad. Estas APIs pueden ser de propósito general o orientadas a SQL y de hecho, estándares basado en objetos como OLE o DSOM y sus procesos de mensajes también pueden ser usados para establecer conectividad. Un producto middleware debe soportar estándares comunes en este área como ODBC, DBLib, OLI, DRDA, SQL/API y X/Open.
- **Optimización de Consultas (Query):** Para acceso a DBMS distribuido. Cuando un JOIN requiere de datos que están ubicados en lugares distintos, el middleware debe proveer inteligencia para navegación para completar el Query. Además en referencia a la navegación distribuida, la

existencia de diferentes estructuras de archivos y esquemas de índices en varios sitios se requiere un enfoque inteligente para evitar costos en la ejecución del Query. Obviamente la lógica del middleware debe trabajar en forma relacional, no relacional, estructura de archivos plano u orientado a objeto.

- **Llamadas Procedimiento Remoto (RPC):** Diferentes motores de DBMS soportan diferentes formas de procedimientos remotos. Además hay otras formas de procedimientos remotos tales como OSF, DCE que el middleware debe soportar sin problemas.

Servicios de Programación (Scheduling)

- **Manejo de Hilos:** Proveer una capacidad de explotar comunicaciones cruce de proceso (cross-process) y sistemas basados en transacciones seguras, tales como CICS¹ o IMS/DS. Estos permiten el manejo de múltiples procesos simultáneamente. Ya que en diferentes entornos el manejo de estas funciones difiere, el middleware debe enmascarar estas diferencias, haciendo mas fácil el diseño de aplicaciones que puedan correr bien en los entornos cliente/servidor.
- **EL Balance de Carga:** Puede o no ser soportada por entornos operativos (como el caso de sistemas paralelos), el middleware debe proveer habilidad para cumplir esta función.
- **Seteo de Prioridad:** El middleware debe brindar facilidades para permitir que algunas tareas se ejecuten como “privadas” o como compartidas.

Servicios Intermedios

Posiblemente algunos de los servicios que se presentan como de categoría intermedia podrían pertenecer a servicios avanzados dado que no hay una línea definida para ello.

Servicios de Seguridad

- Cada entorno operativo puede tener distintos mecanismo de seguridad que difieren entre si como controles de login o productos de seguridad separados como RACF o Top Secret, también administradores de Base de Datos pueden tener restricciones de seguridad. El middleware debe manejar múltiples entornos de seguridad ofreciendo una interfase heterogénea para los usuarios.
- El uso de “recursos confiables” permite el mapeo de IDs auténticos dentro del sistema. Por ejemplo un ID valido puede mapear a alguien en un Sistema Digital, eliminando que el usuario necesite distintas passwords para cada subsistema.

11.2.4. Estrategia básica de interoperabilidad

El objetivo que se persigue cuando se necesita incorporar al nodo existente una serie de Productos nuevos sea hardware o software en las condiciones que se indican en la Fig. 11.05, entonces se plantean los siguientes PROBLEMAS:

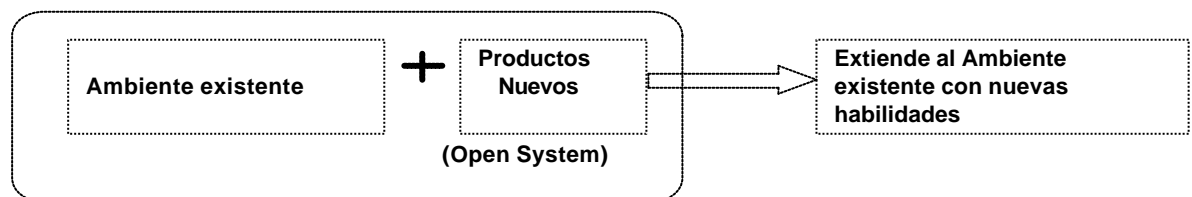


Fig. 11.05 Estrategias de ampliación de productos en una nodo

- Solapamiento con funciones existentes en los sistemas instalados.
- Generalmente incompatibilidad tecnológica
- Proveer compatibilidad de productos.
- Estrategias de Venta de los Proveedores que generalmente aseguran compatibilidad de sus productos que no existe (ni a nivel de Software, ni a nivel de Hardware). Ejemplo: print-server para OS/2, UNIX, Decnet, etc. Todos son PROPIETARIOS, entonces se producen problemas de Licencias.
- Compatibilidad de códigos...
- Problemas de protección.
- Portabilidad.

¹ CICS (Customer Information Control System) es un sistema de control de información al cliente de IBM

- Integración.
- Capacitación tanto la Empresa proveedora como a los Usuarios.
- Documentación.
- etc., etc.

De todas formas hay proveedores de middleware que resuelven la mayor parte de estos problemas. En general la metodología que se sigue para resolver el problema de la Interoperabilidad de sistemas existentes es:

- Se establece mediante interfases.
- La información se mueve de un ambiente a otro mediante mecanismos de Comunicación normalizados.

11.3. SOFTWARE DISTRIBUIDO

Un sistema operativo distribuido proporciona a los usuarios acceso a los distintos recursos que ofrece el sistema. El acceso a estos recursos lo controla el sistema operativo. Existen dos esquemas básicos complementarios para proporcionar este servicio:

Software de comunicaciones en red o S.O. de red: Los usuarios están enterados de la multiplicidad de máquinas y para el acceso a estos recursos necesita conectarse a la máquina remota apropiada o transferir datos de la máquina remota a la propia.

Sistemas operativos distribuidos: Los usuarios no necesitan saber de la multiplicidad de máquinas y pueden acceder a los recursos remotos de la misma manera que lo hacen para los recursos locales (Visión de una máquina virtual).

11.3.1. Software de Comunicaciones (Sistemas operativos de red)

La principal función de un Software de comunicaciones (comúnmente llamado sistema operativo de red) es ofrecer un mecanismo para transferir archivos de una máquina a otra. En este entorno, cada nodo mantiene su propio sistema de archivos local y si un usuario de el nodo A quiere acceder a un archivo de el nodo B, hay que copiar explícitamente el archivo de una nodo a otra. En este esquema, la ubicación del archivo no es transparente para el usuario; tienen que saber exactamente donde está el archivo. Además, los archivos no se comparten realmente porque un usuario solo puede copiar un archivo de una nodo a otra. Por lo tanto, pueden existir varias copias del mismo archivo, lo que representa un desperdicio de espacio y un serio problema de consistencia de los datos.

Esta forma es primitiva en extremo y provoca que los diseñadores busquen formas mas convenientes de comunicación y distribución de la información. Un método consiste en proporcionar un sistema de archivos global compartido, accesible desde todas las estaciones de trabajo.

Una o varias máquinas, llamadas *servidores de archivos*, soportan al sistema de archivos. Los servidores aceptan solicitudes de escritura y lectura de archivos de los programas de usuarios, los cuales se ejecutan en máquinas llamadas *clientes*. Cada una de las solicitudes que llegan al servidor se examina, se ejecuta y la respuesta es enviada de regreso, como se muestra en la figura 11.06.

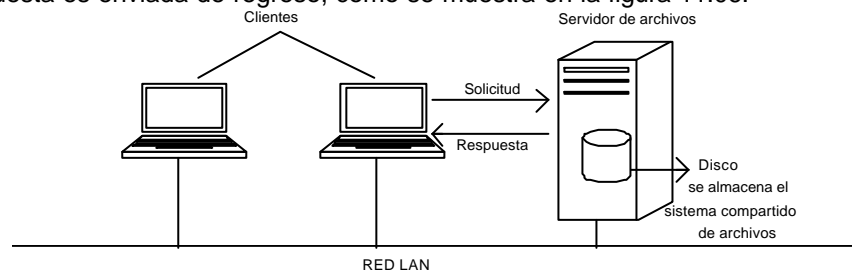


Fig. 11.06. Dos clientes y un servidor en un sistema operativo de red.

Los servidores de archivos tienen, por lo general, un sistema jerárquico de archivos, cada uno de los cuales tienen un directorio raíz, con subdirectorios y archivos. Las estaciones de trabajo (clientes) pueden importar estos sistemas de archivos, lo que aumenta sus sistemas locales de archivos con aquellos localizados en los servidores (Duplicando la información).

El sistema operativo a utilizar en este tipo de ambiente debe controlar las estaciones de trabajo en lo individual, a los servidores de archivos y también debe encargarse de la comunicación entre ellos.

Existe la posibilidad de que todas las máquinas ejecuten el mismo sistema operativo, pero no es condición necesaria. En el caso que ejecuten distintos sistemas operativos, deben coincidir en el formato y significado de todos los mensajes que podrán ejecutar.

Uno de los mejores sistemas operativos de red es el *Network File system* de Sun Microsystems, conocido como *NFS*. NFS soporta sistemas heterogéneos; por ejemplo clientes de MS-DOS que hagan uso de servidores UNIX y no necesita que las máquinas utilicen el mismo hardware.

El NFS es un ejemplo de software débilmente acoplado en hardware débilmente acoplado

11.3.2. Sistemas Operativos Distribuidos (SOD)

Otra combinación para analizar es el software fuertemente acoplado en hardware débilmente acoplado. El objetivo de un sistema de este tipo es crear la ilusión en los usuarios que toda la red de computadoras es un solo sistema de tiempo compartido, en vez de una colección de máquinas diversas.

Muchos autores se refieren a esta propiedad como la imagen de un único sistema. Otros proponen que un sistema distribuido es aquel que se ejecuta en una colección de máquinas enlazadas mediante una red pero que actúan como un uniprocador virtual.

La idea esencial es que los usuarios no deben ser conscientes de la existencia de varias CPUs en el sistema. Es por ello que analizaremos cuales son las características de un sistema distribuido.

Características de un S.O. Distribuido:

- **Un mecanismo de comunicación global estandarizado entre los procesos:** En primera instancia, debe existir un mecanismo de comunicación global entre los procesos, de forma que cualquier proceso pueda hablar con cualquier otro. No tienen que haber distintos mecanismo en distintas máquinas o distintos mecanismos para la comunicación local o remota.
- **Un esquema global de protección:** También debe existir un esquema global de protección. La mezcla de acceso a las listas de control, los bits de protección de UNIX y las diversas capacidades no producirán la imagen de un único sistema.
- **La administración de procesos unificada:** La administración de procesos debe ser la misma en todas partes. La forma en que se destruyen, inician y detienen los procesos no debe variar de una máquina a otra.
- **Conjunto estandarizado de llamadas al sistema:** No solo debe existir un único conjunto de llamadas al sistema disponible en todas las máquinas, sino que estas llamadas deben ser diseñadas de manera que tengan sentido en un ambiente distribuido. Como consecuencia del hecho de tener una misma interfase de llamadas al sistema en todas partes es normal que se ejecuten núcleos (Kernels) idénticos en todas las CPUs del sistema. Esto facilita la coordinación de las actividades globales del sistema. Por ejemplo, cuando se deba iniciar un proceso, todos los núcleos deben cooperar en la búsqueda del mejor lugar para ejecutarlo.
- **Accesos Transparente a recursos remotos:** En un sistema operativo distribuido los usuarios pueden acceder a recursos remotos de la misma manera que lo hacen para recursos locales. La migración de datos y procesos de una nodo a otra queda bajo el control del sistema operativo distribuido.

Migración de Datos, Cálculos y Procesos:

En el procesamiento distribuido hay tres tipo de migración: de datos, de cálculos y de procesos. Cada uno tiene sus propias motivaciones. Veremos cada una de ellas.

Migración de datos

Supongamos que un usuario en el nodo A quiere acceder a datos que residen en el nodo B. Existen dos métodos básicos para que el sistema transfiera los datos. Una estrategia es transmitir todo el archivo al nodo A, y a partir de ese momento todo acceso al archivo es local. Cuando el usuario ya no requiere acceso al archivo, se envía de regreso una copia al nodo B.

El otro enfoque consiste en transferir al nodo A solo las porciones del archivo que en ese momento realmente son necesarias para la tarea actual. Si mas tarde se requiere otra porción, se efectuara otra

transferencia y, cuando el usuario ya no quiera acceder al archivo, todas las porciones modificadas se deberán mandar de regreso a B.

El sistema también debe realizar varias traducciones de los datos si la transferencia implica a dos instalaciones que no son directamente compatibles.

Migraciones de cálculos

En ciertas circunstancias puede ser mas eficiente transmitir los cálculos por el sistema en vez de los datos.

Hay varias maneras de transmitir los cálculos. Supongamos que el proceso **p** quiere acceder a un archivo en el nodo A. El acceso al archivo se lleva a cabo en el nodo A y podría iniciarse mediante una llamada a un procedimiento remoto. El proceso **p** invoca un procedimiento predefinido de el nodo A, el cual se ejecuta y devuelve a **p** los parámetros necesarios.

Como alternativa, el proceso **p** puede enviar un mensaje al nodo A. El sistema operativo en A crea un nuevo proceso **q** cuya función es realizar la tarea especificada; cuando **q** termina su ejecución, envía a **p** el resultado por medio del sistema de mensajes. Obsérvese que en este esquema los procesos **p** y **q** pueden ejecutarse concurrentemente y, de hecho, puede haber varios procesos concurrentes en distintas instalaciones.

Migración de procesos

Definimos como migración de un proceso en procesamiento distribuido a la transferencia de un proceso o parte de un proceso de una máquina a otra, para que pueda ejecutarse en la máquina final.

La migración de procesos es la transferencia de una parte suficiente del estado de un proceso desde una máquina a otra para que el proceso se pueda ejecutarse en la máquina de destino. El interés en este concepto surgió de la investigación sobre formas de equilibrar la carga entre varios sistemas en red, aunque la aplicación del concepto se extiende actualmente más allá de este campo.

Motivaciones:

La migración de procesos es deseable en los sistemas distribuidos por una serie de razones, incluyendo a las siguientes:

- 1) **Compartir y Balance de la carga:** Trasladando los procesos desde sistemas muy sobrecargados hacia otros menos cargados, la carga puede equilibrarse y así mejorar el rendimiento global.
- 2) **Performance de la comunicación:** Los procesos que interactúan de forma intensiva pueden moverse a un mismo nodo para reducir el coste de las comunicaciones durante su interacción.
 - a) Dos procesos que se comunican intensivamente conviene que estén en la misma máquina.
 - b) Si un proceso es más chico que la información que está consultando remotamente, es preferible mover el proceso que la información.
- 3) **Disponibilidad (Availability):** El Proceso que quiere sobrevivir a la planificación puede migrar. Los procesos que duran mucho tiempo pueden necesitar moverse para sobrevivir frente a los fallos de los que se pueda obtener previo aviso o en prevención del plazo planificado. Si el sistema operativo proporciona tal aviso, un proceso que desea continuar puede emigrar a otro sistema o bien asegurarse de que se reanudará más tarde en el sistema actual.
- 4) **Velocidad de ejecución (Computation SpeedUp):** Se puede dividir un proceso en Subprocesos que corren concurrentemente en diferentes máquinas.
- 5) **Uso de capacidades especiales:** significa migrar para hacer uso de Software o Hardware únicos en cierto nodo. Un proceso puede trasladarse para sacar partido de algunas capacidades de hardware o software disponibles en un nodo determinado.

Cuando un proceso es migrado es necesario **destruir el proceso en la fuente del sistema y crearlo en el otro sistema**. Esto es un movimiento de parámetros del proceso, no una replicación. Por lo tanto, la imagen del proceso, consistiendo al menos que una parte del bloque de control del proceso (PCB), debe ser movido. Además, cualquier conexión (link) entre este proceso y otros procesos, como por ejemplo el paso de mensajes y señales, debe ser actualizado.

Existen varias estrategias para mover el contenido parcial del PCB:

➤ **Intenso (todos):** Transfiere el espacio de dirección entero en el momento de la migración. Esta forma es la más clara y limpia porque no se deja ninguna pista del proceso que será necesaria usar del viejo sistema. Sin embargo, si la dirección de espacio es muy grande y si el proceso parece no tener que necesitar la mayoría de ésta, entonces esto puede ser innecesariamente caro. El costo inicial de la migración debe ser en el orden de algunos segundos.

➤ **Precopia:** El proceso continúa ejecutando en el nodo fuente mientras el espacio de dirección es copiado a otro nodo. Las páginas modificadas en el nodo fuente durante la operación de la precopia deben ser copiadas una segunda vez. Esta estrategia reduce el tiempo que un proceso está congelado y no puede ejecutar durante la migración.

➤ **Intenso (sucio):** Transfiere sólo aquellas páginas del espacio de dirección que están en memoria central y fueron modificados. Cualquier otro bloque adicional en el espacio de dirección virtual será transferido en la demanda solamente. Esto minimiza la cantidad de datos que serán transferidos. Requiere, sin embargo, que la fuente de la máquina continúe para ser implicada en la vida del proceso manteniendo la página o la tabla de entrada de segmentos y requiere soporte de paginación remota.

➤ **Copia por referencia:** Esta es una variación de la anterior en la cual las páginas son traídas sólo cuando son referenciadas. Esto tiene el costo inicial más bajo de la migración de procesos, tardando desde un poco tiempo a unos pocos cientos de miles de microsegundos.

➤ **Mover (flushing):** Las páginas de los procesos son limpiadas desde la memoria central de la fuente moviendo las páginas viejas al disco. Después se realiza una copia por referencia. Esta estrategia mitiga la fuente de la necesidad de mantener cualquier página del proceso migrado en memoria central inmediatamente liberando un bloque de memoria para ser usado por otro proceso.

Mecanismo de la migración de procesos:

Se deben realizar tres determinaciones básicas:

- 1) Quién inicia la migración?
- 2) Qué porción de proceso se migra?
- 3) Qué pasa con los Mensajes y las Señales?

1) Quién inicia la migración?

Depende del objetivo:

— Si es por compartir cargas (Load Sharing) o Velocidad de procesamiento (SpeedUp), quien inicia la migración lo hace el S.O.

- Algún módulo del S.O. monitorea la migración:
- Es responsable de **expropiar** o señalar el proceso a ser migrado.
- Se comunica con módulo análogo en el otro Sistema.

— Si cierto proceso quiere alcanzar cierto recurso, él inicia la migración.

Quién inicia la migración dependerá del objetivo del servicio de migración. Si el objetivo es equilibrar la carga, algún módulo supervisor de la carga del sistema operativo es normalmente el responsable de decidir cuándo tendrá lugar la migración. Este módulo es responsable de expulsar o indicar a un proceso que va a emigrar. Para determinar dónde va a emigrarse, el módulo necesita estar en comunicación con módulos similares de otros sistemas, de forma que se pueda supervisar la composición de la carga de otros sistemas. Si el objetivo es llegar hasta unos recursos determinados, el proceso puede emigrar por sí mismo cuando surja la necesidad

2) Qué porción de proceso se migra?

Cuando un proceso emigra, hace falta destruirlo en el sistema de origen y crearlo en el sistema de destino. Esto es un movimiento de procesos y no una duplicación. Por tanto, debe moverse la imagen del proceso, que consta de por lo menos, el bloque de control del proceso. Además, debe actualizarse cualquier enlace entre éste y otros procesos, como los de paso de mensajes y señales.

El movimiento del bloque de control del proceso es sencillo. Desde el punto de vista del rendimiento, la dificultad estriba en el espacio de direcciones del proceso y en los archivos abiertos que tengan asignados

— Cuando se migra proceso:

- Se destruye en el origen
- Se crea en el destino
 - Es un MOVIMIENTO de parámetros del proceso y no una Réplica del PCB.
- Se tienen que mantener todos los links de mensajes y señales con otros procesos.
 - Proceso 3 de A migra a B pasando a ser P4 (Ver Fig. 11.7b).
 - Todos los Links se mantienen.

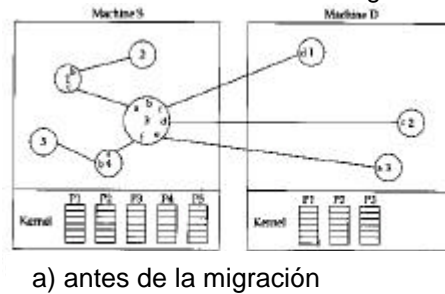
→ Responsabilidad del S.O.

- Mover PCB
- Mantener Links

→ Migración es invisible para:

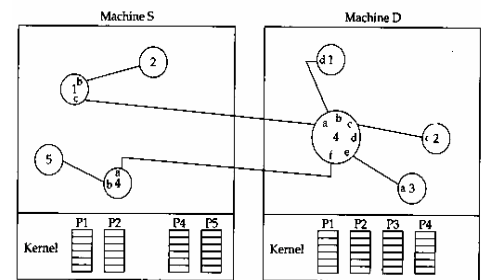
- Procesos migrados.

- Procesos comunicados con Proceso Migrado.



a) antes de la migración

Fig. 10.7 migración de un proceso



b) el proceso se ha migrado

Se presentan algunos problemas de la migración del

1) Espacio de Direccionamiento:

Suponiendo memoria virtual (paginación o segmentación): dos estrategias.

- a) Transferir todo el espacio de direccionamiento en el momento de la migración.

Si el espacio de direccionamiento es grande y solo se necesita una parte esto es CARO.

- b) Transferir solo lo que está en Memoria Central.

El resto se transfiere por demanda.

⇒ La máquina de origen siga pendiente del proceso (mientras viva) para la paginación o la segmentación.

2) Archivos asignados al Proceso Migrado:

- Si va a seguir accediendo al Archivo.
Y es el único, conviene migrar el Archivo también.
- Si el proceso va y vuelve en seguida, conviene dejar el Archivo o solo moverlo por un requisito.

3) Qué pasa con los Mensajes y las Señales?

El destino de los mensajes y las señales pendientes, se puede tratar mediante un mecanismo de almacenamiento temporal, durante la migración, de los mensajes y señales pendientes para, posteriormente, dirigirlos a su nuevo destino. Puede hacer falta mantener una información de desvío en el emplazamiento inicial durante algún tiempo, para asegurar que llegan todos los mensajes y las señales pendientes.

— Los Mensajes que lleguen para el proceso durante la migración son guardados, por el sistema temporalmente y después reenviados.

→ Ejemplo de Self Migration: AIX (IBM) UNIX Distribuido.

1) P decide migrarse ⇒ elige una máquina de destino: entonces: Manda un mensaje con IMAGE del Proceso y Archivo con Información.

2) En la máquina destino: el Kernel crea un hijo dándole Información.

3) El Hijo toma datos, contexto, argumento stack... necesarios para completar la operación.

4) Proceso Original:

- Es suspendido durante la migración
- Es señalado de que la migración terminó.
- Proceso se destruye.

Negociación De Migración:

Otro aspecto de la migración de procesos está relacionado con la decisión de emigrar. En algunos casos, la decisión la toma una única entidad. Por ejemplo, si se pretende equilibrar la carga, un módulo supervisará la carga relativa de varias máquinas y llevará a cabo la migración, si es necesaria, para mantener la carga equilibrada. Si se usa automigración (Self migration) para hacer que los procesos accedan a servicios especiales o grandes archivos remotos, el mismo proceso puede tomar la decisión. Sin embargo, algunos sistemas permiten que el destino designado participe en la decisión, entre otras cosas para conservar el tiempo de respuesta a los usuarios.

→ Política de migración la lleva el STARTER:

Cuando migrar?
Que proceso?
En que Máquina?

STARTER: Un proceso que es responsable en una o más máquinas de:

- Política de migración.
- Long Term Scheduling.
- Memory allocation.

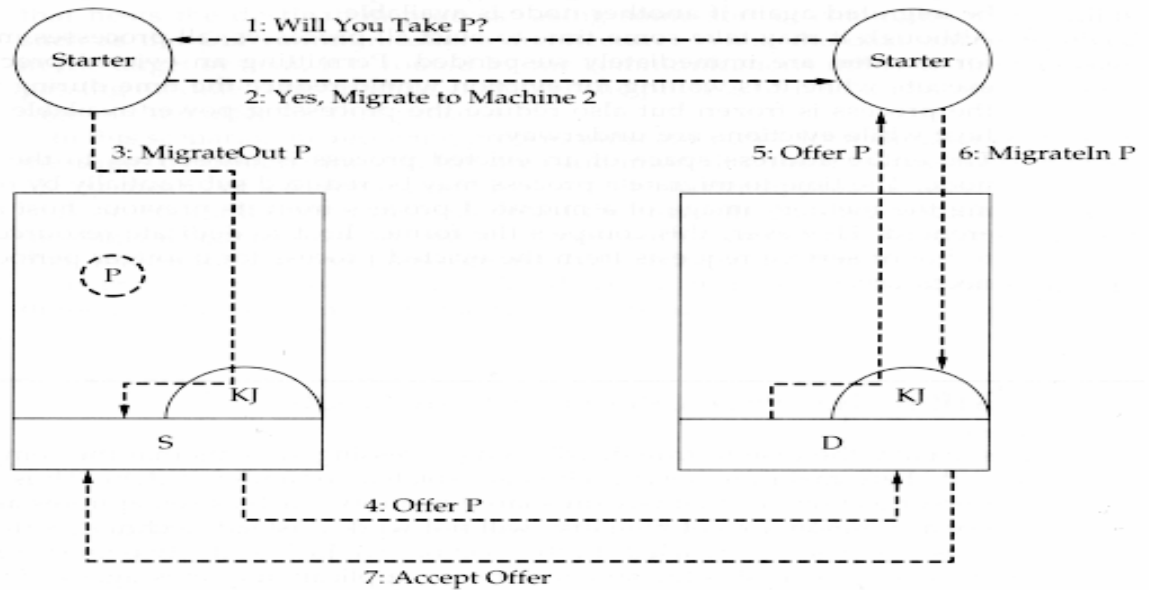


Fig.11.08. Migración de procesos

- 1) STARTER de S decide migrar P a Sys. D
⇒ Manda Mensaje a STARTER de D pidiendo transferencia.
- 2) STARTER de D está listo para recibirlo
⇒ Manda Acknowledge.
- 3) STARTER de S comunica la decisión a su Kernel (por un Service call o Mensaje a KJ).
↓
Proceso que convierte mensajes de procesos remotos en SysCall.
- 4) Kernel de S ofrece a P con ciertos parámetros para la ejecución.
- 5) Si D no tiene Recursos, puede rebotar el ofrecimiento.
SINO Kernel de D le pasa ofrecimiento al STARTER de D con los parámetros.
- 6) La decisión del STARTER de D es comunicada a D con MigIn(P).
- 7) D reserva recursos (para Deadlock y Flow Control) y acepta ofrecimiento.

Rechazo de una migración: El proceso de negociación permite que un sistema de destino rechace la inmigración de un proceso. Además, puede ser útil hacer que un sistema desaloje un proceso que ha emigrado hacia él. Por ejemplo, si un puesto de trabajo se activa, puede hacer falta desalojar los procesos inmigrados para ofrecer un tiempo de respuesta apropiado.

Transferencias expropiativas y No expropiativas

La **migración expropiativa** de procesos, que consiste en la transferencia de un proceso parcialmente ejecutado o, al menos, cuya creación se haya completado. Una función más simple es la **transferencia no expropiativa**, que involucra solamente a procesos que no han comenzado su ejecución y, por tanto, no se necesita transferir el estado del proceso. En ambos casos debe transferirse información al nodo remoto sobre el entorno de ejecución del proceso. Esta puede incluir el directorio de trabajo actual del usuario y los privilegios heredados por el proceso.

Básicamente existen dos técnicas complementarias que se emplean para mover procesos en una red. El sistema puede intentar ocultar el hecho de que el proceso se ha movido del lugar donde está el cliente. Este esquema tiene la ventaja de que el usuario no necesita codificar su programa explícitamente para lograr la migración, y generalmente se emplean para equilibrar las cargas y acelerar los cálculos entre sistemas homogéneos.

El otro enfoque consiste en permitir (o requerir) que el usuario especifique explícitamente como debe migrar el proceso. Por lo general este enfoque se utiliza cuando el proceso se debe mover para cumplir requerimientos de hardware o de software.

Un sistema distribuido es aquel que se ejecuta en una colección de máquinas sin memoria compartida, pero que aparece ante sus usuarios como una sola computadora.

No importa la forma en que se exprese, la idea esencial es que los usuarios no deben ser conscientes de la existencia de varios procesadores en el sistema.

En particular, este tema lo ampliaremos mas adelante dado que es una de las tareas mas importantes que debe soportar un S.O.

Comparación de los 3 últimos métodos:

El siguiente cuadro (tabla 11.1) muestra algunas diferencias entre los tres tipos de sistemas analizados.

Elemento	S. O. de red.	S.O. Distribuido	S. O. de multiprocesador
¿Se ve como un procesador virtual?	No	Si	Si
¿Todos tienen que ejecutar el mismo S.O?	No	Si	Si
¿Cuántas copias del S.O existen?	n	n	1
¿Como se logra la comunicación ?	Archivos compartidos.	Mensajes.	Memoria compartida.
¿Se requiere un acuerdo en los protocolos de la red?	Si	Si	No
¿Existe una única cola de ejecución?	No	No	Si(*)
¿Existe una semántica bien definida para los archivos compartidos?	Por lo general no.	Si	Si

(*) Si los procesadores son idénticos o sea homogéneos.

Tabla 11.1 Comparación de las tres formas distintas de organizar n procesadores.

El S.O.D. debe cumplir con los requisitos de diseño del Software distribuido. Veamos algunas de las características mas importantes:

TRANSPARENCIA

Se necesita ofrecer al usuario transparencia en el uso y existencia de multiplicidad de recursos como ser: procesadores, dispositivos de almacenamiento, impresión, captura de datos, etc..

El shell, o sea, la interfase con el usuario de un sistema operativo distribuido no debe hacer distinciones entre recursos locales y remotos; es decir, los usuarios deben ser capaces de acceder a los nodos remotos como si fueran locales y debe ser responsabilidad del sistema operativo localizar los recursos y hacer los arreglos para una interacción adecuada.

El concepto de transparencia se puede aplicar a varios aspectos de un sistema operativo distribuido como se muestra en la tabla 11.2.

Tipo de Transparencia	Significado
de localización.	Los usuarios no pueden indicar la localización de los recursos.
de migración.	Los recursos se pueden mover a voluntad sin cambiar sus nombres.
de replica.	Los usuarios no pueden indicar el numero de copias existentes.
de concurrencia.	Varios usuarios pueden compartir recursos de manera automática.
de paralelismo	Las actividades pueden ocurrir en paralelo sin el conocimiento de los usuarios.

Tabla 11.2. Distintos tipos de *transparencia* en un sistema distribuido.

FLEXIBILIDAD

Existen dos modelos de SOD. Estos dos modelos son conocidos como kernel *monolítico* y *microkernel*, respectivamente. (Fig. 11.09).

La diferencia son visiones de la evolución del procesamiento. Hoy día, se impone el microkernel en el modelo cliente servidor por ser mas flexible, pues tiene muy pocas tareas. En términos básicos, proporciona solo cuatro servicios mínimos:

- Un mecanismo de comunicación entre procesos.
- Cierta administración de memoria.
- Una cantidad limitada de planificación y administración de procesos de bajo nivel.
- Entrada/Salida de bajo nivel.

Los restantes servicios que debería prestar el micronúcleo esta concebido como servidores a nivel usuario que son mas eficientes en el procesamiento local del servicio. El usuario que requiere un servicio, envía un mensaje al servidor apropiado, el cual realiza el trabajo y regresa el resultado. Además, cada servicio es igual de accesible para todos los clientes y es fácil de implantar, instalar y depurar los nuevos servicios que se requieran. Todo esto debido a la gran reducción de costos del hardware y el aumento de la velocidad de las computaciones.

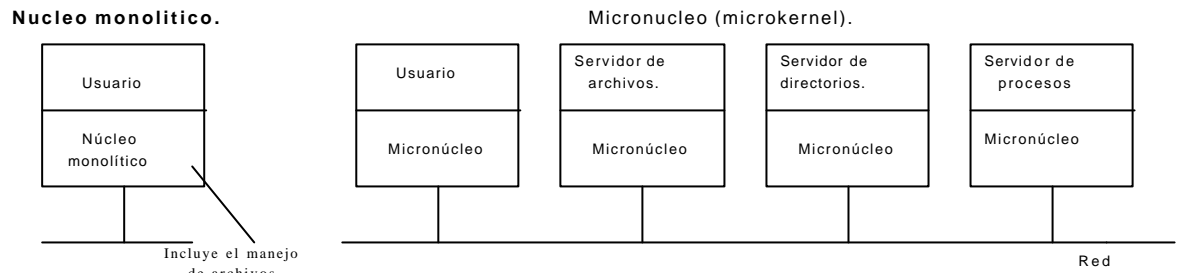


Fig. 11.09. Nucleo monolítico y Micronúcleo.

Es precisamente esta capacidad de añadir, eliminar y modificar servicios lo que le da al micronúcleo su *flexibilidad*.

La única ventaja potencial de los núcleos monolíticos es el rendimiento. Las llamadas al núcleo y la realización de todo el trabajo puede ser mas rápido que el envío de mensajes a los servidores remotos y esperar sus respuestas.

CONFIABILIDAD

La idea es que si una máquina presenta durante su ejecución, errores, defectos o fallas, el S.O. debe disponer de suficiente habilidad para que otra maquina del sistema se encargue del trabajo en cuestión en forma transparente para el usuario. Para ello se requiere que los procesos, sus datos y argumentos migren en forma segura a una de las maquinas disponibles que pueda continuar con el procesamiento.

DESEMPEÑO

Uno de los problemas que presenta el procesamiento distribuido es desempeño. Se pueden recurrir a diversas métricas para determinar el desempeño, como por ejemplo: el tiempo de respuesta, el rendimiento (números de trabajos por hora), el uso del sistema y la cantidad consumida de la capacidad de la red o el balance de la carga de trabajo de cada nodo.

Generalmente se presentan demoras en el intercambio de mensajes, por lo que es aconsejable disminuirlo al mínimo necesario para optimizar el desempeño.

ESCALABILIDAD

A la capacidad de un sistema para adaptarse a un incremento en el servicio se le llama **capacidad de ampliación o escalabilidad**. Los sistemas tienen recursos limitados y pueden saturarse al aumentar su carga, pero deberían adaptarse a un incremento en el servicio dado que una carga de trabajos crece continuamente..

El problema puede resolverse añadiendo nuevos recursos, pero se puede generar una carga indirecta adicional en otros recursos o, lo que es peor, la ampliación del sistema puede representar costosas modificaciones en el diseño.

Un sistema escalar debe tener el potencial para crecer sin estos problemas. Esta habilidad para crecer con modularidad es de suma importancia en un sistema distribuido, pues es habitual ampliar la red añadiendo nuevas máquinas o interconectando dos redes.

La tolerancia a fallas y la escalabilidad están relacionadas. Un componente con mucha carga puede paralizarse y comportarse como si tuviera una falla. Generalmente es esencial contar con recursos de repuesto para asegurar la confiabilidad y para manejar con suavidad las cargas de trabajo pico.

Las consideraciones de tolerancia a fallas y capacidad de escalabilidad requieren un diseño que presente la distribución del control y los datos.

En gran medida, los sistemas distribuidos a muy gran escala son solo teóricos. No existen líneas maestras mágicas que aseguren la capacidad de escalabilidad de un sistema.

Un principio para diseñar sistemas de gran escala es que la demanda de servicio de cualquier componente del sistema debe estar limitada por una constante independiente del número de nodos del sistema. Cualquier mecanismo de servicio cuya demanda es proporcional al tamaño del sistema se saturara en cuanto el sistema sobrepase cierto tamaño. Este problema no se aliviará añadiendo mas recursos; la capacidad de este mecanismo limita el crecimiento del sistema.

No deben emplearse los recursos centrales ni los esquemas de control central para construir sistemas escalares. La alternativa ideal es una configuración funcionalmente simétrica; es decir, todas las máquinas componentes tienen un papel igual en el funcionamiento del sistema y por consiguiente cada máquina tiene cierto nivel de autonomía.

La aproximación práctica a la configuración simétrica y autónoma es el agrupamiento (clustering). El sistema se particiona en un conjunto de agrupamientos semiautónomos y cada uno consiste en un

conjunto de maquinas y un servidor dedicado al agrupamiento. Para que las referencias a recursos entre agrupamientos sean poco frecuentes, las solicitudes de cada maquina deben satisfacerse, la mayoría de las veces con su propio servidor de agrupamiento.

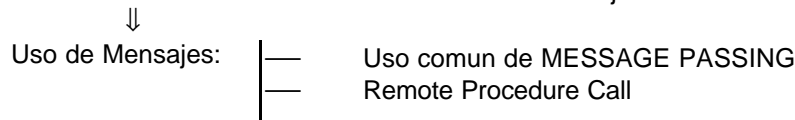
Si un agrupamiento esta bien equilibrado (si el servidor es suficiente para satisfacer todas las demandas del agrupamiento), se puede emplear como bloque de construcción modular para el crecimiento del sistema.

La estructura de procesos del servicio es un problema mayor en el diseño de cualquier servicio. Supuestamente, los servicios deben operar de manera eficiente en los periodos pico, cuando hay que dar servicio simultáneo a cientos de clientes activos. Obviamente, un servicio monoprocesador no es una buena opción, ya que cuando una solicitud requiera una E/S de disco todo el servicio se bloqueará. Es mejor opción asignar un proceso para cada cliente; pero hay que considerar el costo de los cambios de contexto.

Una de las mejores soluciones para la arquitectura del servidor es la utilización de procesos ligeros o hilos. La abstracción que presenta un grupo de procesos ligeros es que los diversos hilos de control están asociados a algunos recursos compartidos. Usualmente, un proceso ligero no esta limitado a un cliente, sino que atiende la solicitud de varios.

11.3.3. Comunicación de procesos distribuidos:

En Sistemas Distribuidos no se comparte Memoria Central generalmente. El uso de semáforos y áreas de memoria común NO funciona. Se usan mensajes.



Procesamiento Distribuido mediante Paso de Mensajes (MESSAGE PASSING)

Un proceso cliente solicita un servicio (por ejemplo, leer un archivo o imprimir) y envía un mensaje que contiene una petición de servicio a un proceso servidor. El proceso servidor cumple con la petición y envía una respuesta. En su forma más simple, sólo se necesitan dos funciones: Enviar (sent) y Recibir (receive). La función Enviar debe especificar un destino e incluir el contenido del mensaje. La función Recibir dice de quién se desea recibir mensaje y proporciona un almacenamiento intermedio (buffer) donde se guardará el mensaje.

En el método de paso de mensajes, los procesos hacen uso de los servicios de un módulo de paso de mensajes, que forma parte del S.O.. las solicitudes de servicio pueden expresarse en forma de primitivas y parámetros. La primitiva especifica la función a realizar y los parámetros se usan para datos e información de control.

Cuando se recibe la unidad de datos en el sistema de destino, se encaminará, mediante el servicio de comunicaciones, hacia el módulo de paso de mensajes. Éste módulo examina el campo identificador del proceso y almacenará el mensaje en el buffer de dicho proceso.

Confiables frente a No Confiables

Un servicio de paso de mensajes fiable es aquél que garantiza el envío si es posible. Dicho servicio debería hacer uso de un protocolo de transporte fiable o de alguna lógica similar y llevaría a cabo chequeos de errores, acuses de recibo, retransmisiones y reordenamiento de mensajes desordenados. Como el envío está garantizado, no es necesario hacer que el proceso emisor sepa que el mensaje fue enviado. Sin embargo, puede ser útil proporcionar un acuse de recibo al proceso emisor de manera que se entere de que tuvo lugar el envío.

El servicio de paso de mensajes puede enviar simplemente el mensaje a la red de comunicaciones sin informar de su éxito ni de su fracaso. Esta alternativa reduce enormemente la complejidad y la sobrecarga de proceso y de comunicaciones en el servicio de paso de mensajes.

Bloqueantes frente a No Bloqueantes

Con primitivas no bloqueantes, un proceso no es suspendido como resultado de hacer un Enviar o un Recibir. Cuando un proceso emita una primitiva enviar, el S.O. le devolverá el control tan pronto como el mensaje se haya puesto en cola para su transmisión o se haya hecho una copia. Cuando el mensaje se haya transmitido o se haya copiado a un lugar seguro para su posterior transmisión, el proceso emisor se verá interrumpido e informado de que el buffer del mensaje puede reciclarse. De forma similar, un Recibir no bloqueante lo emite un proceso para después seguir ejecutando. Cuando llegue un mensaje, el proceso es informado mediante interrupción o bien puede muestrear su estado periódicamente.

Las primitivas no bloqueantes ofrecen un empleo eficiente y flexible del servicio de paso de mensajes para los procesos. La desventaja de este enfoque es que los programas que emplean estas primitivas son difíciles de probar y depurar. Las secuencias irreproducibles dependientes del tiempo pueden originar problemas sutiles y complicados.

La otra alternativa es emplear primitivas bloqueantes. Un Enviar bloqueante no devuelve el control al proceso emisor hasta que el mensaje se haya transmitido (servicio no fiable) o hasta que el mensaje se haya enviado y obtenido un acuse de recibo (servicio fiable). Un Recibir bloqueante no devuelve el control hasta que el mensaje se haya ubicado en el buffer asignado.

Resumen sobre _Message Passing:

Es el más común en el modelo Cliente-Servidor.

- 1) Cliente Pide Servicio (por ejemplo: lectura de archivo).
- 2) Servidor manda la Respuesta del Pedido.

Confiabilidad de la Mensajería:

—	Asegurar entrega, dentro de lo posible.	Podría ser transparente al Proceso emisor.
—	Error Checking	
—	Acknowledge.	A lo sumo avisarle de imposibilidad.
—	Retransmisión	
—	Reordenamiento de algún desorden de mensajes.	

Llamadas a Procedimientos Remotos (RPC - Remote Procedure Call)

Es un método común muy aceptado actualmente para encapsular la comunicación en un sistema distribuido. Lo fundamental de la técnica es permitir que programas de máquinas diferentes interactúen mediante la simple semántica de los procedimientos de llamada/retorno, como si los dos programas estuvieran en la misma máquina. Es decir, se va a usar la llamada a procedimiento para acceder a servicios remotos. Ventajas:

1. La llamada a procedimientos es una abstracción muy usada, aceptada y bien comprendida
2. Permite que las interfases remotas se especifiquen como un conjunto de operaciones con nombre y tipo determinado. De este modo, la interfase puede documentarse de forma clara y los programas distribuidos pueden chequearse para detectar errores de tipo.
3. Como la interfase es estándar y está definida de forma precisa, el código de comunicaciones de una aplicación puede generarse automáticamente.
4. Los productores de software pueden escribir módulos clientes y servidores que pueden trasladarse entre computadores y S.O. con pocas modificaciones.

Puede considerarse como un refinamiento de mensajes confiable y bloqueante.

Paso de Parámetros

La mayoría de los lenguajes de programación permiten pasar parámetros por valor (llamada por valor) o como punteros a ubicaciones que contienen el valor (llamada por referencia). El paso de parámetros por valor es sencillo para las llamadas a procedimientos remotos. Los parámetros se copian simplemente en el mensaje y se envían al sistema remoto. Las llamadas por referencia son más difíciles de implementar. Hace falta un único puntero para cada objeto, válido en todo el sistema.

Representación de Parámetros

Otra cuestión es cómo representar los parámetros y los resultados en los mensajes. Si el programa llamador y el invocado están construido en los mismos lenguajes de programación, sobre el mismo tipo de máquinas y con el mismo S.O., los requisitos de representación no son un problema.

El mejor enfoque para este problema es ofrecer un formato estándar para los objetos comunes, como los enteros, números en coma flotante, caracteres y cadenas de caracteres. De esta forma, los parámetros propios de cualquier máquina pueden convertirse a la representación estándar.

Enlace Cliente-Servidor

El enlace especifica la forma en que se establecerá la relación entre un procedimiento remoto y el programa llamador. Un enlace se forma cuando dos aplicaciones han establecido una conexión lógica y se encuentran preparadas para intercambiar órdenes y datos.

Los enlaces no persistentes suponen que la conexión lógica se establece entre dos procesos en el momento de la llamada remota y que la conexión se pierde tan pronto como se devuelvan los valores. Como una conexión requiere el mantenimiento de información de estado en ambos extremos, se consumirán recursos. El estilo no persistente se utiliza para reservar dichos recursos.

Con enlaces persistentes, una conexión establecida para una llamada a un procedimiento remoto se mantiene después de terminar el procedimiento. La conexión puede utilizarse para llamadas futuras. Si transcurre un periodo de tiempo específico sin actividad en la conexión, se finaliza la misma. El enlace persistente mantiene la conexión lógica y permite que una secuencia de llamadas y retornos utilicen la misma conexión.

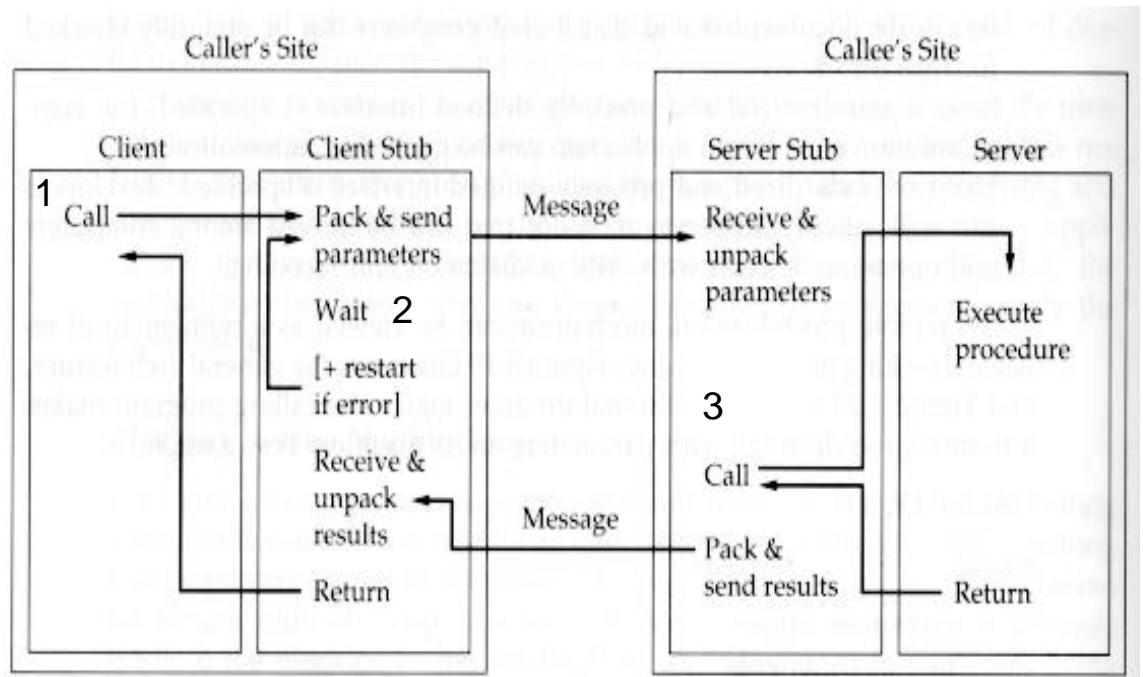


Fig. 11.11. Llamadas a procedimientos Remotos

Sincrónico frente a Asincrónico

La llamada tradicional a procedimiento remoto es sincrónica, lo que requiere que el proceso llamador espere hasta que el proceso llamado devuelva un valor.

Una aplicación típica de las RPC asincrónicas es hacer que un cliente invoque repetidamente a un servidor, generando una serie de peticiones de una vez, cada una con su propio conjunto de datos. La sincronización del cliente y el servidor puede conseguirse de dos maneras:

1. Una aplicación de nivel superior en el cliente y en el servidor puede iniciar el intercambio y comprobar al final que se han llevado a cabo todas las acciones solicitadas.
2. Un cliente puede emitir una cadena de RPC asincrónicas, seguidas de una RPC sincrónica final. El servidor responderá a la RPC sincrónica sólo después de culminar todo el trabajo solicitado en las RPC asincrónicas precedentes.

Resumen sobre Remote Procedure Call (RPC):

- Encapsulación de la comunicación en SD.
- Idea: Permitirle a Programas en diferentes máquinas:

Interactuar

Como?

↓
Usando llamadas a Procedimientos (semántica: Call / Return).

Como si los programas estuviesen en la misma máquina.

∴ Llama a procedimiento y accede a Servicios Remotos.

Lógica de RPC:

- 1) Call P(X,Y)
 - P = nombre del procedimiento
 - X = Argumento que se pasa. Y= Argumento que se devuelve.
 - Puede o no ser transparente al usuario que es Remoto.
- 2) Client Stub, Packea los parametros. DIR=PID+Procedimiento.
- 3) Server Stub, Examina lo que recibió y genera un Call P(X,Y) que es una llamada local.

11.3.4. Estados Globales en Sistemas Distribuidos

Todos los problemas de concurrencia que se plantea un sistema fuertemente acoplado, como la mutua exclusión, el Deadlock y la inanición, también aparecen en un sistema distribuido. Las estrategias de diseño en este campo se complican con el hecho de que no existe un estado global del sistema. Es decir, no es posible que el sistema operativo o algún proceso conozcan el estado actual de todos los procesos del sistema distribuido. Un proceso puede conocer solamente el estado actual de todos los procesos del sistema local, accediendo a los bloques de control de proceso en la memoria. De todos los procesos remotos, un proceso sólo puede conocer la información de estado que recibe mediante mensajes, pero estos representan el estado del proceso remoto en algún instante pasado. Para comprender la dificultad afrontada y poder formular la solución, se van a definir los siguientes términos:

- **Canal:** Existe un canal entre dos procesos siempre que intercambian mensajes. Se puede pensar en un canal como un camino o un medio por el que se transmiten los mensajes. Por conveniencia, los canales se contemplarán como si fueran de un solo sentido. De este modo, si dos procesos intercambian mensajes, se necesitarán dos canales, uno para cada dirección de la transferencia.
- **Estado:** El estado de un proceso es la secuencia de mensajes enviados y recibidos por los canales del proceso.
- **Instantánea:** Una instantánea registra el estado de un proceso. Cada instantánea incluye un registro de todos los mensajes enviados y recibidos por todos los canales desde la instantánea anterior.
- **Estado global:** El estado combinado de todos los procesos.
- **Instantánea distribuida:** Un conjunto de instantáneas, una para cada proceso.

El problema que se afronta es no poder determinar un estado global que sea cierto, debido al lapso de tiempo asociado con cada transferencia de un mensaje. Se puede intentar definir un estado global reuniendo instantáneas de todos los procesos.

Algoritmo de Instantáneas Distribuidas

Existe un algoritmo de instantáneas distribuidas en el que se registra un estado global consistente. El algoritmo supone que los mensajes se entregan en el mismo orden en que son enviados y que no se producen pérdidas. Un protocolo de transporte fiable (como TCP) cumpliría con estos requisitos. El algoritmo hace uso de un mensaje de control especial, llamado Token (ficha o marcador).

11.4. Procesos y procesadores en sistemas distribuidos

Al tratar el tema, haremos énfasis a los aspectos del manejo de procesos que usualmente no se estudian en el contexto de los sistemas operativos clásicos.

Veremos como se enfrenta la existencia de muchos procesos.

En muchos sistemas operativos distribuidos, es posible tener muchos hilos de control dentro de un proceso. Esta capacidad tiene algunas ventajas importantes, pero también introduce varios problemas.

Finalmente analizaremos la asignación de procesadores y planificación en los sistemas distribuidos.

11.4.1 Los Hilos (threads)

Existen situaciones en donde se desea tener varios hilos de control que comparten un único espacio de direcciones, pero se ejecutan de manera casi paralela, como si fuesen de hecho, procesados independientemente, excepto por el espacio de direcciones compartidas.

Hilos

En la figura 11.11a, vemos una máquina con tres procesos de manera que cada uno tiene un solo hilo de control. A los hilos (Threads), por lo general, se los llaman simplemente hilos o a veces procesos ligeros. Los hilos son como miniprosesos. Cada hilo se ejecuta en forma estrictamente secuencial y tiene su propio contador de programa y una pila para llevar un registro de su posición. Los hilos comparten la CPU de la misma forma que los hacen los procesos: primero se ejecuta un hilo, después otro y así

sucesivamente (tiempo compartido). Solo en un multiprocesador se puede ejecutar realmente en paralelos hilos como se ilustra en la Fig. 11.11b.

Los hilos pueden crear hilos hijos y se pueden bloquear en espera que se terminen llamadas al sistema al igual que los procesos regulares. Mientras un hilo esta bloqueado, se puede ejecutar otro hilo del mismo proceso, exactamente en la misma forma de que, cuando se bloquea un proceso, se puede ejecutar en la misma máquina otro proceso.

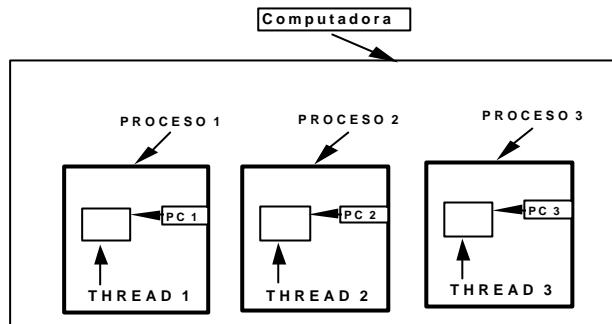


Fig. 11.11a Tres procesos con un Thread cada uno

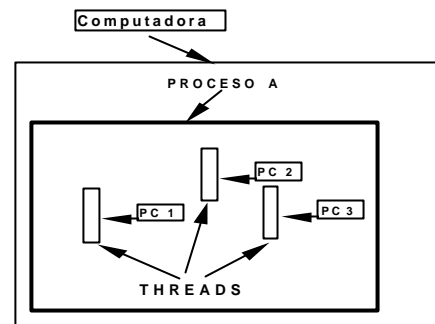


Fig. 11.11b un proceso con 3 Threads

Sin embargo los distintos hilos de un proceso no son tan independientes como los procesos distintos. **Todos los hilos tienen el mismo espacio de direcciones**, lo que quiere decir que comparten también las mismas variables globales. Puesto que cada hilo puede tener acceso a cada dirección virtual, un hilo puede leer, escribir o limpiar de manera completa la pila de otro hilo. No existe protección entre los hilos, debido a que:

Es imposible
No debe ser necesaria

A diferencia de los procesos, en procesos distintos, un proceso siempre es poseído por un único usuario, que puede haber creado varios hilos para que estos cooperen y no compitan entre sí. Además de compartir un espacio de direcciones, todos los hilos comparten el mismo conjunto de archivos abiertos, procesos hijos, Relojes o cronómetros, señales, etc.

Los hilos pueden tener uno de los siguientes estados:

- En ejecución
- Bloqueado
- Listo o terminado

TCB	PCB
Elementos por hilo	Elementos por procesos
Identificación	Identificación
Contador del programa	Espacio de dirección
Pila	Variables globales
Conjunto de registros	Archivos abiertos
Hilos de los hijos	Procesos hijos
Estado	Cronómetros
	Señales
	Semáforos
	Información Contable
	pointers
	Una entrada por cada hilo

Tabla 11.3. Contenidos de los bloques de control de hilos y de procesos

Un hilo en ejecución posee la CPU y está en estado activo. Un hilo bloqueado espera que otro elimine el bloqueo (por ejemplo en un semáforo)

Un hilo listo esta programado para su ejecución, la cual se llevará a cabo tan pronto le llegue su turno. Esto implica tener un planificador de hilos.

Por último un hilo terminado es aquel que ha hecho su salida pero que todavía no ha sido recibido por su padre.

Utilización de los hilos

Los hilos se inventaron para permitir la combinación del paralelismo con la ejecución secuencial y el bloqueo de las llamada al sistema.

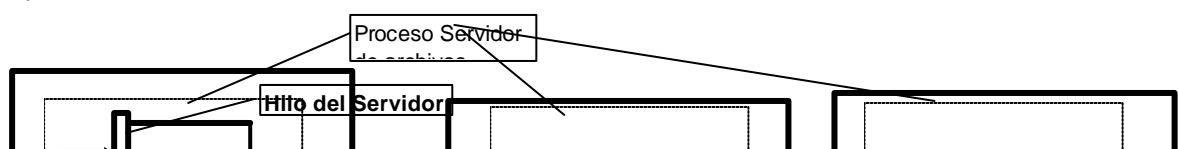


Fig 11.12. (a) Modelo de servidor/trabajador (b) Modelo de Equipo (c) Modelo de Pipeline

Una posible organización, es la que un hilo servidor lee las solicitudes de trabajo en el buzón del sistema como se observa en la figura 11.12a. Después de examinar la solicitud, elige a un hilo trabajador inactivo y le envía dicha solicitud. El hilo servidor despierta entonces al trabajador dormido (por ejemplo, lleva a cabo una operación **P()** o **UP()** o **Wait()** en el semáforo en donde duerme).

Cuando el trabajador despierta, verifica si se puede satisfacer la solicitud por medio del bloque caché compartido. Si no, envía un mensaje al disco para obtener el bloque necesario y se duerme en espera de la conclusión de la operación del disco. Se llama, entonces, al planificador y se inicializa otro hilo para pedir mas trabajo o bien a otro trabajador listo para realizar un trabajo.

Imaginémonos ahora como se podría describir el servidor de archivos en ausencia de hilos. Una posibilidad es que opere como un único hilo. El ciclo principal del servidor de archivos obtiene una solicitud, la examina y concluye antes de obtener la siguiente.

Mientras espera al disco, el servidor esta inactivo y no procesa otras solicitudes. Si el servidor de archivos se ejecuta en una maquina exclusiva para el, el procesador esta inactivo mientras el servidor espera al disco. el resultado neto es que se pueden procesar muchas menos solicitudes /segundo.

Ahora, supongamos que no se dispone de hilos, pero los diseñadores del sistema, consideran inaceptable la perdida de desempeño debida al uso de hilos únicos.

Entonces, una tercera posibilidad es ejecutar el servidor como una gran máquina de estado finito. Al recibir una solicitud, el único hilo la examina. Si lo logra, esta todo bien sino, envía una solicitud al disco. Pero en vez del Deadlock, se registra el estado de la solicitud actual en una tabla, va hacia ella, pero puede ser una solicitud de nuevo trabajo o una respuesta del disco con respecto a una operación anterior. En el caso de nuevo trabajo, a este se le da comienzo. Si es una información del disco, se busca información relevante en la tabla y se procesa la respuesta.

Hay que rescatar que hay que guardar el estado del computo y resguardarlo en la tabla, para cada mensaje enviado o recibido.

Debe quedar en claro , entonces, lo que los hilos ofrecen. Ellos mantienen la idea de procesos secuenciales que hacen llamadas al sistema con Deadlock.

La estructura del servidor en la figura 11.12a, es una manera de organizar un proceso de muchos hilos. Existen otras variantes. En el modelo de EQUIPO, todos los hilos son iguales y cada uno obtiene y procesa sus propias solicitudes. No hay servidor. Esto se ejemplifica en la Figura 11.12b.

Modelo	Característica
Hilos	Paralelismo, Deadlock de llamadas al sistema
Proceso de un solo hilo	Sin paralelismo, Deadlock de llamadas al sistema
Máquina de estado finito	Paralelismo, sin Deadlock de llamadas al sistema

Tabla 11.4 Tres formas para construir un servidor

A veces llega trabajo que un hilo no puede manejar. En este caso, se puede utilizar una cola de trabajo, la cual contiene todos los trabajos pendientes. Con este tipo de organización, un hilo debe verificar primero la cola de trabajo antes de buscar en el buzón del sistema.

Los hilos se pueden organizar, también, mediante el modelo de entubamiento o pipeline como se muestra en la figura 11.12c. Es decir, el primer hilo, genera ciertos datos y los transfiere al siguiente para su procesamiento. Los datos pasan de hilo en hilo y en cada etapa se lleva a cabo cierto procesamiento. Esta puede ser una buena opción en ciertos problemas, como el de los PRODUCTORES Y CONSUMIDORES pero no funciona como servidor de archivos.

Características del diseño de un paquete de hilos

Un conjunto de primitivas relacionadas con los hilos, disponibles para los usuarios, se denomina un **paquete de hilos**.

- a) **Manejo de los hilos:** aquí se tienen dos alternativas, los hilos dinámicos y los estáticos. En un diseño estático, se elige el número de hilos al escribir el programa o durante su compilación. Cada uno de ellos tiene asociada una pila fija. Este método es simple pero poco flexible. Es mas práctico consiste en permitir la creación y destrucción de los hilos durante la ejecución
- b) **Creación de hilos:** La llamada para la creación de hilos, determina el programa principal del hilo (como un puntero a un procedimiento) y un tamaño de pila , así como otros posibles parámetros. La llamada retorna un identificador de hilo para usarlo en las posteriores llamadas relacionadas con el hilo. En este modelo, un proceso se inicia de manera implícita con un único hilo, pero puede crear el número necesario de ellos.
- c) **Terminación de los hilos:** Los hilos pueden finalizar de dos maneras: puede hacer su salida por su cuenta al terminar su trabajo y destruirse o puede ser eliminado desde el exterior.
- d) **Datos compartidos:** Puesto que los hilos comparten una memoria común, pueden utilizar a ésta para guardar los datos que comparten los distintos hilos, por ejemplo buffers, direcciones de memoria de variables globales, etc.. El acceso a los datos compartidos se programa mediante regiones críticas con semáforos o monitores, para evitar que varios hilos intenten tener acceso a los mismos datos al mismo tiempo.
- e) **sincronización de hilos:** Otra característica de sincronización que a veces esta disponible en los paquetes de hilos, es la variable de condición o un semáforo mutex. Un hilo cierra un MUTEX para obtener la entrada a una región crítica. Una vez dentro de ella, examina las tablas de sistema y encuentra que ciertos recursos necesarios para el , esta ocupado. Si simplemente cierra un segundo MUTEX (asociado al recurso) el MUTEX exterior permanecerá cerrado y el hilo que conserva el recurso no podrá entrar a la región crítica para liberarlo. Ocurre un bloqueo. Si se elimina la cerradura del MUTEX exterior, otros hilos pueden entrar a la región crítica, lo cual provoca un caos.

Una solución es el uso de las **variables de condición** para adquirir el recurso. En este caso, la espera de la variable de condición, se define de manera atómica, de modo que ejecute en forma automática la espera y elimine la cerradura. Mas adelante, cuando el hilo que conservaba el recurso lo libera, llama a WAKEUP que se define de forma que despierta a un único hilo o bien a todos los hilos que esperan la variable de condición especificada.

Por ejemplo en el siguiente código se detalla la forma de resolver esta situación:

Entrada_RC()	Salida_RC()
lock mutex;	lock mutex;
check data structures;	mark resource as free;
while (resource busy)	unlock mutex;
wait (condition variable);	wakeup (condition variable);
mark resource as busy;	
unlock mutex;	

Tabla 11.5 candados mutex.

El código de un hilo consta por lo general de varios procedimientos al igual que un proceso. Estos pueden tener variables locales, variables globales y parámetros del procedimiento. Las variables locales y los parámetros no provocan ningún problema, pero las variables globales de un hilo, que no son globales en todo el programa, pueden provocar ciertas dificultades.

En otra alternativa se pueden introducir nuevos procedimientos de biblioteca para crear, dar valores y leer estas variables globales a lo largo de todo un hilo.

Implementación de un paquete de hilos

Existen dos vías para implantar un paquete de hilo: en el espacio del usuario y en el espacio del kernel. La Fig. 11.13 muestra estas formas de implantar.

En el primer caso, consiste en colocar todo el paquete de hilos en el espacio del usuario. El núcleo no sabe de su existencia. En lo que respecta a este, maneja procesos comunes con un único hilo. La primera ventaja, (la mas trivial) es que, un paquete de hilos implantados en el espacio del usuarios, se puede implantar en un sistema operativo que no soporta dichos hilos. Por ejemplo, UNIX no soporta hilos, pero para él existen distintos paquetes de hilos en el espacio del usuario.

Los hilos se ejecutan en la parte superior de un sistema en tiempo de ejecución (Run Time), el cual es una colección de procedimientos que manejan los hilos. Cuando un hilo ejecuta una llamada al

sistema, se duerme, desarrolla una operación en un semáforo o MUTEX, o bien lleva a cabo cualquier acción que pueda provocar su suspensión. Este procedimiento verifica si hay que suspender al hilo. En tal caso, almacena los registros del hilo (es decir su bloque de control TCB) en una tabla, busca un hilo no bloqueado para ejecutarlo, y vuelve a cargar los registros de la máquina con los valores resguardados del nuevo hilo.

Los hilos a nivel usuario también tienen otras ventajas. Permiten que cada proceso tenga su propio algoritmo de planificación. Para ciertas aplicaciones, aquellas que cuentan con un hilo recolector de basura (Garbage Collection), es muy bueno no tener que preocuparse por el hecho de que un hilo se detenga en un momento inconveniente.

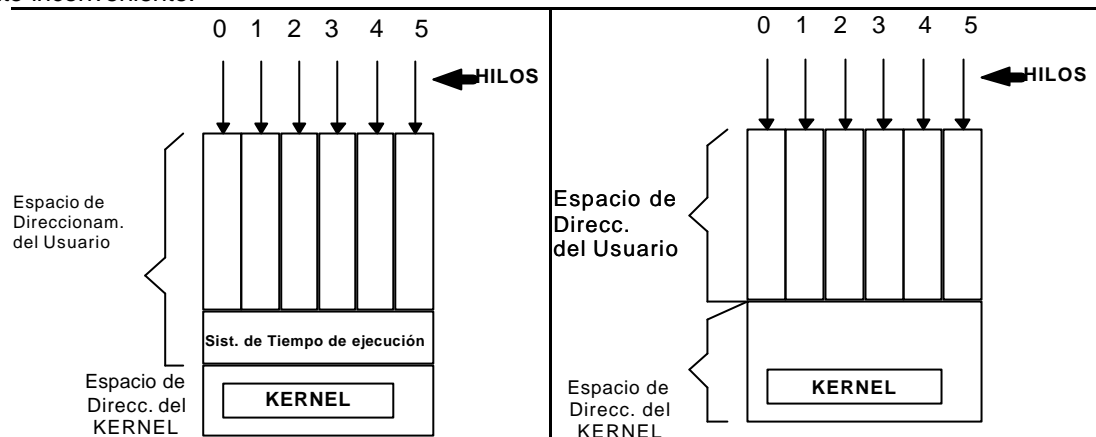


Fig. 11.13 (a) Un paquete de hilos a nivel usuario (b) Paquete de hilos manejado por el núcleo

Ahora, consideremos que el núcleo sabe de la existencia de los hilos y debe manejarlos. No se necesita un sistema de tiempo de ejecución. Para cada proceso, el núcleo tiene una tabla (Tabla 11.4 TCB) con una entrada para cada hilo, con los estados, registros y demás información referidas a cada hilo. La información es la misma que en el caso de los hilos a nivel usuario, solo que ahora se encuentra en el espacio del núcleo y no en el espacio del usuario.

Entonces, todas las llamadas que pueden bloquear un hilo se implementan como llamadas al sistema con un costo considerablemente mayor que una llamada a un procedimiento del sistema en tiempo de ejecución. Cuando un hilo se bloquea, el núcleo puede optar por ejecutar otro hilo del mismo proceso o un hilo de otro proceso. Con los hilos a nivel usuario, el sistema de tiempo de ejecución mantiene en ejecución los hilos del propio proceso hasta que el núcleo los retira del procesador.

Pero los paquetes de hilos, tienen algunos problemas fundamentales: uno de ellos, el problema de la implantación de las llamadas al sistema con bloqueos.

Supongamos que un hilo lee un pipeline vacío o que realiza alguna acción que provoca un bloqueo. En la implantación del núcleo, el hilo hace una llamada al núcleo, el cual bloquea el primer hilo e inicia otro. En la implantación del usuario, no se puede permitir que el hilo en realidad realiza una llamada al sistema, pues detendría todos los hilos. Uno de los principales objetivos a implementar con los hilos, es permitir a cada uno de ellos utilicen llamadas con bloqueo, pero evitar que un hilo bloqueado afecte a los demás.

Otro problema que surge con los paquetes de hilos a nivel usuario, es que si un hilo comienza su ejecución, ninguno de los demás hilos de ese proceso puede ejecutarse, a menos que ese hilo devuelva en forma voluntaria la CPU. Con los hilos a nivel núcleo se presenta otro problema: las interrupciones de reloj se generan en forma periódica, lo que obliga al planificador a ejecutarse. Dentro de un único proceso, no existen interrupciones de reloj, lo que imposibilita la planificación ROUND ROBIN. El planificador no tendrá oportunidad a menos que un hilo entre al sistema de tiempo de ejecución por voluntad propia.

El tercer problema en contra de los hilos a nivel usuario, es que por lo general, los programadores desean los hilos en aplicaciones donde estos se bloquean a menudo, como por ejemplo, un servidor de archivos con varios hilos.

Estos hilos, realizan constantes llamadas al sistema. Una vez que se hace una llamada al kernel para ejecutar un servicio del sistema, es probable que no haya mayor trabajo para el kernel que el de la conmutación de hilos, en el caso de que el primero de ellos este bloqueado. Si el núcleo lo hace, no hay necesidad de la constante verificación de la seguridad de las llamadas al sistema.

Hilos y RPC (Remote Procedure Call)

Normalmente se observa que en un sistema operativo distribuido, un gran número de llamadas a procedimientos remotos (RPC) se realiza con procesos donde una misma máquina es quien hace las llamadas.

Este resultado, depende del sistema. Una solución es un nuevo esquema que hace posible la llamada de un hilo en un proceso a un hilo de otro proceso en la misma máquina, de manera mucho mas eficiente que la usual.

Al iniciar un hilo servidor S, este exporta su interfase y le informa esta exportación al kernel. La interfase define los procedimientos que pueden llamar sus parámetros. Al iniciar un hilo cliente C, este importa la interfase del kernel y se le proporciona un identificador especial que podrá utilizarlo en la llamada. El kernel sabe ahora que C llamará posteriormente a S y crea estructuras de datos especiales con el fin de prepararse para la llamada. Una de estas estructuras de datos es una pila de argumentos compartida por C y S que se asocia de manera lectura-escritura a ambos espacios de direccionamientos. Para llamar al servidor S, C coloca sus argumentos en la pila compartida mediante el procedimiento normal de transferencia y luego hace una llamada al kernel y coloca su identificador especial en un registro para que sepa que es local. Si no tiene ese identificador el kernel sabría que es una llamada remota.

Después modifica el mapa de memoria del cliente para colocar este en el espacio de direcciones del servidor e inicia el hilo cliente, al ejecutar el procedimiento del servidor. La llamada se lleva a cabo de tal forma que los argumentos se encuentran ya en su lugar, de modo que no sea necesario el copiado o el ordenamiento. El resultado neto es que la RPC local se pueda realizar mas rápido de esta manera.

Aplicación: Ejemplo de un paquete de hilos

Examinaremos el ejemplo propuesto por DISTRIBUTED COMPUTER ENVIRONMENT (DCE). Este paquete, tiene un total de 51 primitivas, relacionadas con los hilos, las cuales pueden ser llamadas por los programas del usuario. Muchas de ellas no son estrictamente necesarias, pero se proporcionan solo por conveniencia. Este método es un tanto análogo a una calculadora de bolsillo de 4 funciones.

Para nuestro análisis, es conveniente agrupar las llamadas en categorías, cada una de las cuales se refiere a un aspecto distinto de los hilos y sus uso. La primera, trata del manejo de los hilos. Estas llamadas permiten la creación de hilos y su salida cuando termina su labor. Estas primitivas que se indican en la tabla 11.6, permite al usuario crear, destruir y manejar patrones para los hilos, MUTEX y variables de condición. El hecho de contar con patrones, elimina la necesidad de especificar todas las opciones como parámetros independientes.

Llamada	Descripción
Create	Crea un nuevo hilo
Exit	Un hilo la llama cuando éste termina
Join	Similar a la llamada al sistema WAIT en UNIX
Detach	Hace innecesaria la espera de un hilo padre cuando termina el proceso que hizo la llamada

Tabla 11.6 Llamadas al sistema

Otras llamadas, permiten a los programas leer y escribir los atributos de los patrones, tales como el tamaño de la pila y los parámetros de planificación. De manera análoga, se dispone de llamada para crear y eliminar patrones para MUTEX y variables de condición.

El tercer grupo se refiere a los MUTEX, que se pueden crear y destruir de manera dinámica. Se definen 3 operaciones en los MUTEX. Estas consisten en:

- Cerrar
- Eliminar la cerradura
- Intentar eliminar la cerradura

En el caso de los semáforos, si un proceso intenta incrementar un semáforo y otro intenta decrementarlo, no importa el orden preciso en que se lleve a cabo esto. Como resultado, no aparecen condiciones de competencia y la programación es relativamente directa. Existen dos tipos de MUTEX:

- Rápidos
- Amigables

Difieren en la forma con que trabajan las cerraduras anidadas. Un MUTEX RÁPIDO es como una cerradura en un sistema de base de datos. Si un proceso intenta cerrar un registro no cerrado, tendrá éxito. Sin embargo, si intenta adquirir la misma cerradura por segunda vez, se bloqueara en espera de la liberación de la cerradura, algo que tal vez nunca ocurra.

Llamada	Descripción
---------	-------------

Mutex_init	Crear un mutex
Mutex_destroy	Eliminar un mutex
Mutex_lock	Intento de cerrar un mutex; si ya está cerrado
Mutex_trylock	Intento de cerrar un mutex; falla si ya está cerrado
Mutex_unlock	Elimina la cerradura de un mutex

Tabla 11.7 Algunas de las llamadas a un mutex

Un MUTEX AMIGABLE, permite que un hilo cierre un MUTEX ya cerrado. A continuación vienen las llamadas relativas a las variables de condición, también se pueden crear y destruir de manera dinámica. Los hilos pueden dormir debido a variables de condición pendientes de la disponibilidad de cierto recurso necesario. Se tiene dos operaciones para despertarlos:

- Señalización: que despierta exactamente un solo hilo.
- Transmisión: que despierta a todos los hilos.

Llamada	Descripción
Cond_init	Crea una variable de condición
Cond_destroy	Elimina una variable de condición
Cond_wait	Espera de una variable de condición hasta que llega una señal o transmisión
Cond_signal	Despierta a lo más un hilo que espera a una variable de condición
Cond_broadcast	Despierta a todos los hilos que esperan a una variable de condición

Tabla 11.8 Llamadas de condiciones para threads

11.4.2 Asignación de procesadores a Procesos Distribuidos:

Un sistema distribuido consta de varios procesadores. Estos se pueden organizar como una colección de estaciones de trabajo personales, una pila pública de procesadores o alguna forma híbrida. Se necesita un cierto algoritmo para decidir cual proceso hay que ejecutar y en que máquina. Para el modelo de estaciones de trabajo, la pregunta es cuando ejecutar el proceso de manera local y cuando buscar una estación inactiva. Para el modelo de pila de procesadores, hay que tomar una decisión por cada nuevo proceso. Nos referiremos a este tema como ASIGNACIÓN DE PROCESADORES.

Modelos de asignación de procesadores

Partamos de la base que, casi todo el trabajo en esta área, supone que todas las máquinas son idénticas, son compatibles con el código y que difieren a lo sumo en la velocidad.

Partamos que todos los modelos presentados suponen que el sistema esta totalmente interconectado entre dos puntos, es decir, que cada procesador se puede comunicar con los demás, Esta hipótesis no quiere decir que cada máquina tenga un cable con cualquier otra máquina, sino que se puede establecer conexiones de transporte entre cualquier par de máquinas.

Se genera una nueva tarea cuando un proceso en ejecución desea crear un hijo o subproceso.

Ciertos casos, el proceso creador es el interprete de comandos (SHELL), que inicia un nuevo trabajo en respuesta a un comando del usuario. En otros, el propio proceso usuario crea uno o mas hijos. Las estrategias de asignación de procesadores se pueden dividir en dos categorías amplias:

- **No Migratoria:** al crearse un proceso, se toma una decisión acerca de donde colocarlo. Una vez colocado en una máquina, el proceso permanece ahí hasta que termina. No se puede mover, no importa lo sobrecargada que este la máquina ni que exista muchas otra máquinas inactivas.
- **Migratoria:** un proceso se puede trasladar aunque haya iniciado su ejecución. Mientras que las estrategias migratorias permiten un mejor balance de la carga, son substancialmente mas complejas y tienen un efecto fundamental en el diseño del sistema. Un algoritmo que asigne procesos a los procesadores, lleva implícito el intento por optimizar algo.

Objetivos

Un posible objetivo, podría ser maximizar el uso de los procesadores existentes en el sistema. Es decir, maximizar el número de ciclos de la CPU que se ejecutan en beneficio de los trabajos del usuario por cada hora de tiempo real (es una forma de decir que hay que evitar a todo costo el tiempo inactivo de la CPU).

Otro, es minimizar el tiempo promedio de respuesta. Una variación de la minimización del tiempo de respuesta es la minimización de la tasa de respuesta, la cual se define como:

$$Tr = \frac{T \text{ para ejecutar un proceso en la máquina } m}{T \text{ en ejecutarse un proceso en un procesador de referencia (sin carga)}}$$

Para muchos usuarios, es una métrica mas útil que el tiempo de respuesta, puesto que toma en cuenta el hecho de que los trabajos mas largos tardan mas que los pequeños.

Aspecto del diseño de algoritmos de asignación de procesadores

Las principales decisiones que deben tomar los diseñadores, se pueden resumir en 5 aspectos:

- Algoritmos deterministas VS heurísticos
- Algoritmos centralizados VS distribuidos
- Algoritmos óptimos VS subóptimos
- Algoritmos locales VS GLOBALES
- Algoritmos iniciados por el emisor VS iniciados por el receptor

- **Algoritmos Determinísticos y Heurísticos:** Son adecuados cuando se sabe de antemano todo acerca del comportamiento de los procesos. Imaginemos que tenemos una lista completa de todos los procesos, sus necesidades de cómputos, archivos, comunicación, etc. Con esta información es posible hacer una asignación perfecta. En teoría esto es así, pero en pocos sistemas, se tiene un conocimiento total de antemano, en general se puede obtener una aproximación razonable. Las solicitudes de trabajo dependen de QUIEN ESTA HACIENDO QUE y puede variar de manera drástica cada hora o cada minuto. La asignación de procesadores en tales sistemas, no se puede hacer de manera determinista o matemáticamente, sino por necesidad y se utilizan técnicas AD HOC, llamadas HEURÍSTICAS.
- **Diseño Centralizado vs Diseño Distribuido:** La recolección de toda la información en un lugar, permite tomar una mejor decisión, pero es menos robusta y coloca una carga pesada en la maquina central. Son preferibles los algoritmos descentralizados, pero se han propuesto algunos algoritmos centralizados por la carencia de alternativas descentralizadas adecuadas.
- **Algoritmos Óptimos vs Algoritmos Subóptimos:** Se pueden obtener las soluciones óptimas, tanto en los sistemas centralizados como en los descentralizados, pero por regla son mas caros que los subóptimos. Hay que recolectar mas información y procesarla un poco mas. En la práctica, la mayoría de los sistemas distribuidos reales buscan soluciones subóptimas heurísticas y distribuidas.
- **Algoritmos Locales vs Algoritmos Globales:** Se relaciona con lo que se llama a menudo POLÍTICA DE TRANSFERENCIA. Cuando se está a punto de crear un proceso, hay que tomar una decisión para ver si se ejecuta o no en la máquina que lo genera. Si esa máquina esta muy ocupada, hay que transferir a otro lugar el nuevo proceso. La opción en ese aspecto, consiste en basar o no a decisión de transferencia completamente en la información local.

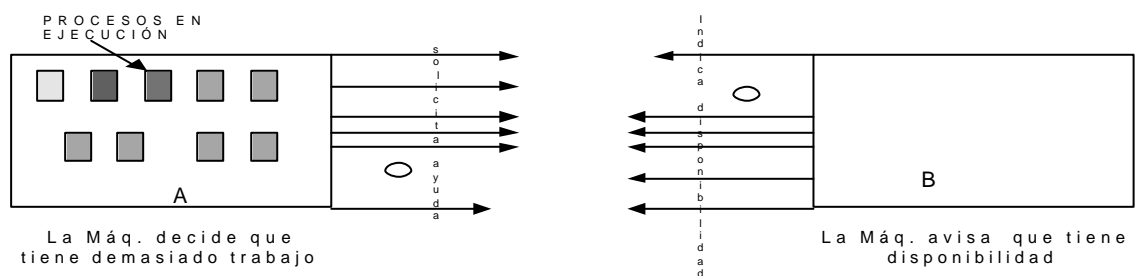


Fig 11.14 (a) Un emisor en búsqueda de una máquina inactiva (b) Un receptor en búsqueda de trabajo por realizar

- **Ejemplo sencillo de algoritmo (local):** Si la carga de la máquina esta por debajo de cierta máquina, se conserva el nuevo proceso. En caso contrario, se deshace de él. Otra escuela, dice que es mejor recolectar información (global) acerca de la carga, antes de decidir si la máquina local esta o no muy ocupada para otro proceso.
- **Algoritmos Iniciados por el emisor vs iniciados por el receptor:** Trata de la política de localización. Una vez que la política de transferencia ha decidido deshacerse de un proceso, la política de localización debe decidir donde enviarlo. Es claro que esta política no puede ser local. Necesita información de la carga en todas partes para poder tomar un decisión inteligente. Esta información se puede dispersar de dos formas:
 - En uno de los métodos, los emisores inician el intercambio de información.

—En el otro, es el receptor el que toma la iniciativa.

La máquina sobrecargada envía una solicitud de ayuda a las demás máquinas, con la esperanza de que descarguen el nuevo proceso en alguna otra máquina.

Aspectos de la implantación de algoritmos de asignación de procesadores

Para comenzar, casi todos los algoritmos suponen que las máquinas conocen su propia carga, de modo que pueden decir si están subcargadas o sobrecargadas y pueden informar a las demás máquinas de su estado. La medición de la carga no es tan sencilla como parece. Un método consiste en contar el número de procesos en cada máquina y utilizar ese número como la carga, pero en un sistema inactivo, pueden ejecutarse muchos procesos como *demonios de correo, noticias, administradores de ventanas...* Así, el contador del proceso, casi no dice nada de la carga actual.

El siguiente método consiste en contar solo los procesos en ejecución o listos. Cada proceso en ejecución o que se pueda ejecutar, impone cierta carga a la máquina, aunque sea un proceso secundario.

Una medida mas directa, es la fracción de tiempo que la CPU esta ocupado. Una máquina con 20 % de uso de la CPU, tiene una carga mayor que la de una máquina con 10 % de uso de CPU, sin importar si ejecuta programas del usuario o demonios o procesos remotos.

Una forma de medir el uso de la CPU es configurar un reloj y dejarlo que interrumpa la máquina en forma periódica. En cada interrupción se observa el estado de la CPU. De esta forma se puede observar la fracción de tiempo que se gasta en el ciclo inactivo.

Otro aspecto de la implantación, es el enfrentamiento con el costo excesivo. Muchos de los algoritmos teóricos para la asignación de procesos, ignoran el costo de recolectar medidas y desplazar los procesos de aquí para allá. Un algoritmo adecuado, tomaría en cuenta el tiempo de CPU, uso de memoria y el ancho de banda de la red utilizada por el propio algoritmo para asignación de procesadores.

Nuestra siguiente consideración en torno a la implantación, será la complejidad. Pocas veces se considera también la complejidad del SOFTWARE en cuestión.

Un estudio realizado por EAGER en 1986, arroja cierta luz en el tema de la persecución de algoritmos complejos y óptimos. Se estudiaron 3 algoritmos.

En todos los casos, cada máquina del sistema mide su propia carga y decide por sí misma si esta subcargada. Al crearse un nuevo proceso, la máquina que lo crea verifica si está sobrecargada. En tal caso, verifica una máquina remota para poder iniciar el nuevo proceso. Los tres algoritmos difieren en la localización de la máquina candidato:

- Elige una máquina de manera aleatoria y tan solo envía allí el nuevo proceso. Si la máquina receptora esta sobrecargada, elige nuevamente una máquina al azar y le envía el proceso. Esto se repite hasta que alguien este dispuesto a tomar el proceso o que se exceda un contador de tiempo, en cuyo caso ya no se permite que avance.
- Elige una máquina en forma aleatoria y envía una prueba para ver si está subcargada o sobrecargada. Si la máquina admite estar subcargada, obtiene el nuevo proceso, en caso contrario, se repite la prueba. Este ciclo se repite hasta que se encuentre una máquina adecuada o se excede el número de pruebas, en cuyo caso, permanece en el sitio de su creación.
- Analiza k máquinas para determinar sus cargas exactas. El proceso se envía entonces a la máquina con la carga mas pequeña.

De forma intuitiva, concluimos en que si el uso de un algoritmo sencillo proporciona casi la misma ganancia que uno mas caro y mas complejo, es mejor utilizar el mas sencillo.

Un algoritmo determinista según la teoría de grafos

Algunos sistemas, constan de procesos con requerimientos conocidos de CPU y memoria, además de una matriz conocida con el trafico promedio entre cada pareja de procesos. Si el número de CPU k es menor que el número de procesos, habrá que asignar varios procesos a la misma CPU.

La idea es llevar a cabo esta asignación de forma que se minimice el trafico en la red. El sistema se puede representar como un grafo con pesos, donde cada nodo es un proceso y cada arco representa el flujo de mensajes entre dos procesos.

El problema se reduce entonces a encontrar una forma de parti el grafo en dos subgrafos ajenos, sujetas a ciertas restricciones. Por ejemplo, el total de requerimientos de CPU y memoria, deberá estar por debajo de ciertos limites para cada subgrafo. Para cada solución que cumpla las restricciones, los arcos contenidos totalmente dentro de un único subgrafo, representan la comunicación dentro de la máquina y se pueden ignorar.

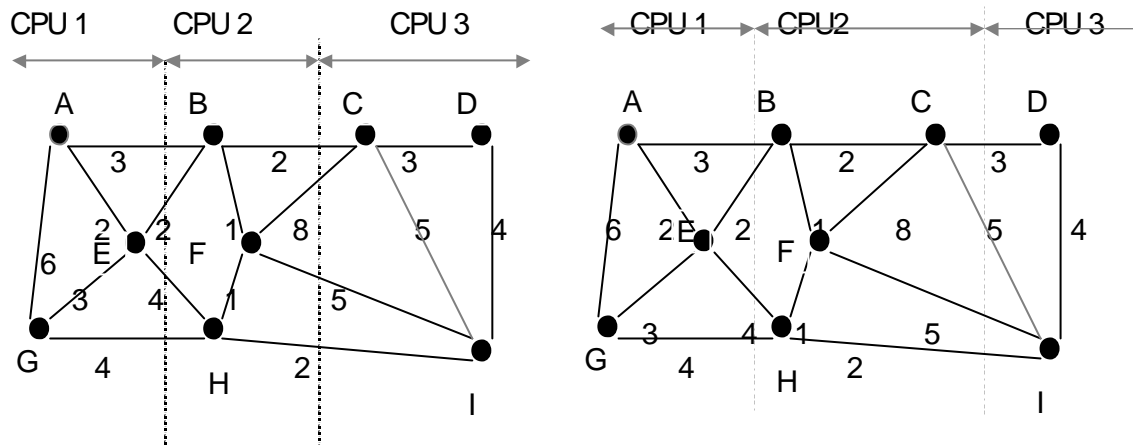


fig. 11.15 (a) y (b) Dos formas de asignar 9 procesos a 3 procesadores

Los arcos que van de un subgrafo a otro, representan el tráfico en la red. El objetivo es entonces encontrar la partición que minimice el tráfico en la red, a la vez que satisfaga todas las restricciones. En la Fig. 11.12 se ejemplifica esto.

Un algoritmo centralizado

Regresaremos a un algoritmo heurístico, que no necesita la información completa sobre los procesos que se deben ejecutar en el sistema.

El algoritmo se conoce con el nombre de ARRIBA-ABAJO y es centralizado en el sentido de que un coordinador mantiene una tabla de uso con una entrada por cada estación de trabajo personal (es decir, por usuario) con un valor inicial 0.

Cuando ocurren eventos significativos, se pueden enviar mensajes al coordinador para actualizar la tabla. Las decisiones de asignación se basan en ésta tabla. Se realiza una solicitud, se libera el procesador o bien el reloj hace una marca de tiempo.

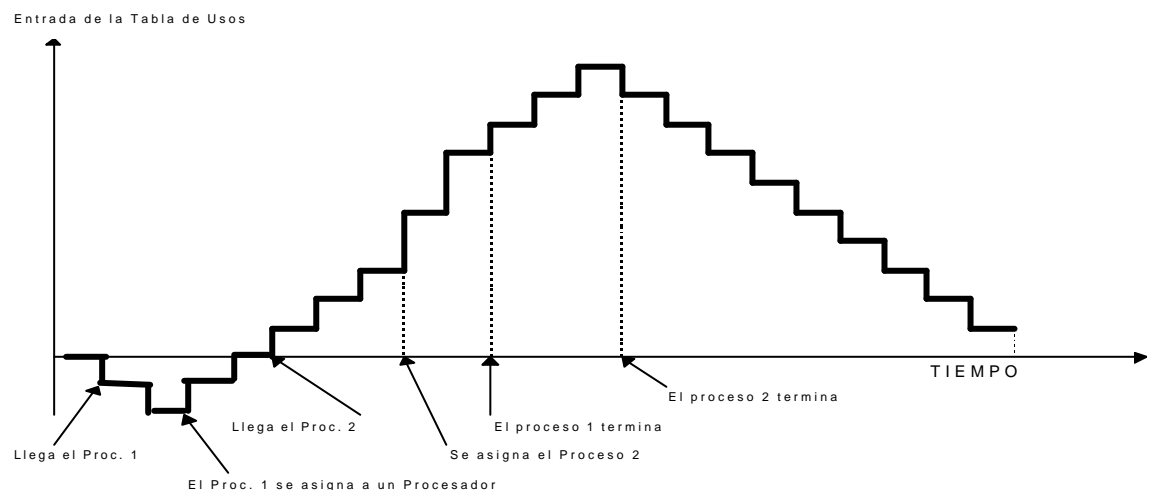


Fig. 11.16 Operación del algoritmo arriba-abajo

La razón de ser centralizado es que en vez de intentar maximizar el uso de CPU, se preocupa por darle a cada poseedor de una estación de trabajo una parte justa del poder del cómputo. Otros algoritmos otorgan todas las máquinas a un único usuario, con la única condición de que las mantenga ocupadas.

Cuando se va a crear un proceso y la máquina donde se crea decide que el proceso se debe ejecutar en otra parte, le pide al coordinador de la tabla de usos que le asigne un procesador. Si existe uno disponible, y nadie mas lo desea, se otorga el permiso. Si no existen procesadores libres, la solicitud se niega por el momento y se toma nota de ella.

Cuando el poseedor de una estación de trabajo ejecuta procesos en las máquinas de otras personas, acumula puntos de penalización. Estos se añaden a su entrada en la tabla de usos. Cuando tiene solicitudes pendientes no satisfechas, los puntos de penalización se restan de su entrada en la tabla.

de usos. Si no existen solicitudes pendientes, y ningún procesador esta en uso, la entrada de la tabla de usos se desplaza un cierto número de puntos hacia el 0 hasta que llega ahí. De esta forma, su puntuación se mueve hacia arriba o hacia abajo. De ahí el nombre del algoritmo.

Una puntuación + (positiva) indica que la estación de trabajo es un usuario de los recursos del sistema, una - (negativa) significa que necesita recursos. La puntuación 0 es neutra.

Entonces, resumiendo, cuando un procesador se libera, gana la solicitud pendiente cuyo poseedor tiene la puntuación mas baja. En consecuencia, un usuario que no ocupe procesadores y que tenga pendiente una solicitud durante mucho tiempo, siempre vencerá a alguien que utilice muchos procesadores. Esta propiedad es la intención del algoritmo: ASIGNAR LA CAPACIDAD DE MANERA JUSTA.

Un algoritmo jerárquico

Otro método propuesto para etiquetar una colección de procesadores, es organizarlo mediante una jerarquía lógica, independiente de la estructura física de la red. Este método, organiza las máquinas como las personas en jerarquías corporativas, académicas. Algunas de las máquinas son trabajadores y otras administradores. Por cada grupo de k trabajadores hay una administradora cuya principal tarea es mantener un registro de las máquinas ocupadas y las inactivas. Si el sistema es grande, existirá un gran numero de jefes de departamento, por lo que algunas máquinas funcionarán como gerentes, por encima de cierto número de jefes.

Si existen muchos gerentes, también se pueden organizar de forma jerárquica y un jefe (gerente general) puede controlar a una colección de gerentes. Puesto que cada procesador solo necesita comunicarse con un superior y unos cuantos subordinados, el flujo de información es controlable.

Pero que ocurre si un jefe de departamento o un gran jefe detiene su funcionamiento?

La elección se puede llevar a cabo por los propios subordinados, por los compañeros del descompuesto o, en un sistema mas autocrático, por el jefe del administrador enfermo.

Para evitar un único administrador, en la parte superior del árbol, se puede tener un comité como ultima autoridad. Cuando un miembro del comité empieza a fallar, los demás miembros promueven a alguien del nivel inmediato inferior para el reemplazo.

La estrategia que se utiliza es que cada administrador mantenga un registro aproximado del número de trabajadores a su cargo que estén disponibles. Si el administrador que recibe la solicitud piensa que tiene muy pocos procesadores disponibles, transfiere la solicitud arriba, a su jefe. Si el jefe tampoco la puede manejar, la solicitud se sigue propagando hacia arriba donde tiene un número suficiente de trabajadores a su disposición. En ese momento, el administrador divide la solicitud en partes y las esparce entre los administradores por debajo de él, los cuales a su vez, repiten la operación hasta que la ola de asignación llega al punto inferior. En este nivel, los procesadores se señalan como ocupados y el número de procesadores asignados se informa de regreso hacia arriba del árbol.

Para que esta estrategia funcione bien, R (número de trabajadores) debe ser lo bastante grande como para que sea alta la probabilidad de encontrar cantidad suficiente de trabajadores para manejar todo el trabajo.

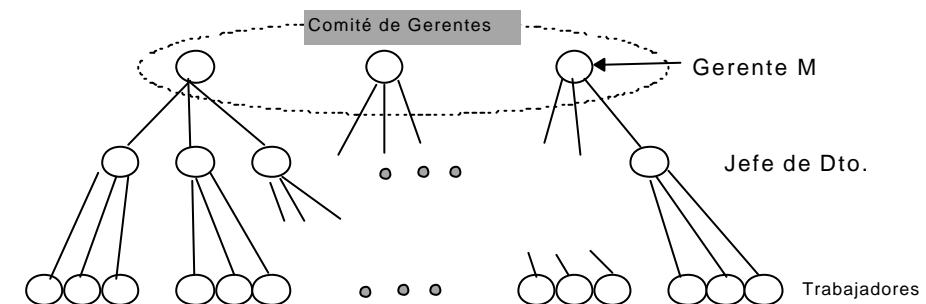


Fig. 11.17 Una jerarquía de procesadores se puede modelar como una jerarquía organizacional

11.5. Sincronización en sistemas distribuidos

En los sistemas que solo cuentan con una CPU, los problemas relativos a las regiones críticas, mutua exclusión y a la sincronización se resuelven en general mediante métodos tales como semáforos y monitores. Estos métodos no son adecuados para su uso en sistemas distribuidos, puesto que siempre se basan (de manera implícita) en la existencia de la memoria compartida. Por ejemplo dos procesos que interactúan mediante un semáforo deben poder tener acceso a este.

Si se ejecutan en la misma máquina pueden compartir el semáforo al almacenarlo en el núcleo y realizar llamadas al sistema para tener acceso a él.

Sin embargo si se ejecutan en máquinas distintas este método no funciona, por lo que se necesitan otras técnicas. Incluso las cuestiones mas sencillas como el hecho de determinar si el evento A ocurrió antes del evento E requieren una reflexión cuidadosa en los sincronizados puesto que los primeros deben utilizar algoritmos distribuidos

La sincronización es mas compleja en los sistemas distribuidos que en los centralizados puesto que los primeros deben utilizar algoritmos distribuidos

Por lo general no es posible reunir toda la información relativa al sistema en un solo lugar o nodo y después dejar que cierto proceso la examine y tome una decisión como se hace en el caso centralizado.

En general los algoritmos distribuidos tienen las siguientes propiedades

—La información relevante se distribuye en varias maquinas.

—Los procesos toman las decisiones solo con base en la información disponible en forma local

—Debe evitarse un único punto de falla en el sistema

—No existe un reloj común o alguna otra fuente precisa del tiempo global

Los tres primeros puntos indican que es inaceptable reunir toda la información en un solo lugar para su procesamiento.

El último punto de la lista es crucial. En un sistema centralizado, el tiempo no tiene ambigüedades, cuando un proceso desea conocer la hora llama al sistema y el kernel se lo dice. Si el proceso A pide la hora y un poco después, el proceso B también la pide el valor obtenido por B es mayor o igual al valor obtenido por A. Por supuesto no debe ser menor. En un sistema distribuido no es trivial poner de acuerdo a todas las máquinas en la hora. Esto implica que no existe un acuerdo global en el tiempo.

11.5.1. Sincronización de relojes

Puesto que el tiempo es fundamental para el punto de vista de la gente y el efecto de carácter de sincronización en los relojes puede ser muy problemático.

Características básicas para procesos que se intercambian datos:

El sincronismo es un atributo de los dos procesos y la comunicación.

Decimos que un sistema es SINCRONIZADO si cumple las siguientes propiedades:

- Tiene un límite máximo conocido en el retardo del mensaje, este consiste en el tiempo que toma enviar, transportar y recibir un mensaje a través de un vínculo de la red.
- Cada proceso p tiene un reloj local C_p un valor conocido p^o con respecto al tiempo real.
- Estos para todo p y para todo $t > t'$

$$(1+p)^{-1} \leq \frac{C_p(t) - C_p(t')}{(t - t')} \leq (1+p)$$

Tiene un límite máximo conocido para el tiempo que requiere un proceso en ejecutar un paso de trabajo (Job STEP).

En sistemas sincronizados es posible medir un time out de los mensajes, lo que provee un mecanismo para detectar fallas.

Además es posible implementar relojes aproximadamente sincronizados que además de las condiciones anteriores cumple lo siguiente:

- Hay un E tal que para todo t , y cualesquiera dos procesos p y q ,
- $|C_p(t) - C_q(t)| \leq E$, En tal caso se pueden implementar estos relojes aun en presencia de fallas
- Los relojes aproximados Sincrónicos tienen muchas aplicaciones como control de procesos en tiempo real, manejo de archivos, cache consistency, autenticación etc.

Para muchos problemas este tipo de reloj aproximados pueden ser usados como un verdadero reloj sincronizado, esto es con $E=0$ lo que simplifica el diseño de algoritmos distribuidos.

Un sistema asincrónico no tiene límite máximo conocido en el retardo del mensaje ni un límite máximo para que un proceso ejecute un STEP. Los sistemas asincrónicos no llevan un control del tiempo, sino que tienen tiempos específicos preestablecidos, lo que los hace atractivos para muchas aplicaciones.

Los modelos sincrónico y asincrónico son los dos extremos del espectro de modelos posibles. Han sido estudiados muchos modelos entre estos dos, como por ejemplo los procesos, pueden tener sus velocidades comprometidas y relojes perfectamente sincronizados pero los retardos de los mensajes estar no relacionados con ese reloj.

11.5.2. Relojes Lógicos

Puesto que todos los procesos de la máquina utilizan el mismo reloj, tendrán consistencia interna.

Tan pronto se comienza a trabajar con varias máquinas, cada una con su propio reloj, la situación es distinta. Aunque la frecuencia de un oscilador de cristal es muy estable, es importante garantizar que los cristales de computadoras distintas oscilen precisamente con la misma frecuencia. En la práctica, cuando un sistema tiene n computadoras, los n cristales correspondientes oscilarán a tasas un poco distintas, lo que provoca una pérdida de sincronización en los relojes (de software) y que al leerlos tengan valores distintos. La diferencia entre los valores del tiempo se llama **distorsión del reloj**. Como consecuencia de esta distorsión, podrían fallar los programas que esperan el tiempo correcto asociado a un archivo, objeto, proceso o mensaje. Esto es independiente del sitio en donde haya sido generado (es decir, el reloj utilizado).

Lamport (1978) demostró que la sincronización de los relojes, para obtener un estándar de tiempo único, sin ambigüedades, es posible y presentó un algoritmo para lograr esto.

La solución de Lamport se sigue en forma directa de la relación “ocurre antes de”.

Puesto que C sale en 60, debe llegar en 61 o en un tiempo posterior. Por lo tanto, cada mensaje acarrea el tiempo de envío, de acuerdo con el reloj de emisor. Cuando un mensaje llega y el reloj del receptor muestra un valor anterior al tiempo en que se envió el mensaje, rápidamente el receptor adelanta su reloj para que tenga una unidad más que el tiempo de envío.

Con una pequeña adición, este algoritmo cumple con las necesidades para el tiempo global. La adición es que entre cualesquiera dos eventos, el reloj debe marcar al menos una vez. Si un proceso envía o recibe dos mensajes en serie muy rápidamente, debe avanzar el reloj en una marca entre ellos.

Este algoritmo nos da una forma de obtener un orden total de todos los eventos en el sistema. Muchos otros sistemas distribuidos necesitan tal orden para evitar ambigüedades.

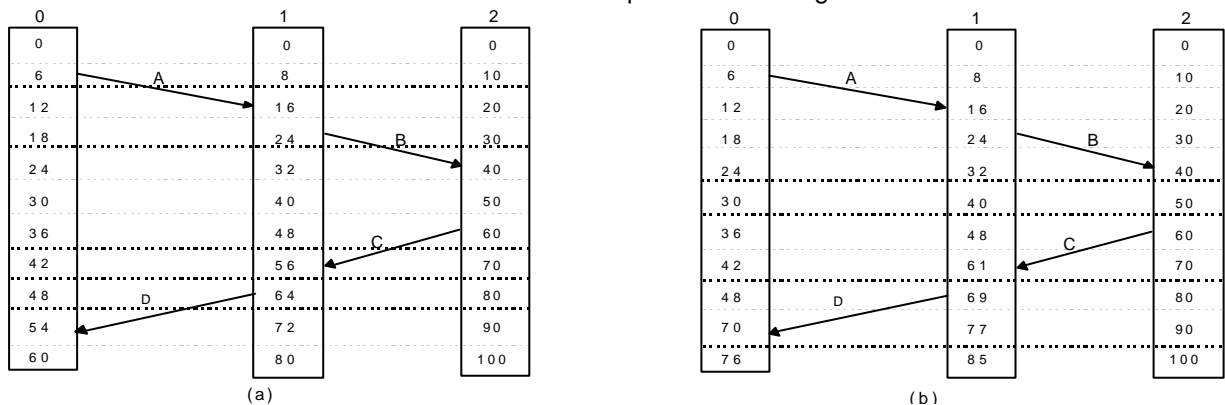


Fig. 11.18 (a) Tres procesos, cada uno con su propio reloj. Los relojes corren a diferentes velocidades (b) El algoritmo de Lamport corrige los errores

11.5.3. Relojes físicos

Algoritmos para la sincronización de relojes

Si una máquina tiene un receptor WWV WWV es una estación de radio de onda corta que emite una señal en cada inicio de segundo), entonces el objetivo es hacer que todas las máquinas se sincronicen con ella. Si una máquina tiene receptores WWV, entonces cada máquina lleva el registro de su propio tiempo y el objetivo es mantener el tiempo de todas las máquinas tan cercano como sea posible.

Se supone que cada máquina tiene un reloj, el cual provoca una interrupción H veces por cada segundo. Cuando este reloj se detiene, el manejador de interrupciones añade 1 a un reloj en software, el cual mantiene el registro de número de marcas (interrupciones) a partir de cierta fecha acordada en el pasado. Llamaremos al valor de este reloj C . Más precisamente, cuando el tiempo universal (universal time coordinated UTC) es t , el valor del reloj en la máquina p es $C * p(t)$. En un mundo perfecto, tendríamos $C * p(t) = t$ para toda p y toda t . En otras palabras, dC/dt debería ser 1.

En la realidad los cronómetros no producen interrupciones exactamente H veces por segundo. En teoría un cronometro con $H = 60$ generaría 216.000 marcas por hora. En la práctica se produce un error de 10^{-5} .

Si dos relojes se alejan de UTC en la dirección opuesta, en un instante Dt después de que fueron sincronizados, podrían estar tan alejados como $2pDt$. Si los diseñadores del sistema operativo desean garantizar que dos relojes cualesquiera no difieran más de d , entonces los relojes deben volverse a sincronizar al menos cada $d/2p$ segundos. Los distintos algoritmos difieren en la forma precisa en que realiza esta resincronización.

Algoritmo de Cristian

En los sistemas en los que una máquina tiene un receptor WWV, el objetivo es hacer que todas las máquinas se sincronicen en ella. La máquina con el receptor WWV se llama un despachador del tiempo.

En forma periódica, en un tiempo que no debe ser mayor que $d/2p$ segundos, cada máquina envía un mensaje al servidor para solicitar el tiempo actual, C_{UTC} , como se muestra en la figura 11.16.

Como primera aproximación, cuando el emisor obtiene la respuesta, puede hacer que su tiempo sea C_{UTC} . Sin embargo, este algoritmo tiene dos problemas. El problema mayor es que el tiempo nunca debe correr hacia atrás. Si el reloj del emisor es rápido, C_{UTC} será menor que el valor actual de C del emisor. El simple traslado de C_{UTC} podría provocar serios problemas.

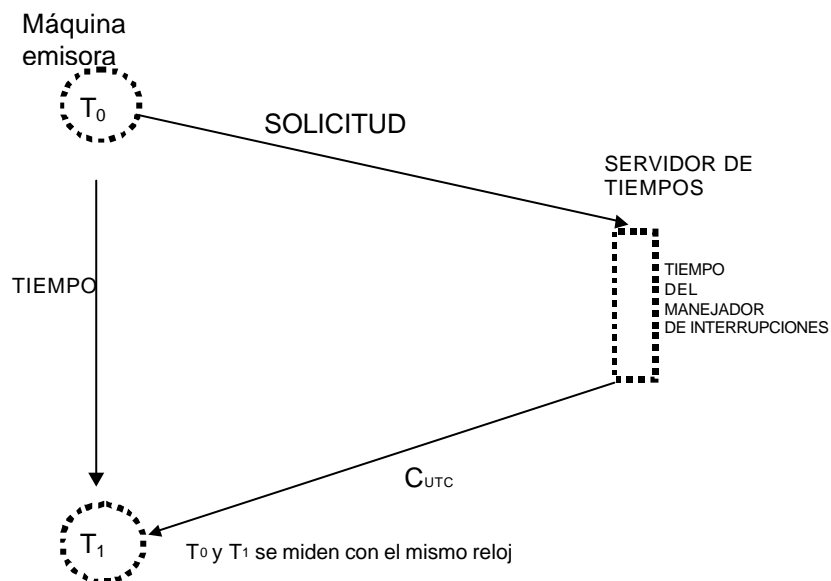


Fig. 11.19 Obtención de la hora actual por medio de un servidor de tiempos

El problema menor es que el tiempo que tarda el servidor del tiempo en responder al emisor es un tiempo distinto de cero. Aun peor, este retraso puede ser de gran tamaño y variar con la carga de la red. La forma de enfrentar este problema por parte de este algoritmo es intentar medirlo. Es muy fácil para el emisor registrar de manera precisa el intervalo entre el envío de la solicitud del servidor de tiempo y la llegada de la respuesta. Tanto el tiempo de inicio, T_0 , como el tiempo final, T_1 , se mide con el mismo reloj, por lo que el intervalo será relativamente preciso, aunque el reloj del emisor esté alejado de UTC por una magnitud sustancial.

El algoritmo de Berkeley

El servidor de tiempo está activo y realiza un muestreo periódico de todas las máquinas para preguntarles el tiempo. Con base en las respuestas, calcula un tiempo promedio y le indica a todas las demás máquinas que avancen su reloj a la nueva hora o que disminuyan la velocidad del mismo hasta lograr cierta reducción específica.

El método es el siguiente: El servidor le indica a las demás máquinas su hora y les pide las suyas, las máquinas responden con la diferencia, con estos datos el servidor calcula el tiempo promedio y le indica a cada máquina como ajustar su reloj.

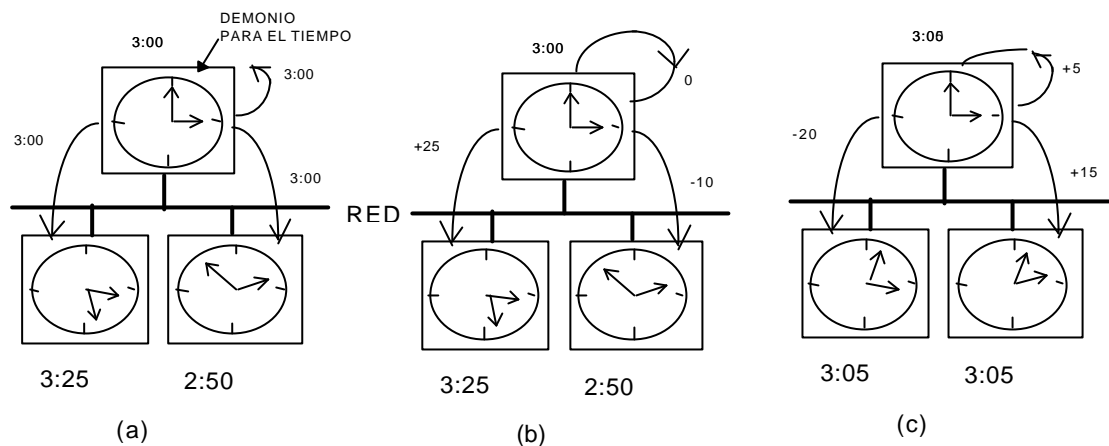


Fig. 11.20 (a) El demonio para el tiempo pregunta a todas las otras máquinas por el valor de sus relojes, (b) Las máquinas contestan, (c) El demonio para el tiempo le dice a todas la forma de ajustar sus relojes

Algoritmo con promedio

Los dos métodos anteriores son altamente centralizados, con sus desventajas usuales. También se conocen algoritmos descentralizados. Una clase de algoritmos de reloj descentralizados trabaja al dividir el tiempo en intervalos de resincronización de longitud fija. El i -ésimo intervalo inicia en $T_0 + iR$ hasta $T_0 + (i+1)R$, donde T_0 es un momento ya acordado en el pasado y R es un parámetro del sistema. Al inicio de cada intervalo, cada máquina transmite el tiempo actual según su reloj. Puesto que los relojes de diversas máquinas no corren precisamente a la misma velocidad, estas transmisiones no ocurrirán exactamente en forma simultánea.

Después de que una máquina transmite su hora, inicia un reloj local para reunir las demás transmisiones que lleguen en cierto intervalo S . Cuando llegan todas las transmisiones, se ejecuta un algoritmo para calcular una nueva hora para ellos. El algoritmo más sencillo consiste en promediar los valores de todas las demás máquinas.

Una ligera variación de este tema es descartar primero los m valores más grandes y los m valores más pequeños y promediar el resto. El hecho de descartar los valores extremos se puede considerar como autodefensa contra m relojes fallidos que envían mensajes sin sentido.

11.5.4. Mutua Exclusión en Sistemas Distribuidos

Los sistemas con varios procesos se programan más fácilmente mediante regiones críticas. Cuando un proceso debe leer o actualizar ciertas estructuras de datos compartidas, primero entra a una región crítica para lograr la mutua exclusión y garantizar que ningún otro proceso utilizara las estructuras de datos al mismo tiempo.

En los sistemas con un único procesador, las regiones críticas se protegen mediante semáforos, monitores y construcciones similares.

- **Algoritmos centralizados:** La forma más directa de lograr la mutua exclusión en un sistema distribuido es similar a la forma en que se lleva a cabo en un sistema con un único procesador. Se elige un proceso como el coordinador. Siempre que un proceso desea entrar a una región crítica, envía un mensaje de solicitud al coordinador, donde se indica a la región crítica que se desea entrar y pide permiso. Si ningún otro proceso está ocupando por el momento esa región crítica, el coordinador envía una respuesta otorgando el permiso. El método centralizado tiene limitaciones. El coordinador es el único punto de fallo, por lo que si se descompone, todo el sistema puede fallar. Además en un sistemas de grandes dimensiones el coordinador puede convertirse en un cuello de botella para el desempeño.
- **Algoritmos distribuidos:** Cuando un proceso desea entrar a una región crítica, construye un mensaje con el nombre de ésta, su número de proceso y la hora actual. Entonces envía el mensaje a todos los demás procesos y a sí mismo también. Cuando un mensaje envía se supone que tiene una función de reconocimiento por lo que se puede afirmar que es confiable. Cuando un proceso recibe un mensaje de solicitud de otro proceso, la acción que realiza depende de su estado con respecto de la región crítica nombrada en el mensaje.

Hay que distinguir 3 casos:

- Si el receptor no está en la región crítica y no desea entrar a ella, envía de regreso mensaje *OK* al emisor.
- Si el receptor ya está en la región crítica, no responde, sino que ingresa la solicitud en una cola.
- Si el receptor desea entrar a la región crítica, pero no lo ha logrado todavía, compara la marca de tiempo en el mensaje que envió cada uno. La menor de las marcas gana. Si el mensaje recibido es menor, el receptor envía un mensaje *OK*. Si su propio mensaje tiene una marca menor, el receptor ingresa la solicitud en una cola y no envía nada.

Después de enviar la solicitud que piden permiso para entrar a una región crítica, un proceso espera hasta que alguien más obtiene el permiso. Tan pronto llegan todos los permisos, se puede entrar a la región crítica. Cuando sale de ella, envía mensajes *OK* a todos los procesos en su cola y elimina a todos los elementos de la cola.

Con este algoritmo se elimina un único punto de fallo reemplazándolo por n puntos de fallo.

Un problema de este algoritmo es que se debe utilizar una primitiva de comunicaciones en grupo; o bien cada proceso debe mantener por sí mismo la lista de membresía del grupo, donde se incluyen los procesos que ingresan al grupo, los que salen de él y los que fallan.

Por último, recordemos que uno de los problemas con el algoritmo centralizado es que si pueden manejar todas las solicitudes, esto puede conducir a un cuello de botella. En el algoritmo distribuido, todos los procesos participan de todas las decisiones referentes a la entrada en las regiones críticas.

Un Algoritmo de TOKEN RING

Una red basada en un bus (por ej. Ethernet), sin un orden inherente en los procesos se muestra en la figura 11.21a. En software se construye un anillo lógico y a cada proceso se le asigna una posición en el anillo como se ve en la figura 11.21b. No importa como sea el orden, lo importante es que cada proceso sepa quien es el siguiente en la fila después de él.

Al inicializar el anillo se le da al proceso 0 una ficha, la cual circula por todo el anillo. Se transfiere del proceso K al $K+1$ (modulo del tamaño del anillo) en mensajes puntuales. Cuando un proceso obtiene la ficha de su vecino, verifica si intenta entrar a una región crítica. En ese caso, el proceso entra a la región, hace todo el trabajo necesario y sale de la región. Después de salir pasa la ficha a lo largo del anillo. No se permite entrar a un segundo proceso a la región crítica con la misma ficha.

Si un proceso recibe la ficha de su vecino y no está interesado en entrar a una región crítica, solo la vuelve a pasar. Cuando ninguno de los procesos desea entrar a una región crítica, la ficha solo circula a gran velocidad en el anillo.

El algoritmo tiene problemas en el caso que la ficha llegue a perderse, debe ser regenerada. También tiene problemas si falla un proceso, pero la recuperación es más sencilla que en los demás casos.

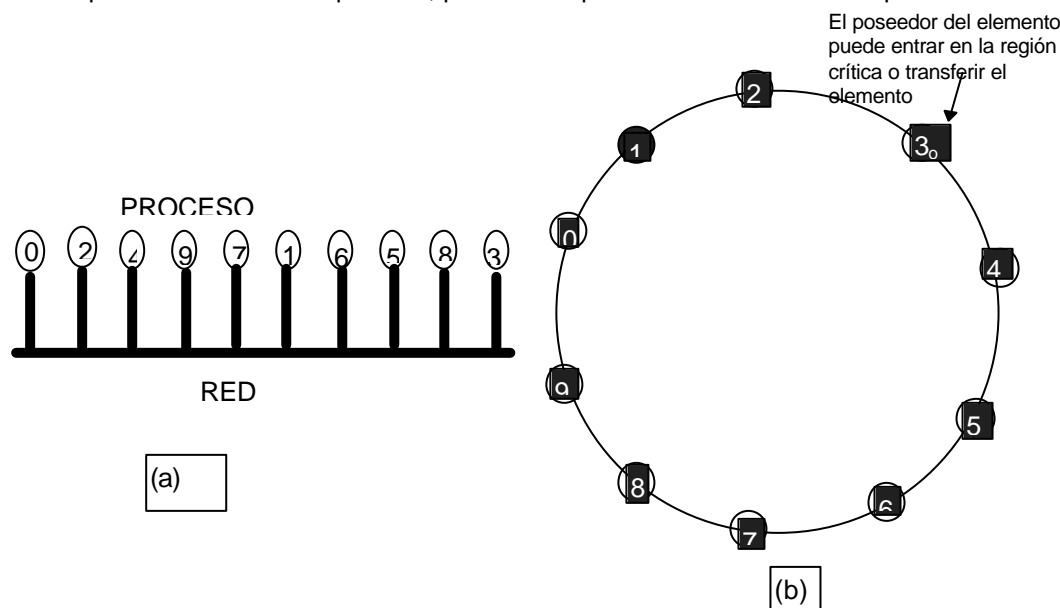


Fig. 11.21 (a) Un grupo no ordenado de procesos en una red (b) Un anillo lógico construido en software

Algoritmos de elección de Coordinadores

Estos algoritmos se encargan de elegir a un coordinador, este es el proceso que actúa como coordinador, iniciador, secuenciador o un desempeño de algún papel especial de coordinación.

EL objetivo de un algoritmo de elección es garantizar que al iniciar una elección, ésta concluya con el acuerdo de todos los procesos con respecto a la identidad del nuevo coordinador.

El algoritmo del grandulón (o mas “pesado”)

Cuando un proceso observa que el coordinador ya no responde a las solicitudes, inicia la elección. Un proceso *P* realiza una elección de la manera siguiente:

- *P* envía un mensaje *ELECCION* a los demás procesos con un número mayor.
- Si nadie responde, *P* gana la elección y se convierte en el coordinador.
- Si uno de los procesos con número mayor responde, toma el control. El trabajo de *P* termina.

En cualquier momento un proceso puede recibir un mensaje *ELECCION* de uno de sus compañeros con un número menor. Cuando llega dicho mensaje, el receptor envía de regreso un mensaje *OK* al emisor para indicar que está vivo y que tomará el control. El receptor realiza entonces una elección, a menos que ya este realizando alguna elección en ese instante. En cierto momento, todos los procesos se rinden, menos uno, el cual se convierte en el nuevo coordinador. Anuncia su victoria al enviar un mensaje a todos los procesos para indicarles que a partir de este momento es el nuevo coordinador.

Si un proceso inactivo se activa, realiza una elección. Si ocurre que es el proceso en ejecución con el número máximo, ganara la elección y tomará para sí el trabajo de coordinador.

Un algoritmo de anillo

Este se basa en el uso de un anillo pero sin ficha. Suponemos que los procesos tienen un orden, físico o lógico, de modo que cada proceso conoce a su sucesor. Cuando algún proceso observa que el coordinador no funciona, construye un mensaje *ELECCION* con su propio número de proceso y envía el mensaje a su sucesor. Si este está inactivo, el emisor pasa sobre el sucesor y va hacia el siguiente número del anillo o al siguiente de éste, hasta que localice un proceso en ejecución. En cada paso el emisor añade su propio número de proceso a la lista en el mensaje.

En cierto momento, el mensaje regresa a los procesos que lo iniciaron. Ese proceso reconoce este evento cuando recibe un mensaje de entrada con su propio número de proceso. En este punto, el mensaje escrito cambia a *COORDINADOR* y circula de nuevo, esta vez para informar a todos los demás quien es el coordinador y quienes son los miembros del nuevo anillo

11.6. Deadlocks (deadlock o abrazo mortal) en sistemas distribuidos

Los deadlocks en sistemas distribuidos son similares a los deadlocks en sistemas con un único procesador, solo que peores. Son más difíciles de evitar, prevenir e incluso detectar, además de ser más difíciles de curar cuando se le sigue la pista, puesto que toda la información relevante esta dispersa en muchas máquinas. En ciertos sistemas, como los sistemas distribuidos de bases de datos, pueden ser extremadamente serios, por lo que es importante comprender sus diferencias con los deadlocks ordinarios y lo que se puede hacer con ellos.

Existen 4 estrategias usuales para el manejo de los deadlocks:

- El algoritmo del avestruz (ignorar el problema)
- Detección (permitir que ocurran los deadlocks, detectarlos e intentar recuperarse de ellos)
- Prevención (hacer que los deadlocks sean imposibles desde el punto de vista estructural)
- Evitarlos (evitar los deadlocks mediante la asignación cuidadosa de los recursos).

11.6.1. Detección distribuida de Deadlocks

La presencia de las transacciones atómicas en ciertos sistemas distribuidos es una diferencia conceptual fundamental. Cuando se detecta un Deadlock en un sistema operativo convencional, la forma de resolverlo es eliminar uno o más procesos. Esto produce uno o varios usuarios infelices o disgustados. Cuando se detecta un Deadlock en un sistema basado en transacciones atómicas, se resuelve abortando una o más transacciones. Pero las transiciones han sido diseñadas para oponerse a ser abortadas. cuando se aborta una transacción por haber contribuido a un Deadlock, el sistema restaura en primer lugar el estado que tenía antes de iniciar la transacción, punto en el cual puede volver a comenzar la misma. Con

un poco de suerte, tendrá éxito la segunda vez. Así, la diferencia es que las consecuencias de la eliminación de un proceso son mucho menos severas si se utilizan las transacciones que en caso que no se utilicen.

11.6.2. Detección centralizada de Deadlocks

Como primer intento, podemos utilizar un algoritmo centralizado para la detección de deadlocks y tratar de imitar al algoritmo no distribuido. Aunque cada máquina mantiene un grafo de recursos con sus propios procesos y recursos, un coordinador central mantiene un grafo de recursos de todo el sistema (la unión de todos los grafos individuales). Cuando el coordinador detecta un ciclo, elimina uno de los procesos para romper el Deadlock.

A diferencia del caso centralizado, donde se dispone de toda la información de manera automática en el lugar correcto, en un sistema distribuido esta información se debe enviar de manera explícita. Cada máquina mantiene el grafo de sus propios procesos y recursos. Existen varias posibilidades para llegar ahí. En primer lugar, siempre que se añada o elimine un arco al grafo de recursos, se puede enviar un mensaje al consumidor para informar de la actualización. En segundo lugar, cada proceso puede enviar de manera periódica una lista de los arcos añadidos o eliminados desde la última actualización. Este método necesita un número menor de mensajes que el primero. En tercer lugar, el coordinador puede pedir la información cuando la necesite.

Ninguno de estos métodos funciona bien. Una posible solución es utilizar el algoritmo de Lamport para disponer de un tiempo global.

El algoritmo analizado es el Chandy-Mistra-Hass. En este algoritmo se permite que los procesos soliciten varios recursos (por ejemplo cerraduras) al mismo tiempo, en vez de uno cada vez. Al permitir las solicitudes simultáneas de varios procesos, la fase de crecimiento de una transacción se puede realizar mucho más rápido. La consecuencia de este cambio al modelo es que un proceso puede esperar ahora a dos o más recursos en forma simultánea.

11.6.3. Prevención distribuida de Deadlocks

La prevención de deadlocks consiste en el diseño cuidadoso del sistema, de modo que los deadlocks sean imposibles, desde el punto de vista estructural. Entre las distintas técnicas se incluye el permitir a los procesos que solo conserven un recurso a la vez, exigir a los procesos que soliciten todos sus recursos desde un principio y hacer que todos los procesos liberen todos los recursos cuando soliciten uno nuevo. Todo esto es difícil de manejar en la práctica. Un método que a veces funciona es ordenar todos los recursos y exigir a los procesos que los adquieran en orden estrictamente creciente. Este método significa que un proceso nunca puede conservar un recurso y pedir otro menor, por lo que son imposibles los deadlocks.

Sin embargo, en un sistema distribuido con tiempo global y transacciones atómicas, son posibles otros algoritmos prácticos. Ambos se basan en la idea de asociar a cada transacción una marca de tiempo global al momento de su inicio. Como en muchos de los algoritmos basados en las marcas de tiempo, en estos dos es esencial que ninguna pareja de transacciones tengan asociadas las mismas marcas de tiempo.

La idea detrás de este algoritmo es cuando un proceso está a punto de bloquearse en espera de un recurso que está utilizando otro proceso, se verifica cuál de ellos tiene la marca de tiempo mayor (es decir, es más joven). Podemos permitir entonces la espera sólo si el proceso en estado de espera tiene una marca inferior (más viejo) que el otro. De esta forma, al seguir cualquier cadena de procesos en espera, las marcas siempre aparecen en forma creciente, de modo que los ciclos son imposibles. Otra alternativa consiste en permitir la espera de procesos sólo si el proceso que espera tiene una marca mayor (es más joven) que el otro proceso, en cuyo caso las marcas aparecen en la cadena de forma decreciente.

Aunque ambos métodos previene los deadlocks, es más sabio dar prioridad a los procesos más viejos. Se han ejecutado durante más tiempo, por lo que el sistema ha invertido mucho en ellos y es probable que conserven más recursos. Además, un proceso joven eliminado en esta etapa llegará en cierto momento al punto en que sea el más antiguo del sistema, de modo que esta opción elimina la inanición. Como ya hemos señalado antes la eliminación de una transacción no causa daños en términos relativos, ya que por definición puede volver a iniciar más tarde.

11.6.4. Ordenación de Sucesos en un Sistema Distribuido

La ordenación temporal de sucesos es fundamental para la operación de la mayoría de los algoritmos distribuidos de mutua exclusión y Deadlock. La carencia de reloj común o de un medio de sincronizar los relojes locales es, por lo tanto, la restricción principal.

Cola Distribuida

Uno de los primeros enfoques propuestos para mantener la mutua exclusión distribuida está basado en el concepto de cola distribuida. El algoritmo se basa en los siguientes supuestos:

- 1- Un sistema distribuido consta de N nodos, numerados de forma única desde 1 hasta N. Cada nodo alberga un proceso que hace peticiones de acceso mutuamente exclusivo a recursos de parte de otros procesos; este proceso también sirve como un árbitro para resolver las peticiones entrantes que se solapan en el tiempo.
- 2- Los mensajes enviados desde un proceso a otro se reciben en el mismo orden en que fueron enviados.
- 3- Cada mensaje se entrega correctamente a su destino dentro de un tiempo finito.
- 4- La red está totalmente conectada, lo que significa que cada proceso puede enviar directamente mensajes a cualquier otro proceso sin necesidad de que un proceso intermedio pase el mensaje.

Enfoque de Paso de Testigo

Un conjunto de investigadores han propuesto un enfoque bastante diferente para la mutua exclusión que consiste en pasar un testigo (token) entre los procesos participantes. El testigo es una entidad que, en un momento dado, es retenida por un proceso. El proceso que posee el testigo puede entrar a su sección crítica sin pedir permiso. Cuando el proceso abandona su sección crítica, pasa el testigo a otro proceso.

11.7. TRANSACCIONES ATÓMICAS

Para que el programador no se enfrente directamente con los detalles de la mutua exclusión, el manejo de las regiones críticas, prevención de deadlocks y recuperación de un fallo (que son aspectos de bajo nivel), se usa una abstracción de mayor nivel que oculta estos aspectos técnicos y permite a los programadores que se concentren en los algoritmos y en la forma en que los procesos trabajan juntos en paralelo. Esta abstracción se llama una **transacción atómica**.

Introducción a las transacciones atómicas

El modelo de la transacción atómica es el siguiente: un proceso anuncia que desea comenzar una transacción con uno o más procesos. Pueden negociar varias opciones, crear y eliminar objetos y llevar a cabo ciertas operaciones durante unos momentos. Entonces, el iniciador anuncia que desea que todos los demás se comprometan con el trabajo realizado hasta entonces. Si todos coinciden, los resultados se vuelven permanentes. Si uno o más procesos se niegan (o fallan antes de expresar su acuerdo), entonces la situación regresa al estado que presentaba antes de comenzar la transacción, sin que existan efectos colaterales en los objetos, archivos, base de datos, etc. Esta propiedad del todo o nada facilita el trabajo del programador.

11.7.1. EL MODELO DE LA TRANSACCIÓN

Almacenamiento estable

El almacenamiento tiene tres categorías. La *Memoria RAM* común, que es volátil al fallar el suministro de energía o un fallo de la máquina. El *Almacenamiento en Disco*, que sobrevive a las fallas de la CPU, pero que se puede perder debido a problemas de la cabeza lectora del disco. El almacenamiento estable, diseñado para sobrevivir a todo, excepto catástrofes mayores, como las inundaciones y terremotos. El almacenamiento estable se puede implantar con una pareja de discos ordinarios, como se muestra en la figura 11.22. Cada bloque en la unidad 2 es una copia exacta del bloque correspondiente en la unidad 1. Esta técnica se conoce como mirroring. Cuando se actualiza un bloque, primero se actualiza y verifica el bloque de la unidad 1, para después encargarse del mismo bloque en la unidad 2.

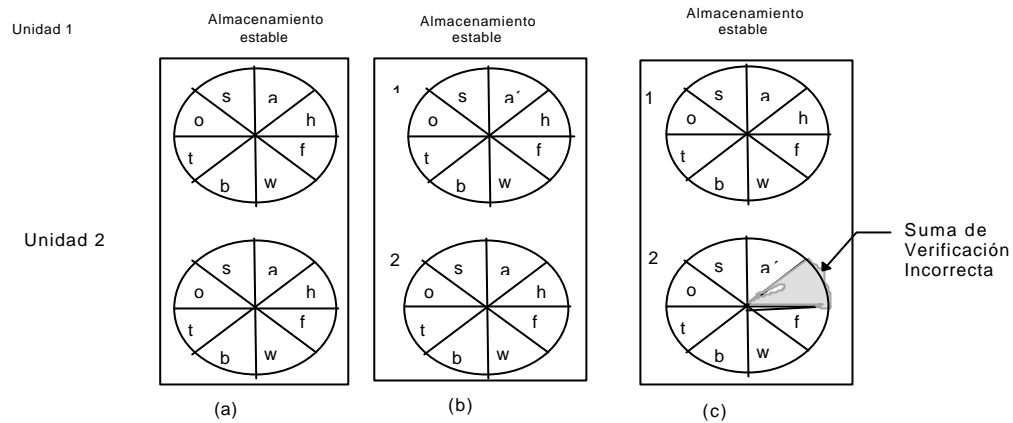


Fig 11.22 (a) Almacenamiento estable (b) Fallo después de la actualización de la unidad 1 (c) Punto negro

Si el sistema falla después de la actualización de la unidad 1, pero antes de actualizar la unidad 2. Después de la recuperación, el disco se puede comparar bloque por bloque. En caso que 2 bloques correspondientes difieran, se puede suponer que la unidad 1 es la correcta por lo que el nuevo bloque se copia de la unidad 1 a la unidad 2. Al terminar el proceso de recuperación, ambas unidades vuelven a ser idénticas.

Primitivas de Transacción

La programación con uso de transacciones requiere de primitivas especiales, las cuales debe proporcionar el sistema operativo o por el sistema de tiempo de ejecución del lenguaje. Algunos ejemplos son:

- **BEGIN_TRANSACTION.** Los comandos siguientes forman una transacción.
- **END_TRANSACTION.** Termina la transacción y se intenta un compromiso.
- **ABORT_TRANSACTION.** Se elimina la transacción; se recuperan los valores anteriores.
- **READ.** Se leen datos de un archivo (o algún otro objeto).
- **WRITE.** Se escriben datos en un archivo (o algún otro objeto).

La lista exacta de primitivas depende del tipo de objetos que se utilicen en la transacción. En un sistema de correo, podrían existir primitivas para el envío, recepción y direccionamiento del correo. En un sistema de contabilidad, podrían ser un poco distintas. READ y WRITE son ejemplos típicos.

BEGIN_TRANSACTION y END_TRANSACTION se utilizan para establecer los límites de una transacción. Las operaciones entre ellas forman el cuerpo de la transacción. Todas o ninguna de ellas deben ejecutarse.

Propiedades de las transacciones

Las transacciones tienen 3 propiedades fundamentales. Estas son:

- **Serialización:** Las transacciones concurrentes no interfieren entre sí.
- **Atomicidad:** Para el mundo exterior, la transacción ocurre de manera indivisible.
- **Permanencia:** Una vez comprometida una transacción, los cambios son permanentes.

La serialización garantiza que si dos o más transacciones se ejecutan al mismo tiempo, para cada una de ellas y para los demás procesos, el resultado final aparece como si todas las transacciones se ejecutasen de manera secuencial en cierto orden.

La atomicidad, garantiza que cada transacción no ocurre o bien, se realiza en su totalidad, en cuyo caso se presenta como una acción instantánea e indivisible. Mientras se desarrolla una transacción, otros procesos (ya sea que estén relacionados o no con las transacciones) no pueden ver los estados intermedios.

La permanencia, se refiere al hecho de que una vez comprometida una transacción, no importa lo que ocurra, la transacción sigue adelante y los resultados se vuelven permanentes. Ningún fallo después del compromiso puede deshacer los resultados o, provocar la pérdida de los mismos.

Las transacciones pueden contener subtransacciones, a menudo llamadas transacciones anidadas. La transacción de nivel superior puede producir hijos que se ejecuten en paralelo entre sí, en procesos distintos, con el fin de mejorar el desempeño o hacer más sencilla la programación. Cada una de estos hijos puede ejecutar una o más subtransacciones o bien producir sus propios hijos.

IMPLANTACIÓN

Espacio de trabajo particular o privado

Desde un punto de vista conceptual, cuando un proceso inicia una transacción, se le otorga un espacio de trabajo particular, el cual contiene todos los archivos (y otros objetos) a los cuales tiene acceso. Hasta que la transacción se comprometa o aborte, todas sus lecturas y escrituras irán al espacio de trabajo particular, en vez del espacio "real", donde éste último es el sistema de archivo normal. Esta observación conduce de manera directa al primer método de implantación: otorgar un espacio particular a cada proceso en el momento en que inicie una transacción.

El problema con esta técnica es el costo prohibitivo de copiar todo en un espacio de trabajo particular, pero se puede hacer ciertas optimizaciones. La primera de ellas se basa en el hecho de que, cuando un proceso lee un archivo, pero no lo modifica, no existe necesidad de una copia particular. Puede utilizar la verdadera (a menos que haya sido modificada después del inicio de una transacción).

Log² de escritura anticipada

Con este método, los archivos realmente se modifican, pero antes de cambiar cualquier bloque, se escribe un registro en el **log de escritura anticipada** en un espacio de almacenamiento estable para indicar la transacción que realiza el cambio, el archivo y bloque modificados y los valores anteriores y nuevo. Sólo, después de que se puede escribir en la log se realiza el cambio en el archivo.

Si la transacción tiene éxito y se hace un compromiso, se escribe un registro de éste último en el log, pero las escrituras de datos no tiene que modificarse, puesto que ya han sido actualizadas. Si la transacción aborta, se puede utilizar el log para respaldo del estado original. A partir del final y hacia atrás, se lee cada registro del log y se deshace cada cambio descrito en él. Esta acción se llama retroalimentación.

El log también se puede utilizar para la recuperación de los fallos.

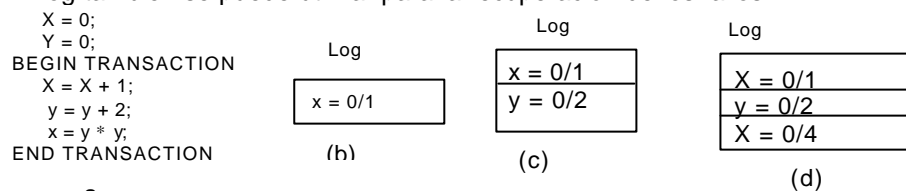


Fig. 11.23 (a) Una transacción - (b) (c) y (d) El log antes de ejecutar cada enunciado

Protocolo de compromiso de dos fases³

La acción de establecer un compromiso con una transacción debe llevarse a cabo de manera atómica es decir, de forma instantánea e indivisible. En un sistema distribuido el compromiso puede necesitar la cooperación de varios procesos en distintas máquinas, cada una de los cuales contenga algunas de las variables, archivos y bases de datos y otros objetos modificados por la transacción.

El protocolo de compromiso de dos fases es tal vez el más utilizado.

Uno de los procesos funciona como el coordinador. Por lo general éste es el que ajusta la transacción. El protocolo de compromiso comienza cuando el coordinador escribe una entrada en el log para indicar que inicia dicho protocolo, seguido del envío a cada uno de los procesos relacionados (subordinados) de un mensaje para que estén preparados para el compromiso.

Cuando un subordinado recibe el mensaje, verifica si está listo para comprometerse, escribe una entrada en el log y envía de regreso su decisión. Cuando el coordinador ha recibido todas la respuestas, sabe si puede establecer el compromiso o abortar.

Si todos los procesos están listos para comprometerse entonces se cierra la transacción. Si uno o más procesos no se comprometen, la transacción se aborta. De cualquier modo, el coordinador escribe una entrada en el log y envía entonces un mensaje a cada subordinado para indicarle de la decisión. Es esta escritura en el log que en realidad cierra la transacción y hace que camine hacia adelante sin importar ya lo ocurrido. Este proceso es altamente alterable en presencia de fallos.

Control de concurrencia

Cuando se ejecutan varias transacciones de manera simultáneas en distintos procesos (o distintos procesadores), se necesita cierto mecanismo para mantener a cada uno lejos del camino del otro. Este mecanismo se llama *algoritmo de control de concurrencia*.

Cerradura (locking)

El algoritmo más antiguo y de más amplio uso es la cerradura puesto por un candado. En su forma más sencilla, cuando un proceso necesita leer o escribir un archivo (u otro objeto) como parte de una

² Algunos traducen log como bitácora

³ protocolo de compromiso de dos fases en inglés: two phase commitment protocol

transacción, primero le pone un candado al archivo en uso. La cerradura se puede hacer mediante un único manejador centralizado de cerraduras, o bien con un manejador local de cerraduras en cada máquina, el cual maneja los archivos locales. En ambos casos, el manejador de cerraduras mantiene una lista de los archivos bloqueados con candados por otros procesos. Puesto que los procesos de buen comportamiento no intentan acceder a un archivo si este tiene un candado cerrado, el establecimiento de una cerradura en un archivo bloquea a todos los accesos a éste, lo cual garantiza que no será modificado durante la vida de la transacción. El sistema de transacciones es el que, por lo general, adquiere y libera las cerraduras y no necesita acción alguna por parte del programador.

Este esquema básico es demasiado restrictivo y se puede mejorar al distinguir las cerraduras para lecturas de las cerraduras para escritura.

Control optimista de la concurrencia

Un segundo método para el manejo de varias transacciones al mismo tiempo es el control optimista de la concurrencia. La técnica de esta idea es la siguiente:

—Solo se sigue adelante y se hace todo lo que se deba llevar a cabo, sin prestar atención a lo que hacen los demás. Si existe un problema, hay que preocuparse por él después. En la práctica, los conflictos son sumamente raros, por lo que la mayoría del tiempo todo funciona muy bien.

Aunque los conflictos pueden ser raros, no son imposibles, por lo que se necesita manejarlos de alguna forma. Lo que hace el control optimista de la concurrencia es mantener un registro de los archivos leídos o en los que se ha escrito algo. En el momento del compromiso, se verifican todas las demás transacciones para ver si alguno de los archivos ha sido modificado desde el inicio de la transacción. Si esto ocurre, la transacción aborta. Si no, se realiza el compromiso.

Las ventajas de esta técnica son la ausencia de deadlocks y que permite un paralelismo máximo, puesto que ningún proceso tiene que esperar una cerradura.

La desventaja es que a veces puede fallar, en cuyo caso la transacción tiene que ejecutarse de nuevo.

Marcas de tiempo

Un método completamente distinto al control de la concurrencia consiste en asociar a cada transacción una marca de tiempo, al momento en que realiza `BEGIN_TRANSACTION`. Mediante el algoritmo de Lamport, podemos garantizar que las marcas son únicas, lo cual es importante en este caso. Cada archivo del sistema tiene asociadas una marca de tiempo para la lectura y otra para la escritura, las cuales indican la última transacción comprometida que realizó la lectura o escritura, respectivamente. Si las transacciones son breves y muy espaciadas con respecto al tiempo, entonces ocurrirá de manera natural que cuando un proceso intente tener acceso a un archivo, las marcas de tiempo de lectura y escritura sean menores (más antiguas) que la marca de transacción activa. Este orden quiere decir que las transacciones se procesan en el orden adecuado, por lo que todo funciona bien.

Cuando el orden es incorrecto, esto indica que una transacción iniciada posteriormente a la transacción activa ha intentado entrar al archivo o ha tenido acceso a éste y ha realizado un compromiso. Esta situación indica que la transacción activa se ha realizado tarde, por lo que se aborta.

No nos preocupa que las transacciones concurrentes utilizan los mismos archivos, siempre que la transacción con el número más pequeño este en primer lugar.

Las marcas de tiempo tienen propiedades distintas a las de los deadlocks. Cuando una transacción encuentra una marca mayor (posterior) aborta, mientras que con las cerraduras, en las mismas circunstancias, podría esperar o poder continuar de manera inmediata. Por otro lado, siempre está libre de deadlocks, que es un punto a su favor.

Todas las transacciones ofrecen distintas ventajas y son una técnica promisorio para la construcción de sistemas distribuidos confiables. Su problema principal es la enorme complejidad de su implantación, lo que provoca un bajo desempeño.

11.8. Sistemas de Archivos Distribuidos

Por lo general, un sistema distribuido de archivos tiene dos componentes razonablemente distintos: el verdadero servicio de archivos y el servicio de directorios. El primero se encarga de las operaciones en los archivos individuales, como la lectura, escritura y adición, mientras que el segundo se encarga de crear y manejar directorios, añadir y eliminar archivos de los directorios, etc.

La protección de archivos en los sistemas distribuidos utiliza en esencia las mismas técnicas de los sistemas con un único procesador, como ser dominios y listas para control de acceso. En el caso de los dominios cada usuario tiene un cierto tipo de permiso, llamado capacidad, para cada objeto al que tiene

acceso. La capacidad determina los tipos de accesos permitidos. Los esquemas de *lista para control de acceso* le asocian a cada archivo una lista implícita o explícita de los usuarios que pueden tener acceso al archivo y los tipos de acceso permitidos a cada uno de ellos. Los servicios de archivos se pueden dividir en dos tipos, a saber:

- modelo de carga/descarga
- modelo de acceso remoto

11.8.1. El modelo de carga/descarga (Upload / Download):

En el primer modelo, el servicio de archivo sólo proporciona dos operaciones principales: la lectura del archivo y la escritura del mismo. La primera operación transfiere todo un archivo de uno de los servidores de archivos al cliente solicitante. La segunda operación transfiere todo un archivo en sentido contrario, del cliente al servidor. Los programas de aplicación buscan los archivos que necesitan y después los utilizan de manera local. Los archivos modificados o nuevos se escriben de regreso al terminar el programa. No hay que manejar una complicada interfase del servicio de archivos para utilizar este modelo. La principal desventaja de este modelo es que el cliente debe disponer de un espacio suficiente de almacenamiento para todos los archivos necesarios.

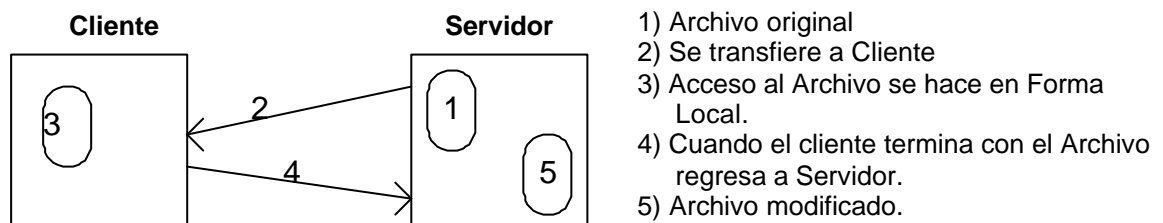


Fig. 11.24 Modelo Carga / Descarga

→ Ventajas:

- Modelo sencillo
- Interfase de servicios simple.

→ Desventajas:

- Cliente debe disponer de espacio para archivo o archivos.
- Si solo se necesita fracción del Archivo porque una transferencia completa es un desperdicio de tiempo y recursos.

11.8.2. El modelo de acceso remoto

En el segundo modelo, el servicio de archivos proporciona un gran número de operaciones para abrir y cerrar archivos, leer y escribir partes de archivos, moverse a través de un archivo, examinar y modificar los atributos de archivo, etc. En este caso el sistema de archivos se ejecuta en los servidores de los clientes, a la vez que elimina la necesidad de transferir archivos completos cuando sólo se necesitan una pequeña parte de ellos.

Acceso Remoto:

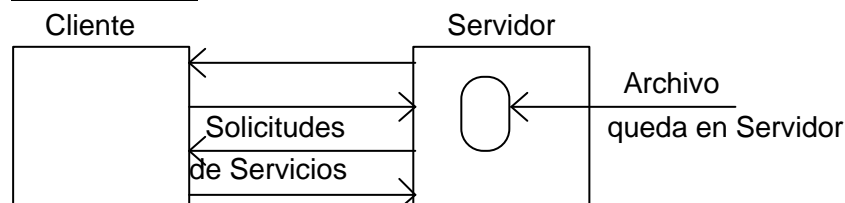


Fig. 11.25 Modelo de acceso remoto

- Todo se ejecuta en Servidor.
- No se necesita espacio en el Cliente.
- No se necesita Transferencia Total del Archivo.

La otra parte del servicio de archivos es el *servicio de directorios*, el cual proporciona las operaciones para crear y eliminar directorios, nombrar o cambiar el nombre de archivos y mover éstos de un directorio a otro. Lo usual es que los nombres de archivos tengan un cierto número máximo de letras, números y ciertos caracteres especiales. Algunos sistemas dividen los nombres de archivo en dos partes,

usualmente separadas mediante un punto. La segunda parte del nombre llamada *extensión de archivo* identifica el tipo de archivo. Todos los sistemas distribuidos permiten que los directorios contengan subdirectorios, para que los usuarios puedan agrupar los archivos relacionados entre sí. Los subdirectorios pueden contener sus propios subdirectorios y así sucesivamente, lo que conduce a un árbol de directorios creado así un sistema jerárquico de archivos. En ciertos sistemas, es posible crear enlaces o apuntadores a un directorio arbitrario. Estos se pueden colocar en cualquier directorio, lo que permite construir no sólo árboles, sino gráficas arbitrarias de directorio, que son más poderosas. Un aspecto fundamental en el diseño de cualquier sistema distribuido de archivos es si todas las máquinas y procesos deben tener exactamente la misma visión de la jerarquía de los directorios.

Existen tres métodos usuales para nombrar los archivos y directorios en un sistema distribuido:

- Nombre máquina + ruta de acceso
- Montaje de sistemas de archivos remotos en la jerarquía local de archivos.
- Un único espacio de nombres que tenga la misma apariencia en todas las máquinas.

Los primeros dos son fáciles de implantar, especialmente como una forma de conectar sistemas ya existentes que no estaban diseñados para uso distribuido. El tercer método es difícil y requiere de un diseño cuidadoso, pero es necesario si se quiere lograr el objeto de que el sistema distribuido actúe como una única computadora.

La mayoría de los sistemas distribuidos utilizan cierta forma de nombres con dos niveles. Los archivos tienen *nombres simbólicos*, para uso de las personas, pero también pueden tener *nombres binarios* internos, para uso del propio sistema. Lo que los directorios hacen en realidad es proporcionar una asociación entre estos dos nombres. Para las personas y los programas, es conveniente utilizar nombres simbólicos, pero para el uso dentro del propio sistema, estos nombres son demasiado grandes y difíciles. Así cuando un usuario abre un archivo o hace referencia a un nombre simbólico, el sistema busca de inmediato el nombre simbólico en el directorio apropiado para obtener el nombre binario, el cual utilizará para localizar realmente el archivo.

Resumen de búsquedas en directorios:

- 1) Cliente manda Mensaje a Servidor 1 que maneja su Directorio de trabajo.
 - 2) Server encuentra "a" pero ve que el nombre en binario corresponde a otro Server.
- A partir de acá tiene dos opciones:
- 3) Le dice al cliente en que Server está "b"
 - 3) Envía resto de la solicitud al Servidor 2.
 - 4) Idem 1), 2) y 3)
 - 4) Idem 1), 2) y 3)
- Uso de más mensajes.
 - Más eficiente.
 - Menos transparente para Usuario.
 - Pero no se puede usar RPC.

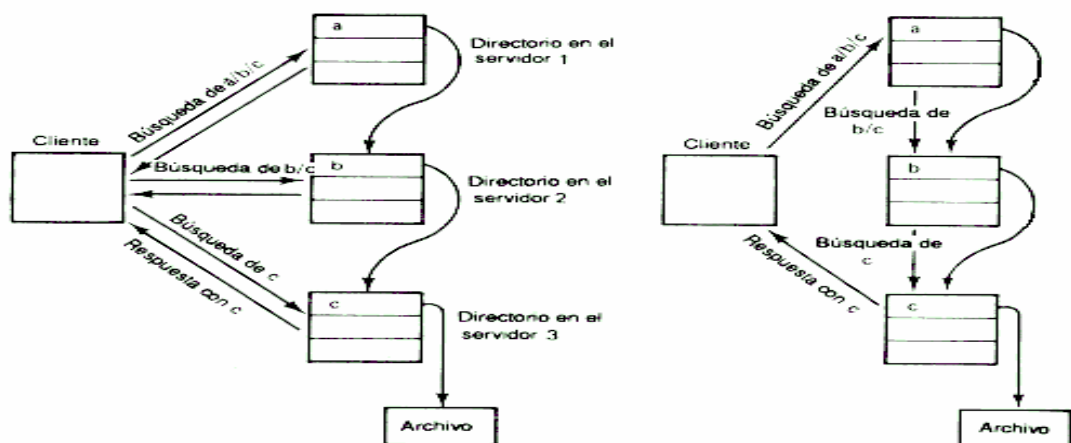


Fig. 11.26 Búsquedas en directorios Remotos.

Si dos o más usuarios comparten el mismo archivo, es necesario definir con precisión la **semántica** de la lectura y escritura para evitar problemas. Todas las instrucciones *read* y *write* pasan en forma directa al servidor de archivos, que los procesa en forma secuencial. El desempeño de un sistema distribuido en donde todas las solicitudes de archivos deban pasar a un único servidor es pobre. Este problema se puede resolver si se permite a los clientes que mantengan copias locales de los archivos de uso frecuente en sus cachés particulares. Si un cliente modifica en forma local un archivo en caché y poco después otro cliente lee el archivo del servidor, el segundo cliente obtendrá un archivo obsoleto. Una forma de salir de esta dificultad es propagar de manera inmediata todas las modificaciones de los archivos en caché de regreso al despachador.

El aspecto importante a considerar en este punto es si los servidores de archivos, directorios o de otro tipo deben contener la información de estado de los clientes. Este aspecto tiene una controversia moderada, donde existen dos escuelas de pensamiento en competencia. Una escuela piensa que los servidores no deben contener los estados, es decir, ser sin estado. O sea, que cuando un cliente envía una solicitud a un servidor, éste la lleva a cabo, envía la respuesta y elimina de sus tablas internas toda la información relativa a dicha solicitud. El servidor no guarda información alguna relativa a los clientes entre las solicitudes. La otra escuela de pensamiento sostiene que es correcto que los servidores conserven información de estado de los clientes entre las solicitudes. La otra escuela sostiene que es correcto que los servidores conserven información de estado de los clientes entre las solicitudes.

En un sistema cliente-servidor, cada uno con su memoria central y un disco, existen cuatro lugares donde se pueden almacenar los archivos o parte de archivos:

- Disco del servidor.
- Memoria central del servidor.
- Disco del cliente.
- Memoria central del cliente.

Estos lugares de almacenamiento tienen distintas propiedades.

El lugar más directo para almacenar todos los archivos es el disco del servidor, Ahí existe mucho espacio y los archivos serían accesibles a todos los clientes con sólo una copia de cada archivo, no surgen problemas de consistencia. El problema con el uso del disco del servidor es el desempeño. Antes de que un cliente pueda leer un archivo, éste debe ser transferido primero del disco del servidor a la memoria central del servidor y luego, a través de la red, a la memoria central del cliente. Ambas transferencias tardan cierto tiempo. Se puede lograr un desempeño mucho mejor si se ocultan los archivos de más reciente uso en la memoria central del servidor. Un cliente que lea un archivo ya presente en el caché del servidor elimina la transferencia del disco, aunque se deba realizar la transferencia a la red. Puesto que la memoria central es menor que el disco, se necesita un algoritmo para determinar los archivos o parte de archivos que deben permanecer en el caché.

Este algoritmo debe resolver dos problemas. El primero es el de la unidad que maneja el caché. Puede manejar archivos completos o bloques del disco. Si se ocultan los archivos completos, éstos se pueden almacenar en forma adyacente en el disco, lo cual permite transferencias a alta velocidad entre la memoria y el disco, así como un buen desempeño en general. El ocultamiento de bloque de disco utiliza el caché y el espacio en disco en forma más eficiente. El segundo, es que el algoritmo debe decidir qué hacer si se utiliza toda la capacidad del caché y hay que eliminar a alguien. Si existe una copia actualizada en el disco, simplemente se descarta la copia del caché. En caso contrario, primero se actualiza el disco. El mantenimiento de un caché en la memoria central del servidor es fácil de lograr y es totalmente transparente a los clientes. Puesto que el servidor puede mantener sincronizadas sus copias en memoria y en disco, desde el punto de vista de los clientes sólo existe una copia de cada archivo, por lo que no hay problemas de consistencia.

Aunque el uso del caché en el servidor elimina una transferencia de disco en cada acceso, tiene aún un acceso a la red. La única forma de deshacerse del acceso a la red es hacer un ocultamiento en el lado del cliente, que es donde aparecen todos los problemas. El uso de la memoria central del cliente o su disco es un problema de espacio vs. desempeño. El disco puede contener más información, pero es más lento. Al considerar las opciones de tener un caché en la memoria central del servidor o en el disco del cliente, la primera es generalmente más rápida y siempre es más sencilla.

Si se decide colocar el caché en la memoria central del cliente, existen tres opciones para su posición precisa. La más sencilla consiste en ocultar los archivos en forma directa, dentro del propio espacio de direcciones de un proceso usuario, lo usual es que el caché sea manejado por la biblioteca de llamadas al sistema. Cuando los archivos se abren, cierran, leen o escriben, la biblioteca mantiene los archivos de más uso en algún sitio, de modo que estén disponibles cuando se vuelvan a utilizar. Cuando el proceso hace su salida, todos los archivos modificados se escriben de nuevo en el servidor. Tiene un costo mínimo, sólo es eficaz si los procesos individuales abren y cierran los archivos varias veces.

El segundo sitio donde se puede colocar el caché es el núcleo. La desventaja en este caso es que siempre hay que llamar al núcleo, incluso en aquellos casos en que los archivos estén dentro del caché, pero que el caché sobreviva al proceso compensa en mucho este hecho.

El tercer lugar donde se puede colocar el caché es en un proceso manejador del caché, independiente y a nivel usuario. La ventaja de un manejador del caché a nivel usuario es que libera al (micro) kernel del código del sistema de archivos, es más fácil de programar y es más flexible.

11.8.3. Replicación de archivos

Con frecuencia, los sistemas distribuidos de archivos proporcionan la *réplica* de archivos como servicio a sus clientes. Se dispone de varias copias de algunos archivos, donde cada copia está en un servidor de archivos independientes

Un aspecto clave en la réplica es la transparencia. En un caso extremo, los usuarios pueden tener total conciencia del proceso de réplica e incluso lo pueden controlar. En el otro extremo, el sistema hace todo a espaldas de los usuarios. En el segundo caso, se dice que el sistema es transparente con respecto a la réplica. Existen tres formas de llevar a cabo la réplica, a saber:

- Réplica explícita de archivos
- Réplica retrasada de archivos
- Réplica de archivos mediante un grupo

Réplica explícita de archivos

En esta forma el programador controla todo el proceso. Cuando un proceso crea un archivo, lo hace en un servidor específico. Entonces puede hacer copias adicionales en otros servidores, si así lo desea. Si el servidor de directorios permite varias copias de un archivo, las direcciones en la red de todas las copias se pueden asociar con el nombre del archivo, de modo que cuando se busque el nombre, se encuentren todas las copias. Cuando el archivo se vuelve a abrir, se pueden buscar las copias de manera secuencial en cierto orden, hasta encontrar una disponible.

Resumen sobre Réplica explícita:

- No es propia de SD.
- Después de crear un archivo que se quiere replicar:
 - Cliente crea copias en diferentes lugares.
 - A un mismo nombre le tiene que vincular varias direcciones.
 - La búsqueda es secuencial hasta encontrar un archivo disponible.

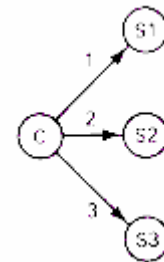


Fig. 10.27

Réplica retrasada de archivos (Lazy Replication)

En esta forma, sólo se crea una copia de cada archivo en un servidor. más tarde, el propio servidor crea réplicas en otros servidores en forma automática, sin el conocimiento del programador. El sistema debe ser lo bastante hábil como para recuperar algunas de estas copias en caso necesario. Al hacer copias secundarias de esta manera, es importante poner atención en el hecho de que el archivo podría cambiar antes de que se hagan las copias.

Resumen sobre Réplica retrasada:

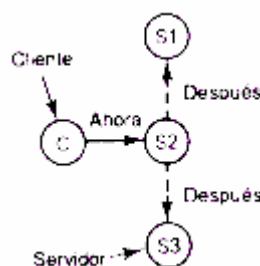


Fig. 10.28

- Solo se crea un Archivo.
- Sistema se ocupa de hacer otras copias cuando tiene Tiempo.
- Sistema tiene que poder encontrar las otras copias.
- Se direcciona un Servidor.

→ **Problema:** Si se cambia el Archivo y la copia no fue efectuada todavía entonces se accede a una Copia Obsoleta.

Réplica de archivos mediante un grupo

En la comunicación en grupo, todas las llamadas al sistema *write* se transmiten en forma simultánea a todos los servidores a la vez, por lo que las copias adicionales se hacen al mismo tiempo que el original.

Existen dos diferencias fundamentales entre la réplica retrasada y el uso de un grupo. En primer lugar, con la primera se direcciona a un servidor y no un grupo. En segundo lugar, la réplica retrasada ocurre en un segundo plano cuando el servidor tiene cierto tiempo libre, mientras que con el uso de grupos de comunicación en grupo, todas las copias se crean al mismo tiempo.

Con anterioridad se describió el problema de la creación de réplicas de archivos. Ahora se describirá cómo se pueden modificar los ya existentes. Existen dos algoritmos conocidos para la solución de este problema. El primero es el de la **réplica de la copia primaria**. Uno de los servidores se denomina

como primario. Todos los demás son secundarios. Si hay que actualizar un archivo duplicado, el cambio se envía comandos a los secundarios para ordenarles la misma modificación. Las lecturas se pueden hacer de cualquier copia, primaria o secundaria. Para protegerse de la situación en que falle el primero antes de que pueda dar instrucciones a todos los secundarios, la actualización debe escribirse a un espacio estable de almacenamiento antes de modificar la copia primaria. De este modo, cuando un servidor vuelve a arrancar después de un fallo, se pueden llevar a cabo. En algún momento, todos los secundarios se actualizarán. Este método tiene la desventaja de que si falla el primario, no se pueden llevar a cabo las actualizaciones.

El segundo método, el del **voto**. La idea fundamental es exigir a los clientes que soliciten y adquieran el permiso de varios servidores antes de leer o escribir un archivo replicado. Por ejemplo, suponiendo que un archivo se replica en N servidores. Se puede establecer que para actualizar un archivo, un cliente debe establecer la regla de que para actualizar un archivo, un cliente debe establecer el contacto con al menos la mitad de los servidores, más 1 (una mayoría) y ponerlos de acuerdo para llevar a cabo la actualización. Una vez de acuerdo, el archivo se modifica y se asocia un nuevo número de versión asociado a cada archivo. El número de versión se utiliza para identificar la versión del archivo y es la misma para todos los archivos recién actualizados. Para leer un archivo replicado, un cliente debe establecer también contacto con la mitad de los servidores, más 1 y pedirles que envíen el número de versión asociado a cada archivo. Si coinciden todos los números de versión, ésta debe ser la más reciente, puesto que un intento por actualizar sólo a los servidores restantes fracasará, porque no son un número suficiente.

Resumen sobre Réplica mediante Grupo:

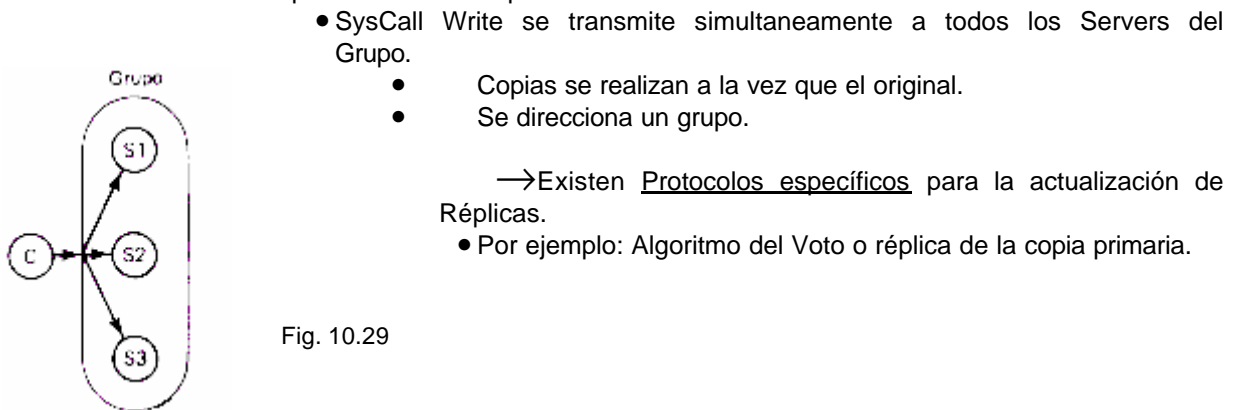


Fig. 10.29

- SysCall Write se transmite simultáneamente a todos los Servers del Grupo.
- Copias se realizan a la vez que el original.
- Se direcciona un grupo.

→Existen Protocolos específicos para la actualización de Réplicas.

- Por ejemplo: Algoritmo del Voto o réplica de la copia primaria.

Resumen sobre Réplica de archivos en procesamiento distribuido (DFS):

- DFS brinda servicio de Réplica.
- Copias de un mismo Archivo en varios dispositivos.
- Razones:
- Confiabilidad. Si falla un Server, otro proporciona el Archivo.
- Load Sharing. Cuando crece el sistema para no tener cuello de botella.
- Grados de la Transparencia de la Réplica:
- No transparente:
- Lo hace el usuario.
- Debe tener conciencia de donde lo pone.
- Transparente: es propia DFS
 - Lo hace el sistema sin que lo note el usuario.
 - Sistema se ocupa de buscar copias

11.9. LOS PROBLEMAS EN LOS SISTEMAS DISTRIBUIDOS

La siguiente parte de este módulo tiene por fin realizar un rápido vistazo sobre los principales problemas que presenta el modelo de Sistemas Operativos Distribuidos, que si bien tiene grandes ventajas sobre los sistemas operativos convencionales, poseen problemas que en estos últimos no se presentan, como el sincronismo, Broadcast, fault tolerance, etc.

11.9.1. Introducción a los problemas en los Sistemas Operativos Distribuidos

Habíamos afirmado que los sistemas distribuidos tienen un buen número de puntos a su favor. Pueden ofrecer una buena proporción precio/desempeño y se pueden ajustar bien a las aplicaciones distribuidas; pueden ser altamente confiables y pueden aumentar su tamaño de manera gradual, al aumentar la carga de trabajo. También tienen ciertas desventajas como el hecho de tener un software mas complejo, potenciales cuellos de botella en la comunicación y una seguridad débil. Sin embargo, existe un considerable interés mundial por su construcción e nodo de estos sistemas.

Los modernos sistemas de computo tienen con frecuencia varias CPUs. Estas se pueden organizar como multiprocesadores (con memoria compartida) o como multicomputadoras (sin memoria compartida). Ambos tipos pueden basarse en un bus o en un conmutador. Los primeros tienden a ser fuertemente acoplados, mientras que los segundos tienden a ser débilmente acoplados.

El software para los sistemas con varias CPUs se pueden dividir en tres clases:

- Los sistemas operativos de red permiten a los usuarios en estaciones de trabajo independientes la comunicación por medio de un sistema compartido de archivos, pero dejan que cada usuario domine su propia estación de trabajo.
- Los sistemas operativos distribuidos convierten toda la colección de hardware y software en un único sistema integrado, muy parecido a un sistema tradicional de tiempo compartido.
- Los multiprocesadores con memoria compartida también ofrecen la imagen de un único sistema, pero lo hacen mediante la vía de centralizar todo, por lo que en realidad, este caso es en realidad un único sistema. Los multiprocesadores con memoria compartida no son sistemas distribuidos.

Los sistemas distribuidos deben diseñarse con cuidado, puesto que existen muchas trampas para los incautos. Un aspecto clave es la transparencia: ocultar la distribución a los usuarios e incluso de los programas de aplicación. Otro aspecto es la flexibilidad.

Puesto que el campo se encuentra todavía en su infancia, el diseño se debe hacer con la idea de facilitar los cambios futuros. A este respecto, los microkernels son superiores a los núcleos monolíticos. Otros aspectos importantes son la confiabilidad, el desempeño y la escalabilidad.

Los Problemas en Fault Tolerance Distributed Computing han sido estudiados en gran variedad de modelos computacionales.

Estos modelos caen en dos categorías:

- Message Passing (pasaje de mensajes).
- Shared Memory (Memoria Compartida)

En el primero los procesos se comunican enviando y recibiendo mensajes por los vínculos (links) de la red. En el segundo se comunican accediendo a objetos compartidos, como registros, colas, etc.

Los parámetros que determinan un modelo particular de *message Passing* son Sincronía entre procesos y comunicación, tipos de fallas, tipos de fallas de comunicación, procesos determinísticos Vs. aleatorios.

11.9.2. Fault tolerance y tipos de fallas

Falla de procesos

Un proceso falló si su comportamiento se desvía del que se especificó en el algoritmo. En otro caso se dice que el proceso es correcto.

Un modelo de fallas especifica en que modo un proceso "defectuoso" puede desviarse de su algoritmo.

La siguiente es una lista de posibles modelos de falla:

- **Crash** - Un proceso se detiene prematuramente y no hace nada a partir de ese punto en que para, pero antes de esto se comporta correctamente.
- **Received omission** - Un proceso defectuoso se detiene prematuramente o intermitentemente omite la recepción de mensajes que le son enviados o ambos casos a la vez.
- **Send omission** - Un proceso defectuoso prematuramente o intermitentemente deja de emitir mensajes que supuestamente debería emitir.
- **General omission** - Un proceso defectuoso puede presentar un Send o Receive omission o ambos.
- **Arbitrary** (a veces llamado malicioso) Un proceso defectuoso puede exhibir cualquier otro comportamiento. Por ejemplo cambiar su estado en forma arbitraria.
- **Arbitrary with message authentication** - Un proceso defectuoso puede indicar haber recibido un mensaje de otro correcto cuando en realidad nunca lo hizo. Un mecanismo de

autenticación de mensajes permite al proceso correcto validar la indicación de recepción del proceso defectuoso.

Estos modelos de fallas pueden ser clasificados en términos de severidad. El modelo A es mas severo que el B si el conjunto de fallas permitido por B es un sub conjunto del permitido por A. Así un algoritmo que tolera fallas de tipo A también tolera las de tipo B.

Las fallas que se ajustan al modelo Arbitrary son las mas "severas" ya que no ponen ninguna restricción en el comportamiento de un proceso defectuoso. El modo Crash es el menos "severo" de todos los listados.

Los modelos de fallas son aplicables tanto a sistemas sincrónicos como a sistemas asincrónicos.

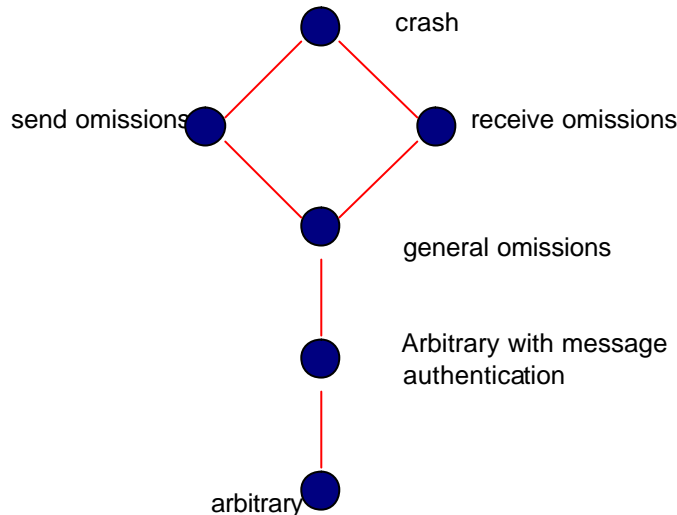


Fig 11.30 Clasificación de severidad de Fallas

FALLAS DE COMUNICACIÓN

Los Links de comunicación pueden verse afectados por los siguientes tipos de fallas:

- **Crash** - Un link defectuoso para de transportar mensajes. Antes de esto funcionaba en forma correcta.
- **Omission** - Un Link defectuoso omite intermitentemente el transporte de mensajes enviados a través de el.
- **Arbitrary** - Un Link defectuoso puede exhibir cualquier tipo de comportamiento, Por ejemplo generar mensajes falsos.

Si se trata de sistemas sincrónicos podemos encontrar también

- **Timing Failure** - Un Link defectuoso transporta mensajes mas rápido o mas lento de lo especificado.

TOPOLOGÍA DE RED

La red puede modelarse como un grafo donde los *nodos* son *procesos* y las *líneas* los *links* de comunicación entre procesos. Este modelo encierra lo mejor posible a las **Broadcast Network** (Redes con Emisión de mensajes). El problema que consideramos no es solucionable si las fallas resultan de una división de la red. Así es fundamental que la red tenga suficiente conectividad como para permitir una correcta comunicación entre procesos, ya sea en forma directa o indirecta a pesar de fallas en procesos y en la comunicación.

11.9.3. El problema del broadcast specification (Especificación de difusión)

Consideremos un sistema donde los procesos se comunican vía Broadcast (Emisión de mensajes). Si ocurren fallas durante la emisión de mensajes, es concebible que un proceso solo distribuya un subconjunto del mensaje que fue emitido.

Semejantes inconsistencias pueden comprometer la integridad del sistema distribuido y por lo tanto los Broadcast poco confiables no son herramientas apropiadas para construir aplicaciones con Fault Tolerance.

A continuación daremos una serie de tipos de Broadcast, que solucionan el problema de la inconsistencia producida por fallas en la emisión y transmisión de mensajes.

Tipos de broadcast

Este primer tipo de Broadcast que consideramos garantiza tres propiedades.

- Todos los procesos correctos están de acuerdo con respecto al conjunto de mensajes que distribuyen.
- Todos los mensajes que provienen de procesos correctos son distribuidos.
- No se distribuyen los mensajes erróneos o falsos.

Los Broadcast de alta performance generalmente no tienen restricciones en cuanto al orden en el cual se distribuyen los mensajes. A veces ese orden es importante. A continuación definiremos una colección de Strongly Broadcast haciéndolo en las garantías que ellos proveen en el orden de distribución de mensajes.

Informalmente los **Fifo Broadcast** son *Reliable Broadcast* que garantizan que los mensajes son distribuidos en el orden en que fueron emitidos. EL **Causal Broadcast**, es otro tipo en el que se requiere que los mensajes se distribuyan de acuerdo a la "relación de precedencia causal", un concepto fundamental en sistemas distribuidos. Por ejemplo, si un proceso distribuye un mensaje m y después emite un m' (la emisión de m' depende de que previamente se haya distribuido m) entonces *Causal Broadcast* requiere procesar la distribución de m antes que la de m' .

De cualquier manera el *Causal Broadcast* permite a los procesos la distribución de mensajes no causalmente relacionados en ordenes diferentes. Esto es prevenido por el *Atomic Broadcast*, un sistema que requiere a los procesos distribuir todos los mensajes en el mismo orden.

El **Fifo Atomic Broadcast** combina los requerimientos de *Fifo Broadcast* y el de *Atomic Broadcast*. Por otro lado la causal *Atomic Broadcast* combina los requerimientos de la *Causal Broadcast* y la *Atomic Broadcast*.

Hasta acá en nuestras definiciones de los varios tipos de Broadcast asumimos que estamos tratando solo con fallas benignas. Esto no solo simplifica las definiciones sino que también hace posible reforzar propiedades que son importantes en la práctica. A continuación desarrollaremos en detalle cada tipo de Broadcast que hemos presentado.

Reliable broadcast

Informalmente el *Reliable Broadcast* requiere que todos los procesos correctos distribuyan el mismo conjunto de mensajes, (*Agreement*) y que este conjunto incluya todos los mensajes emitidos por procesos correctos (*validacion*) y no contenga mensajes fallidos (*integridad*).

Formalmente *Reliable Broadcast* está definido en términos de 2 primitivas: Broadcast (m) y deliver (m) donde m es un mensaje de un conjunto M de mensajes posibles. Cuando un proceso invoca a *Broadcast* (m) decimos que m es emitido. Similarmente cuando un proceso ejecuta *deliver* (m) decimos que m es transmitido.

Como cada proceso puede distribuir varios mensajes, es importante poder determinar la identidad del emisor del mensaje y distinguir diferentes mensajes emitidos por un emisor en particular. Por lo tanto asumimos que cada mensaje m incluye los siguiente campos:

- Identidad del emisor, como *sender* (m)
- Número de secuencia, como *seg#* (m)

Si *sender* (m) = p y *seg#* (m) = i entonces m es el i 'esimo mensaje enviado por p . Estos campos hacen a cada mensaje único.

Un *Reliable Broadcast* tiene las siguiente propiedades

- Validity: Si un proceso correcto emite un mensaje m entonces todos los procesos correctos eventualmente distribuyen a m .
- Agreement: Si un proceso correcto transmite un mensaje m entonces todos los procesos correctos transmiten a m .
- Integrity: Para cada mensaje m , cada proceso correcto distribuye a m , al menos una vez y solo si m había sido previamente emitido por *sender* (m).

Fifo broadcast

Por lo general cada mensaje que no tenga un contexto puede ser mal interpretado. Los mensajes no deben ser distribuidos por un proceso que no conoce su contexto.

En algunas aplicaciones el contexto de un mensaje m esta compuesto por las emisiones previas de el emisor de m .

Por ejemplo: En un sistema de reservación de pasajes, el contexto en un mensaje cancelar reservación consiste en el mensaje que previamente hizo la reservación. El mensaje de cancelación no debe ser entregado en un lugar donde antes no se entregó el mensaje de reservación. Así muchas aplicaciones requieren de la temática de el *Fifo Broadcast*, y estos son los que satisfacen lo siguiente:

Fifo order - si un proceso emite un mensaje m antes de emitir m' entonces ningún proceso correcto distribuye m' sin antes haber distribuido m .

Causal broadcast

El orden Fifo es adecuado cuando el contexto de un mensaje m consiste solo en los mensajes emitidos previamente a m . Pero un mensaje m puede también depender de los mensajes que el emisor de m transmitió antes de emitir a m . En ese caso el mensaje distribuido con un orden garantizado por *Fifo Broadcast* no es suficiente.

Por ejemplo en una red de noticias si usamos para distribuir los artículos *Fifo Broadcast* puede ocurrir lo siguiente:

El usuario A emite un artículo, el usuario B en un lugar diferente lo recibe y emite una respuesta que solo puede ser entendida por A . El usuario C capta la respuesta de B sin antes haber recibido el original de A y entonces mal interpreta la respuesta de B .

El *causal Broadcast* previene el problema generalizado la noción de la dependencia de un mensaje con respecto a otro y asegurado que ese mensaje no es transmitido hasta que todos los mensajes de los que depende fueron transmitidos.

Un causal Broadcast es aquel que satisface el siguiente requerimiento:

—Causal order si la emisión de un mensaje m causalmente precede a la de m' entonces ningún proceso correcto va a transmitir m' sin antes haberlo hecho con m .

Atomic broadcast

EL *Causal Broadcast* no impone ningún orden de transmisión de mensajes que no estén causalmente conexos. Por lo tanto los procesos correctos pueden distribuir causalmente mensajes disconexos en diferente orden y esto crea problemas en algunas aplicaciones.

Por Ejemplo: consideremos una base de datos duplicada con dos copias de una cuenta bancaria X , estas dos bases se encuentran en lugares distintos. Inicialmente X tiene un valor de \$ 100, un usuario deposita \$ 20 entonces se emite un mensaje m (agregar \$ 20 a X) a las dos copias de la base de datos. Al mismo tiempo en un lugar distinto, el banco emite un mensaje m' (Agregar 10% de intereses a X). Como estos dos mensajes no están causalmente conexos, el causal Broadcast permite a las dos copias de X recibir los mensajes en orden distinto, así tendremos dos copias de X con valores distintos creando una inconsistencia en la base de datos.

Para prevenir este problema el *Atomic Broadcast* requiere que todos los procesos correctos transmitan los mensajes en el mismo orden. Este orden total en la distribución de mensajes asegura que todos los procesos correctos tengan la misma "vista" del sistema.

Formalmente un *Atomic Broadcast* es un *Reliable Broadcast* que satisface el:

Total order Si procesos correctos p y f transmite m y m' entonces p transmite m antes que m' si y solo si f transmite m antes que m' .

Fifo atomic broadcast

El *Atomic Broadcast* no requiere que los mensajes sean transmitidos en orden FIFO. Por ejemplo un proceso sufre una falla en la emisión de un mensaje m y después transmite m' y un proceso correcto transmite entonces solo un m' . Por lo tanto el *Atomic Broadcast* carece de la robustez que el *Fifo Broadcast* genera.

Entonces definimos el *FIFO Atomic Broadcast* que satisface el *FIFO Order* y el *total order* siendo mas robusto que los dos métodos por separado.

Causal atomic broadcast

El *FIFO Atomic Broadcast* no requiere que la distribución de mensajes se haga mediante un causal order.

Definimos entonces a un *causal atomic Broadcast* como al que satisface las condiciones de *FIFO Atomic Broadcast* mas el *causal order*.

Este método es el mas robusto de todos y es la clave del mecanismo de *State Machine Approach to Fault-Tolerance*.

Timed broadcast

Muchas aplicaciones requieren que cuando se envía un mensaje a "todos" sea transmitido en un determinado tiempo. Esta propiedad se denomina Δ Timeliness (Δ = Delta).

Como es usual en un ambiente distribuido el tiempo transcurrido puede ser interpretado en dos maneras :

- Tiempo Real, visto por un observador externo.

- Tiempo Local visto como el reloj local de los procesos.

Esto nos da dos maneras diferentes de definir el Δ timeliness. Veamos la primera correspondiente al tiempo real:

- **(Real Time) Δ Timeliness:** Es una constante. Conocida Δ tal que si un mensaje m es emitido en un momento t los procesos no correctos van a transmitir el mensaje después del *real-time* $t+\Delta$

Por otro lado la definición del Δ timeliness en términos de tiempo local radica en la diferencia existente entre el tiempo de emisión local de m y el tiempo de distribución local de m

- **(Local Time) Δ timeliness** Hay una constante. Conocida Δ tal que un proceso no correcto p transmite un mensaje m después del local time $ts(m) + \Delta$ del reloj de p

Un Broadcast que satisface cualquiera de los dos tipos anteriores es llamado *Timed Broadcast*.

Este tipo de Broadcast no puede ser implementado en sistemas asincrónicos, en cambio en los sincrónicos puede ser implementado el *local time Δ timeliness*, pero no el *real time Δ timeliness*.

Uniform broadcast

Agreement, integrity, order y atimeliness son propiedades que definen restricciones que muchas veces permiten la distribución de mensajes aun en presencia de procesos fallidos.

Como hasta ahora tratamos con fallas benignas estas restricciones son deseables y logrables, Por ejemplo el agreement permite que un proceso defectuoso transmita un mensaje que nunca fue transmitido por un mensaje correcto.

Este comportamiento ante fallas es indeseable en algunas aplicaciones como los Atomic Commitment en bases de datos distribuidas. Esto puede ser evitado si esas fallas son benignas, para tal fin vamos a extender la propiedad de Agreement.

- **Uniform Agreement:** Si un proceso (con o sin fallas) transmite un mensaje m , entonces todos los procesos correctos transmiten m .
En forma similar la Integrity permite a un proceso defectuoso transmitir un mensaje mas de una vez, como antes extendiendo la propiedad de integrity podemos evitarlo
- **Uniform Integrity:** Para cada mensaje m , cada proceso (Correcto o no) transmite m una vez, y solo si algún proceso emitió m los demás procesos correctos transmiten m .
Extendemos ahora las propiedad del local timeliness
- **Uniform Local Timeliness** Δ es una constante conocida tal que ningún proceso p (correcto o no) transmite un mensaje m después del tiempo local $ts(m) + \Delta$ según el reloj p .

El concepto de *Real Time Δ Timeliness* es análogo al anterior, definiremos ahora el orden en:

- **Uniform FIFO Order:** Si un proceso emite un mensaje m antes de otro m' , entonces ningún proceso (correcto o no) transmite m' sin haber transmitido previamente m .
- **Uniform Causal Order:** Si la emisión de un mensaje m por algún motivo precede a la emisión de un m' , entonces ningún proceso (Correcto o no) transmite m' sin antes haber transmitido a m .
- **Uniform Total Order:** Si Cualesquiera dos proceso p y q (Correctos o no), transmiten mensajes m y m' , entonces p transmite m antes de m' si y solo si q transmite a m antes de m' .

Para cada tipo de Broadcast definimos una contraparte Uniform reemplazando el Agreement, Integrity, y Δ Timeliness por las citadas aquí. Este tipo de Broadcast se utiliza para solucionar el problema del non-Blocking Atomic Commitment.

INCONSISTENCIA Y CONTAMINACIÓN

Consideremos una aplicación donde los procesos se comunican vía Faut-Tolerance Broadcast y asumimos que solo se presentaran fallas benignas.

Supongamos que un proceso p falla, omitiendo transmitir un mensaje que emiten todos los demás procesos correctos. El estado de p puede ser ahora inconsistente con respecto al estado de los demás procesos correctos.

Supongamos además que p sigue ejecutándose y que entonces en base a su estado inconsistente p emite un mensaje m que es distribuido por todos los demás procesos correctos. Notemos que m esta "corrupto, es decir que refleja el estado erróneo de p , por lo tanto al distribuirse m cambia el estado de los demás procesos correctos que incorporan la inconsistencia de p en su propio estado.

Ahora todos los procesos correctos están "contaminados". Vemos como una falla benigna puede fácilmente llevar a la corrupción del sistema completo.

Desafortunadamente las especificaciones tradicionales de la mayoría de los Broadcast, permiten esta inconsistencia de procesos defectuosos y la subsecuente contaminación de los demás procesos sanos.

Por Ejemplo con Atomic Broadcast un proceso defectuoso puede alcanzar un estado de inconsistencia de muchas maneras, a saber:

- Omitiendo emitir un mensaje que todos los demás procesos emiten.
- Emitiendo un mensaje extra que ningún otro proceso emite.
- Emitiendo mensajes fuera de orden.

Con Uniform Broadcast la inconsistencia puede surgir solo por saltarse “mensajes que emiten el resto de los procesos”. En cualquiera de estos casos el sistema puede contaminarse.

Cabe aclarar que prevenir la inconsistencia de procesos defectuosos o al menos la contaminación de los correctos es deseable en muchas situaciones y afortunadamente puede hacerse con los métodos vistos anteriormente para el Broadcast.

AMPLIFICACIÓN DE FALLAS

Un Fault Tolerance Broadcast es usualmente implementado por un algoritmo de Broadcast que usa primitivas de comunicación de bajo nivel, como mensaje punto a punto, Send And Receive así la emisión o transmisión de un mensaje requiere la ejecución de una gran variedad de instrucciones y puede incluir varios Send and Receives.

El modelo de fallas comúnmente considerados en los libros esta definido en termino de fallas que ocurren en el nivel Send and Receive, como omisión de emitir o recibir mensajes.

Como afectan esas fallas la ejecución de las primitivas de alto nivel, como las Broadcast and deliveries?. En principio podemos asumir que si un proceso sufre un cierto tipo de falla en el nivel Sen and Receive, va a sufrir el mismo tipo de falla en el nivel Broadcast and deliveries.

Por ejemplo si un proceso defectuoso omite la recepción de mensajes va simplemente a omitir transmitir mensajes?. Desafortunadamente esto no es siempre así, en general los algoritmos de Broadcast “gustan” de amplificar las severidad de una falla que ocurre a bajo nivel.

Por ejemplo en el algoritmo de Atomic Broadcast la omisión en la recepción de mensajes causa una falla en el proceso de transmitir mensajes haciendo que este lo haga en orden erróneo.

CONSECUENCIA DE LA COMPLEJIDAD

Hay varias medidas de complejidad que pueden interesarnos cuando queremos medir la eficiencia de un Algoritmo la clave esta entre la cantidad de mensajes y tiempo requeridos por el algoritmo.

Cuando consideramos un algoritmo de Fault Tolerance podemos medir otra “cualidad”, la robustez o tolerancia de fallas del mismo, que puede ser medido como la “fracción” de falla que el algoritmo es capaz de soportar.

11.9.4. El problema del backup en Procesamiento Distribuido

Un camino que se puede usar para implementar los servicios de tolerancia a fallas es disponer de múltiples servidores independientes, es decir que la falla de uno no provoque que los demás también fallen. El estado de los servicios del sistema es duplicado y distribuido entre estos servidores y las actualizaciones son coordinadas en pareja de manera que cuando un subconjunto de servidores falla, el servicio igual permanece disponible.

Estos servicios de tolerancia a fallas son ser estructurados de alguna de las siguientes maneras:

- Un enfoque es la replica del estado de los servicios de todos los servidores y presentar el pedido del cliente a los servidores que no tengan falla. Este enfoque es usualmente llamado “La repetición activa o el enfoque del estado de las maquinas que mencionaremos anteriormente.
- El otro enfoque es designar a un servidor como el principal y al otro como el servidor de resguardo. Los clientes hacen pedidos mandando mensajes al servidor principal. Si el servidor principal falla, entonces el Servidor de resguardo asume la dirección. Este enfoque es usualmente llamado “Principal - Resguardo o el principal y el enfoque de copias (copy approach)” y es muy usado en sistemas comerciales con tolerancia a fallas.

Los dos enfoques tienen como objetivo que el cliente sienta que esta siendo atendido por un único servidor.

La diferencia entre estos dos enfoques está en como se manejan las fallas.

Con el enfoque del estado de los servidores el efecto de las fallas esta completamente enmascarado por la Votación (VOTING) y el servicio resultante es indistinguible desde un único servidor no defectuoso.

El enfoque de *primary-backup* tiene el inconveniente de que pueden perderse requerimientos o pedidos de los clientes, por lo que es necesario emplear protocolos adicionales para revertir la perdida de pedidos. Lo positivo de este enfoque es que involucra menos procesos redundantes, es menos costoso y por lo tanto prevalece mas en la práctica.

Analizaremos algunos fundamentos del costo usando el enfoque de *Primary-Backup*.

Claves del costo de algunos protocolos del enfoque Primary-Backup.

- **Grado de replicación:** El número de servidores que se usan para implementar el servicio.
- **Tiempo bloqueado:** El periodo entre el requerimiento de un pedido y la respuesta sin falla al mismo.
- **Tiempo de fallas:** El periodo en que los pedidos pueden perderse porque el servidor principal esta con fallas.

La cuestión fundamental es:

¿ Puede haber mas de F componentes que fallen con la mas chica cantidad de replicasiones (con la menor cantidad de servidores posible)?

La respuesta a esta pregunta está dada por los limites mas inferiores del grado de replicación, el tiempo bloqueado y el tiempo de falla del protocolo de *primary-backup*, donde el limite mas inferior define los costos necesarios en los que cualquier protocolo debe incurrir para resolver un problema. Sabiendo el limite mas bajo del problema tenemos las bases para evaluar las cualidades del protocolo.

Sin embargo, la existencia de un limite inferior no implica la existencia de un protocolo con estos costos. Cada problema tienen un limite inferior trivial. Nosotros deseamos bajar los limites, los limites mas inferiores que son logrados por algún protocolo. El costo de correr un protocolo define un limite superior para el problema que le protocolo resuelve. Si el limite inferior es igual que el limite superior, entonces el limite inferior es el justo y el protocolo es el óptimo (El limite inferior implica que un protocolo mas barato no puede existir).

Así dado un problema, debemos empeñarnos en identificar los limites mas inferiores que se ajusten mejor al problema y el protocolo mas óptimo.

Especificación del enfoque Primary-Backup

Informalmente el enfoque *Primary-Backup* es implementado con un conjunto de servidores, validando que no haya mas de un servidor como principal en un periodo de tiempo dado.

El cliente envía su pedido al servidor principal, si este cambia, es decir, si deja de ser el principal y pasa a tomar la dirección otro, el cliente aprende cual es el nuevo servidor que atenderá sus pedidos y manda sus pedidos futuros consecuentemente.

Es necesario que el protocolo del enfoque *Primary-Backup* satisfaga cuatro propiedades:

la primera propiedad es que no mas de un servidor sea el principal en un determinado tiempo.

- **Propiedad 1:** Prmy es un pedido local, en cualquier momento solo hay un servidor que satisfaga al Prmy.

Resumiendo, cuando decimos "Este es el *primary* para el tiempo T " significa que ese estado es satisfecho por el pedido Prmy en el tiempo T . Podemos definir ahora formalmente al tiempo de fallas como al periodo mas largo de tiempo en el que el servidor principal no esta disponible para atender algún pedido.

La propiedad 2 distingue el enfoque de *Primary-Backup* del enfoque de estado de las máquinas (*state-machine*). En el enfoque *state-machine* el cliente emite su pedido a todos los servidores, lo que tiene un considerable costo.

- **Propiedad 2:** Cada cliente(i) mantiene la identificación del servidor destinatario, de forma tal que el cliente le mande su pedido al servidor destino(i).

Asumimos que los pedidos mandados son encolados en una cola de mensajes.

La propiedad 3 dice que los pedidos llegados al servidor de resguardo son ignorados por éste.

- **Propiedad 3:** Si el pedido del cliente arribó al servidor que no es el principal, entonces el pedido no es encolado y por lo tanto no es procesado.

Puede parecer que la propiedad 1 y la propiedad 3 eliminan la necesidad de la propiedad 2. Pero este no es el caso. La propiedad 1 y la propiedad 3 aseguran que no mas de un servidor encole lo que el cliente pide. Eliminar la propiedad 2 permitiría a los otros protocolos actuar y lograr que esto puede ser infringido. Con estos protocolos el cliente puede mandar sus pedidos a un montón de servidores. Y si estos pedidos son enviados mientras la identificación del servidor principal es cambiada, entonces el pedido puede conseguir encolarse en mas de un servidor.

Algunos de nuestros números inferiores no se adaptan a tales protocolos.

Las propiedades 1 y 3 especifican el protocolo para la interacción entre el cliente y el servidor pero no las obligaciones del servicio. Por ejemplo las propiedades 1 y 3 no descartan un servidor principal que ignora todos los pedidos. La cuarta propiedad excluye esa trivial posibilidad.

- **La propiedad 4** implica que el protocolo del enfoque *Primary-Backup* puede ser usado solamente implementando el servicio comprometido a la tolerancia de un número grande de fracasos.

En la práctica nosotros podemos implementar un servicio que no este comprometido a tolerar un número determinado de fallas en un periodo dado, y que tampoco este obligado a reparar el servicio y reintegrarlo en un periodo de tiempo. La propiedad 4 implica custodia solo a lo largo de ese periodo, la reintegración es otro tema.

11.9.5. Compromiso atómico no bloqueado (non blocking atomic commitment)

Con sistemas de bases de datos distribuidos, un protocolo de *ATOMIC COMMITMENT* asegura que las transacciones terminen consistentemente, aun en presencia de fallas.

Un protocolo *ATOMIC COMMITMENT* se dice que es *NON-BLOCKING* si permite la terminación de transacciones de las participantes correctas a pesar de las fallas de otros.

Hay dos razones principales por las cuales se puede decidir estructurar un manejo del sistema en forma distribuida, en lugar de centralizada. La primera, los datos que van a ser manipulados pueden ser inherentemente distribuidos, como la base de datos de la cuenta bancaria de un cliente. Segundo porque posibilita la independencia entre fallas aumentando la disponibilidad de la información.

Cuando se realiza una actualización en un sistema distribuido, las fallas parciales pueden desembocar en una inconsistencia en la base de datos. Esta claro que la terminación de una actualización en un sistema distribuido tiene que estar coordinado entre todas las partes donde la información se encuentra almacenada, aun en la presencia de fallas. La coordinación necesaria es especificada por el *ATOMIC COMMITMENT PROBLEM*.

Entre las soluciones propuestas a este problema la mejor de ellas es el *PROTOCOLO TWO-PHASE COMMIT (2PC)*. Ahora, mientras que soluciona el problema del *ATOMIC COMMITMENT* puede resultar en el Deadlock de aplicaciones, donde un participante correcto es obligado a la terminación de una transacción debido a fallas ocurridas en la otra parte. Durante esos periodos de Deadlock puede ocurrir que la parte correcta de la transacción sea obligada a entregar recursos del sistema que fueron adquiridos para uso exclusivo de ella. Por lo tanto es deseable conseguir soluciones que no provoquen Deadlock al problema del *ATOMIC COMMITMENT* que permitan a la parte correcta de la transacción terminarla bajo muchos posibles escenarios de fallas.

Esto es por demás difícil de lograr y todos los protocolos que lo solucionan están tipificados por el *THREE-PHASE COMMITMENT (3PC)*.

Este tipo de protocolos sin Deadlock no son solo mas costosos (en tiempo) que sus contraparte que si producen Deadlock sino que además son mucho mas complejo de programar y entender.

Por ejemplo, para prevenir el Deadlock, las parte correctas necesitan comunicarse con la otra parte considerando una gran cantidad de estados posibles del sistema con el fin de proceder a una correcta finalización de la transacción.

Protocolos como el 3PC invocan subprotocolos para elegir el mejor método y determinar cual fue el último proceso en fallar y estos protocolos son en sí ya muy complejos.

El problema del compromiso atómico (The atomic commitment problem)

El problema del *ATOMIC COMMITMENT* compromete a una conclusión global del sistema antes de realizar una transacción, para evitar fallas. Cada participante de la transacción decide entre dos posibles opciones *COMMIT* o *ABORT*. Si decide *COMMIT* indica que todos los participantes van a realizar actualizaciones a la transacción en forma permanente. Si deciden *ABORT* ninguno lo va a hacer.

Las decisiones individuales son irreversibles una vez que fueron tomadas. Una decisión de *COMMIT* esta basada en un Sí unánime "votado" entre todos los participantes.

Formalizamos estas nociones como un conjunto de propiedades, que juntas definen el *ATOMIC COMMITMENT PROBLEM*

- 1: Todos los participantes alcanzan la misma decisión.
- 2: Si cualquier participante decide *COMMIT* entonces todos los demás deben votar por el sí.
- 3: Si todos los participantes votaron si y no ocurren fallas, entonces todos los participantes deciden *COMMIT*.
- 4: Cada participante decide a lo sumo una vez (esto es una decisión irreversible)

Un protocolo que cumple estas cuatro características es llamado un *ATOMIC COMMITMENT PROTOCOL*.

Non blocking atomic commitment problem

En un protocolo *ATOMOC COMMITMENT* existe un proceso coordinador que es el que "organiza" la transacción, viendo si todos los participantes votaron si, ect. Si un participante llega a un Time out esperando una decisión del coordinador, invoca a un protocolo de finalización.

Este protocolo va a tratar de contactar a algún otro participante que ya haya decidido o no. Si triunfa lo va a llevar a una decisión, de todas maneras vamos a estar frente a situaciones en las cuales un protocolo de terminación no va a poder llegar a una decisión.

Supongamos que usamos un determinado protocolo de *ATOMIC COMMITMENT* y ocurre una falla en la emisión del mensaje que contenía la decisión por parte del coordinador. Podría ocurrir esto:

- Todos los participantes defectuosos transmiten la decisión y después fallan y
- Todos los participantes correctos votaron previamente sí y no distribuyen la decisión.

Si un proceso defectuoso no se recupera, ningún protocolo de terminación puede conducir a los correctos a tomar una decisión.

Cualquier decisión puede contradecir a la decisión tomada por el participante que falló.

Decimos que un *ATOMIC COMMITMENT PROTOCOL* es *BLOCKING* si admite ejecuciones en las cuales un proceso correcto no puede decidir.

Como definimos antes un *BLOCKING ATOMOC COMMITMENT PROTOCOL* es poco deseable porque lleva a una pobre utilización de los recursos del sistema. Un protocolo se dice non-Blocking si además de las propiedades 1-4 cumple con la siguiente:

5. cada participante correcto que ejecuta un *ATOMIC COMMITMENT PROTOCOL* eventualmente decide.

11.9.6. Conclusión sobre problemas:

Como vimos hasta aquí, los sistemas operativos distribuidos presentan una serie de problemas y dificultades de diseño que no vemos en los sistemas operativos de tipo convencional. Por ello es muy importante conocer estos puntos débiles del modelo para poder construir aplicaciones que aprovechen al máximo el potencial de los sistemas distribuidos, que todavía no se han expandido a gran escala, pero no tardaran mucho en hacerlo.

Los sistemas de computo de la actualidad, excepto algunos muy especializados como los que se utilizan en el control de tráfico aéreo o centrales nucleares, no son tolerantes a fallos.

Cuando una computadora falla se espera que los usuarios acepten esto como algo inevitable. La población en general no acepta esto. Si un canal de televisión, o el sistema telefónico o la compañía de luz eléctrica fallan durante media hora, al otro día habrá gran número de usuarios disconformes o poco felices..

Con la difusión de los sistemas distribuidos crecerá la demanda de sistemas que esencialmente nunca fallan. Los sistemas actuales no pueden cumplir ese requisito.

Es claro que tales sistemas necesitarán de una considerable redundancia en el Hardware y la estructura de comunicación, pero también la necesitará el software y particularmente los datos. La replica de archivos que a menudo es una idea tardía en los sistemas distribuidos actuales será un requisito esencial en los sistemas futuros.

También se tendrán que diseñar los sistemas de modo que puedan funcionar cuando solo se disponga de una parte de los datos y aun en la presencia de fallas en el sistema tanto de hardware como de software.

Resúmen sobre problemas de File System Distribuidos

→ Varios usuarios COMPARTEN:

Archivos y Almacenamientos



Diseminación física por todo el SD.

(Recordar Local y Remoto)

Definición:

· Servicio de Archivos:

→ Es la especificación de los servicios que se le ofrece al cliente.

Interfase con File System.

• Primitivas		Cliente sabe con lo que cuenta.
• Parámetros		NO cómo está implementado.
• Acciones		

- Server de Archivos:

→ Proceso en alguna máquina que ayuda con la implantación (funcionamiento interno) del Servicio de Archivos.

— Cliente no debe saber:

- Número de Servers
- Ubicación
- Funcionamiento
- Ni siquiera que es un SD

— Se pueden disponer de varios Servers con diferentes servicios: DOS, UNIX, Mac...

En el DISEÑO DE DFS: (Punto de vista del Usuario)

→ Hay generalmente dos componentes

- Servicio de Archivos
- Servicio de Directorios.

Servicio de Archivos:

→ Mantiene lo que se vió en FILE SYSTEM Clásico:

Performance: tiempo que se tarda en satisfacer una serie de pedidos.

- Acceso a disco (Clásico)
- Pequeño CPU burst (Clásico)
- Entrega de pedido al Server a lo largo de la Red. (DFS)
- Devolver respuesta a lo largo de la Red (CPU burst para software de Red). (DFS)

→ En DFS hay dos modelos básicos:

- Upload / Download
- Remote Access

Problemática de los Nombres de Archivos:

→ El Nombre de un archivo: Mapeo entre objetos Lógicos y Físicos.

- Le abstrae al usuario Dónde y Cómo el archivo está almacenado.
- En DFS hay un nuevo ocultamiento: Dónde en la Red?

→ Dos virtudes que se buscan para un DFS:

- Transparencia de Ubicación: Nombre no debe indicar nada de la ubicación física del Archivo.
- Independencia de Ubicación: No se tendría que cambiar el nombre si cambia la ubicación física. (por ej: Réplica)

→ Esquemas para Nombres:

1) Esquema Simple:

— NOMBRE = Host + Nombre Local.

- Garantiza único nombre en la Red.
- Se usan las mismas operaciones en forma Local y Remota.
- La ubicación NO es ni transparente ni independiente.

2) SUN's NFS:

- Vincula Directorios Remotos con Locales.

3) Integración Total:

- Un único espacio de nombres que tenga la misma apariencia en todas las máquinas.

Semántica de los Archivos Compartidos:

→ Semántica de UNIX:

— Read devuelve lo que hizo el último Write.

— En DFS es fácil de implementar la Semántica de UNIX ,si:

- Hay un solo Server
- No hay CACHÉ en los Clientes.

— El Server procesa las instrucciones Read y Write secuencialmente.

- (Retrazo de la Red es un Problema menor).

→ Problema del Caché:

- 1^{er} Solución: Propagar todas las modificaciones que se dan en el CACHÉ al Server → Muy Ineficiente.
- 2^a Solución: Cambiar semántica: "Solo se puede ver un archivo cuando nadie más lo esté usando."

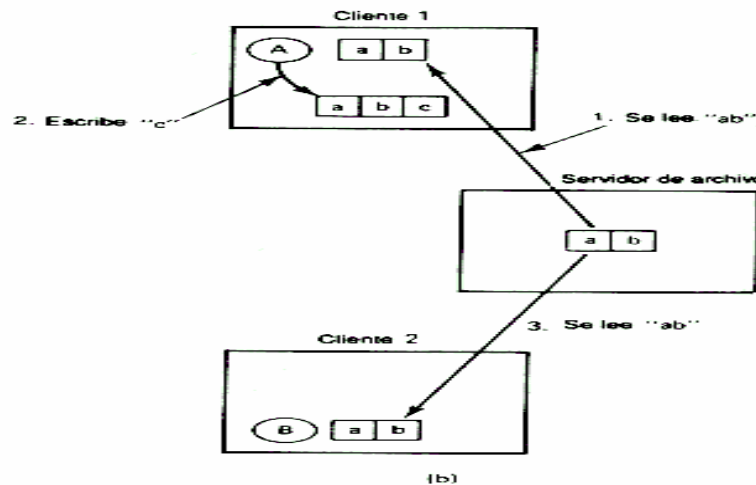


Fig. 11.31 El caché en procesamiento distribuidos

IMPLANTACIÓN de DFS:(Punto de Vista del Sistema \neq del Usuario)**Estructura del Sistema:**

→Cuál es el límite entre el cliente y el servidor?

NFS:

- No hay diferencia.
- Todas las máquinas con el mismo Software.
- Cualquier máquina puede brindar servicios de Archivos.

Otros sistemas:

- Por ejemplo: Peer to Peer.
- Se da elección al definir las máquinas:
 - Cliente
 - Servidor
 - Ambos

Otro extremo (de NFS):

- Cliente y Servidor en máquinas diferentes (Hardware y Software).
- Puede separarse en:
 - Servidor de Archivos.
 - Servidor de Directorios (entrega Dirección = Máquina + Inodo).
 - Por ejemplo:
 - Un Servidor de directorios DOS.
 - Un Servidor de directorios UNIX.
 - Un ÚNICO Servidor de Archivos.
 - Desventaja: más comunicación.
 - 1) Cliente envía nombre simbólico a Server de Dirección.
 - 2) Servidor de Dir. devuelve número en binario.
 - 3) Servidor de Archivos comprende dirección binaria.

→ Lo que distingue un DFS es la MULTITPLICIDAD y la AUTONOMÍA de y entre Clientes y Servidores.

11.10. CONCLUSIÓN

Los sistemas distribuidos son una buena opción en cuanto al costo ya que tienen en potencia una proporción precio-desempeño mucho mejor que la de un único sistema centralizado.

Cualquiera sea el objetivo, redimiendo normal a bajo costo o alto rendimiento con un mayor costo, los sistemas distribuidos son la mejor opción.

Otra ventaja potencia que ofrecen los sistemas distribuidos es que son muy confiables por sobre todo si lo comparamos con los sistemas centralizados. La carga de trabajo, en un sistema distribuido, es repartida entre muchas máquinas de manera que la falla de un chip descompondrá a lo sumo a una máquina y las demás podrán continuar su trabajo como si nada hubiera pasado.

Además los sistemas distribuidos ofrecen la posibilidad de crecimiento del sistema sin trastornos. Se pueden añadir instalaciones al sistema para ampliarlo de acuerdo a las necesidades.

En el siguiente cuadro podemos observar los puntos fuertes de los sistemas distribuidos.

Elemento	Descripción
Economía	Los microprocesadores ofrecen una mejor proporción precio /rendimiento que los mainframe.
Velocidad	Un sistema distribuido puede tener un mayor poder de computo que un mainframe.
Distribución inherente.	Algunas aplicaciones utilizan máquinas que están separadas una cierta distancia.
Confiabilidad	Si una máquina se descompone sobre-vive el sistema como un todo.
Crecimiento por incrementos.	Se puede añadir poder de computo en pequeños incrementos.

Tabla 11.09 Puntos sobresalientes de los sistemas distribuidos.

Es necesario destacar que unos de los motivos por lo que en (muchas ocasiones) se hace necesario la implementación de un sistema distribuido es por la posibilidad de compartir datos y recursos que estos ofrecen.

Otro punto fuerte de los sistemas distribuidos es que estos pueden albergar en el sistema a máquina de diferentes capacidades . Esto nos da la posibilidad de ejecutar los trabajos de la forma mas adecuada a sus características, encomendándoselos a las instalaciones mas aptas para resolverlo.

Sin embargo los sistemas distribuidos también presentan puntos negativos.

Su principal punto negativo son las redes de comunicaciones. Estas son lentas y pueden perder mensajes, lo cual requiere un software especial para su manejo y puede verse sobrecargado el sistema.

Además, si el sistema llega a depender de la red, la perdida o saturación de esta puede negar lagunas de las ventajas que un sistema distribuido debe conseguir.

Otro punto negativo es que, si bien la facilidad que otorga para compartir datos es una ventaja considerable este aspecto es un arma de doble filo. Las personas no deben tener acceso a todos los datos del sistema. Así surge el problema de la seguridad.

Pero todavía no nombramos el peor de los problemas, el software. En la actualidad no existe mucha experiencia en el diseño, implementación y uso del software distribuido.

En definitiva podemos concluir que el sistema distribuido ofrece muchas ventajas pero también implica mucho trabajo de diseño e implementación para que estas ventajas alcancen el esplendor que prometen.

11.11. Bibliografía recomendada para este Módulo

1. Distributed System, edited by Sape Mullander. (Second Edition),; Addison wesley, 1994, 601 pages.
2. Open System. Gary J. Nutt
3. Operating Systems. (Fourth Edition), Stallings Willams; Prentice Hall, Englewood Cliff, NJ. 2001, 779 pages.
4. Operating Systems Concepts (Fifth Edition), Silberschatz, A. and Galvin P. B; Addison Wesley 1998, 850 Pages.
5. Operating Systems Concepts and Design (Second Edition), Milenkovic, Millan; Mc Graw Hill 1992
6. Modern Operating Systems, Tanenbaum, Andrew S.; Prentice - Hall International 1992, 720 pag.
7. Sistemas Operativos Conceptos y diseños.(Segunda Edición); Milenkovic Milan; Mc Graw Hill; 1994. 827 páginas

GLOSARIO DE TÉRMINOS EN IDIOMA INGLÉS

File	client - server	peer to peer	mainframe
on-line	Local Area Network	Wide Area Network	Host
Off-line	Routers	Frames	Middleware
File System	Client -Server computing	Scheduling	Data
Kernel - Microkernel	Single Instruction Stream	Single Data Stream	Multiple Instruction Stream
First In First Out	Multiple Data Stream	Stack	Open System
Print Server	Thightly coupled	Lowley coupled	Uniform Memory Access
Not Uniform Memory	Not Remote Memory	Cross switch	Cross Point switch

Access	Access		
Crosstalk	Link	write	Network File System
Threads	Mailbox	Deadlock	Master -Slave
Overhead	Task	Context switch	Run time system
Garbage collection	Make	Daemon	Dispatcher
Universal	Time out	Fault Tolerance	Distributed Computing
Coordinated			
Message Passing	Shared Memory	Crash	Received omission
Send omission	General omission	Arbitrary with message authentication	Timing failure
		Broadcast especification	FIFO Broadcast
Broadcast Network	Strongly Broadcast	Atomic Broadcast	Agreement
Reliable Broadcast	Causal Broadcast	Validity	Integrity
Deliver	Sender	Total Order	State Machine approach to
FIFO Order	Causal Order	Uniform agreement	Non blocking Atomic commitment
Timed broadcast	Atomic commitment		Tree phase commitment
		Two phase commitment	Starter
Copy approach	Primary Backup	Image	service call
Others	Spooling	Acknowledge	Execute Procedure
Long term scheduler	Memory allocation	Pack / Unpack	File server
error cheking	Call / Return	Disk server	Lazy replication
client stub	Server stub	Remote access	Begin / End
caché	Upload /Download	Transaction	Layer
Pipeline	Workstation	Network	Recovery
Clock	Load Sharing	Check point	set
Session	File transfer	Bus /tree topology	native mail
Star topology	Ring topology	mailbox	
frame	checksum	Message Passing	
full	half		

American Standard Code for information interchange

GLOSARIO DE TÉRMINOS EN CASTELLANO

Tipos de procesamientos	Procesamiento Centralizado
Procesamiento Distribuido	Procesamiento Cooperativo
Procesamiento Cliente - Servidor	Procesamiento Par a Par
Objetivos del Procesamiento Distribuido	Fuertemente acoplado - Debilmente Acoplado
Ventajas y desventajas de los Sistemas Distribuidos	Eescalabilidad
Redes	Topología de Redes
Tipo de redes	Comunicaciones
Estrategias de ruteo de mensajes	Estrategias de conexión
Conflictos	Estrategias de diseño
El modelo OSI	Interoperatividad
Middleware	Plataformas
S.O. para Multiprocesadores	Software de comunicaciones (S.O. de redes)
S.O. realmente distribuidos	Migración de : Datos, Computos y Procesos
Características de un S.O. Distribuido	Consistencia de datos y caché
Multiprocesamiento con tiempo compartido	S.O. Distribuidos
procesos y procesadores en Sistemas Distribuidos	Los hilos (threads)
Paquetes de hilos	Modelos de Sistemas
Asignación de Procesadores /algoritmos	Sincronización en Sistemas Distribuidos/Algoritmos
La Mutua Exclusión en Sistemas Distribuidos/Algoritmos	Las transacciones en Sistemas Distribuidos/Algoritmos
Los Deadlock en Sistemas Distribuidos	Los problemas en Sistemas Distribuidos
Las comunicaciones en Sistemas Distribuidos/Protocolos	Estructuras de redes
Protocolos de comunicaciones y arquitecturas	Las migraciones en Sistemas Distribuidos
Las comunicaciones en el procesamiento Distribuido	

ACRÓNIMOS USADOS EN ESTE MÓDULO

E/S	Entrada / Salida	I/O	Input / Output
DMA	Direct Memory Access	O.K.	okey

PCB	Process Contol Block	PID	Process IDentifier
IEEE	Institute of Electronic and Electric Engineers	SYSCALL	System Call
RAM	Random Access Memory	LAN	Local Area Network
ROM	Read Only Memory	WAN	Wide Area Network
CPU	Central Processing Unit	ethernet	ether network
S.O.	Sistema Operativo	internet	inter network
ISO	International Standard Organization	CP	Communication Processor
OSI	Open System Interconnection	SQL	Structured Query Language
r/w/x	Read / Write/Execute	RPC	Remote Procedure Call
VMS	Virtual Machine System	B.D.	Base de Datos
SISD	Single Instruction stream, Single Data Stream	SIMD	Single Instruction stream, Multiple Data Stream
MISD	Multiple Instruction stream, Single Data Stream	MIMD	Multiple Instruction stream, Multiple Data Stream
PRG	PRoGram	SW	SoftWare
API	Application Programming Interface	HW	HardWare
UMA	Uniform Memory Access	NUMA	Not Uniform Memory Access
NORMA	NOt Remote Memory Access	NFS	Network File System
TCB	Thread Block Control	MUTEX	Mutua Exclusión
SCSI	Small Computer System Interface	RAID	Redundant Array of Inexpensive Disk
rsh	remote shell	PD	Procesamiento Distribuido
USR	User	RRP	Request/Ready Protocol
REQ	Request	TCP	Transport Control Protocol
REP	Reply	IP	Internet Protocol
ACK	Ack	FTP	File Transfer Protocol
AYA	Are you Alive?	SMTP	Simple Mail Transfer Protocol
IAA	I am alive	TELNET	Teletype Network
TA	Try Again	GW	Gate Way
AU	Address Unknown	PH	Phisical Header
FIFO	First In, First Out	SD	Sistemas Distribuidos
DFS	Distributed File System	NFS	Network File System
FS	File System	IPC	Inter Process Communication
SAP	Service Access Point	CC	Conmutador Central
PDU	Protocol Data Unit	EBCDIC	Extended Binary-Coded Decimal Interchange Code
ASCII	American Standard Code for information interchange		

AUTOEVALUACIÓN DEL MODULO 11:

Preguntas:

Múltiple Choice:

- 1.-
- 2.-
- 3.-
- 4.-
- 5.-
- 6.-
- 7.-
- 8.-
- 9.-
- 10.-
- 11.-
- 12.-
- 13.-
- 14.-
- 15.-
- 16.-
- 17.-
- 18.-
- 19.-
- 20.-

Respuestas a las preguntas

1. ¿

Respuestas del múltiple choice.