CAPITULO 23

SINCRONIZACIÓN EN SISTEMAS DISTRIBUIDOS

En sistemas con una única CPU las regiones críticas, la exclusión mutua y otros problemas de sincronización son resueltos generalmente utilizando métodos como semáforos y monitores.

Estos métodos no se ajustan para ser usados en sistemas distribuidos ya que invariablemente se basan en la existencia de una memoria compartida.

23.1. - SINCRONIZACIÓN DE RELOJES

No es posible reunir toda la información sobre el sistema en un punto y que luego algún proceso la examine y tome las decisiones.

En general los algoritmos distribuidos tienen las siguientes propiedades:

1)- la información relevante está repartida entre muchas máquinas
2)- los procesos toman decisiones basadas solamente en la información local disponible
3) - debería poder evitarse un solo punto que falle en el sistema
4)- no existe un reloj común u otro tiempo global exacto

Si para asignar un recurso de E/S un proceso debe recolectar información de todos los procesos para tomar la decisión esto implica una gran carga para este proceso y su caída afectaría en mucho al sistema.

Idealmente, un sistema distribuido debería ser más confiable que las máquinas individuales.

Alcanzar la sincronización sin la centralización requiere hacer cosas en forma distinta a los sistemas operativos tradicionales.

23.1.1 - Introducción a Relojes lógicos

El timer de un computador es generalmente una máquina precisa de cristal de cuarzo.

Asociado a cada cristal existen 2 registros : Un contador y un "holding register".

El contador se decrementa con cada oscilación del cristal y cuando llega a cero produce una interrupción y se lo carga de nuevo con el valor del registro holding.

Cada interrupción es llamada un golpe de reloj (clock tick).

Con este reloj se actualiza cada segundo el reloj que está en memoria (reloj por software).

En distintas CPUs con el transcurso del tiempo los relojes de software comienzan a diferir, esto se llama clock skew (oblicuo).

Lamport en 1978 presentó un algoritmo para sincronizar relojes.

Él apuntó que la sincronización no tiene por que ser absoluta. Si dos procesos no interactúan entre sí, no es necesario que sus relojes estén sincronizados, también indicó que no es importante que los tiempos coincidan sino que concuerden en cuanto al orden en el cual ocurren los eventos.

Por convención se denominan relojes lógicos a este tipo de relojes.

Si se agrega la restricción de que además los relojes no se pueden apartar del tiempo real en más menos un cierto valor se los denomina relojes físicos.

23.1.2. - **Relojes lógicos**

Lamport define la relación "sucede antes" (happened before)

a---> b a sucede antes que b

Existen dos situaciones:

1 - si a y b son eventos en el mismo proceso y a ocurrió antes que b entonces a --> b es verdadero

2 - si a es el evento de envío de un mensaje de un proceso y b es el evento de recibir el mensaje por otro proceso, entonces a -- b es verdadero. Un mensaje no puede ser recibido antes de haber sido enviado o a la misma hora.

La relación "sucede antes" es transitiva, o sea que si :

a --> b y b --> c entonces a --> c

Pero la relación no es reflexiva pues no se da a --> a

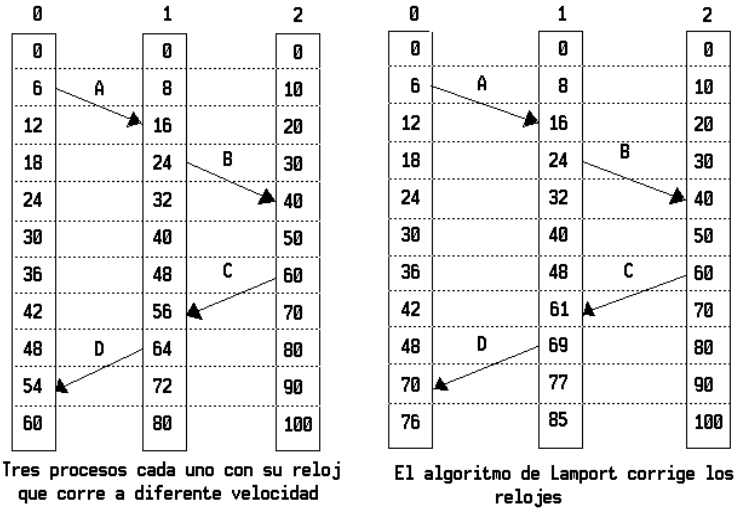


Fig. 23.1. - Algoritmo de Lamport.

Sea LC (logical clock) el valor del tiempo de un cierto evento entonces si

a -- >b entonces LC(a) < C(b)

Además el valor de LC va siempre aumentando, nunca disminuye.

Si no existe la relación --> entre 2 procesos diremos que esos dos procesos ejecutan concurrentemente.

En la Figura 23-1 se puede ver cómo los que tienen relojes más lentos los aceleran a los valores del más veloz.

Si dos eventos llegan en rápida sucesión a un proceso él debe avanzar su reloj por lo menos una vez por cada uno de ellos.

Es decir si a envía un mensaje a b y resulta que LC(a) > LC(b) se hace :

LC(b) = LC(a) + 1

Si además se agrega a la hora el número de proceso que envía el mensaje se pueden establecer las siguientes condiciones :

1 - si a sucede antes que b en el mismo proceso LC(a) < LC(b)

2 - si a y b son la emisión y recepción de un mensaje LC(a) < LC(b)

3 - para todos los eventos a y b LC(a) $\neq$ LC(b)

También pueden numerarse los nodos comenzando desde 1 con lo cual es equivalente decir LC(a) a LC_1.

23.1.3. - Relojes físicos

Si bien Lamport da un algoritmo no ambiguo para ordenar eventos los valores asignados a dichos eventos no son necesariamente cercanos a la hora actual en que ocurren.

En algunos sistemas (por ejemplo los de tiempo real) la hora actual es importante. Para estos sistemas se necesitan relojes físicos externos.

Por razones de eficiencia y redundancia son deseables múltiples relojes de este tipo lo que nos lleva a los siguientes problemas :

a) cómo los sincronizamos con los del mundo real ?

b) cómo los sincronizamos entre sí ?

La hora astronómica GMT (Greenwich Mean Time) ha sido reemplazada por la hora UTC (Universal Coordinated Time) basada en los relojes de átomos de Cesio 133.

Estos relojes son tan exactos que es necesario introducir “segundos de salto” para equipararla a la hora real. En realidad la hora basada en los relojes de cesio es la hora TAI (de Tiempo Atómico Internacional) y la hora TAI corregida con los segundos de salto es la hora UTC.

La mayoría de las compañías que proporcionan la luz eléctrica basan sus relojes de 60 Hz o 50 Hz en el UTC.

Esta hora UTC es provista al público por el NIST (National Institute of Stanford Time) por medio de una estación de radio ubicada en Fort Collins Colorado, cuyas letras son WWV que envía un corto pulso al inicio de cada segundo UTC. Su exactitud es de 10 msec. También algunos satélites proveen la hora UTC.

23.1.3. - ALGORITMOS DE SINCRONIZACIÓN DE RELOJES

Si una máquina recibe la hora WWV entonces la tarea es lograr que las otras se sincronicen con ésta.

Si la hora UTC es t, el valor en la máquina p es

$C_p(t)$

lo ideal sería que

$C_p(t) = t$

es decir que

dC / dt debería ser 1

Como existen ligeras variaciones entre los ticks de los relojes se puede decir que existe una constante δ tal que

$1 - \delta \leq C/dt \leq 1 + \delta$

esta constante δ viene especificada por el fabricante y se denomina "tasa máxima de desplazamiento" (maximum drifting rate)

Si se desea que dos sistemas operativos no difieran en más de α entonces deberán resincronizarse por lo menos cada α / 2 δ segundos.

Los distintos algoritmos difieren en cómo se resincronizan.

23.1.3.1. - **Algoritmo de Cristian** (1989)

Llamamos a la máquina WWV un servidor de tiempo

Periódicamente (no más allá de α / 2 δ segundos) cada máquina envía un mensaje al servidor preguntándole la hora.

El servidor responde inmediatamente con su C_{UTC}.

Lo que puede hacer el receptor es setear su hora a la que recibió.

Pero existen dos problemas, uno más grave que el otro.

El grave es que el reloj del receptor no puede retrocederse. Si su reloj interno es más rápido que el del servidor setearse con C_{UTC} le traería muchos problemas.

Una forma de hacerlo es realizar las correcciones al reloj software en forma gradual.

Si el timer genera 100 interrupciones por segundo cada interrupción agrega 10 msec a la hora. Para retrasarlo la rutina de la interrupción solo agregaría 9 msec hasta llegar a la hora correcta y para adelantarlo podría sumar 11 msec.

El otro problema menor es que existe un tiempo distinto de cero que le lleva a la hora del servidor el llegar al receptor, este tiempo además empeora si el tráfico de la red es alto.

Lo que hace el algoritmo es utilizar la hora de envío del requerimiento T_0 y la hora de recepción T_1 y hace una estimación (T_1 - T_0) / 2.

Se pueden hacer mejores estimaciones si se conoce la velocidad de propagación u otras propiedades.

Para mejorar la exactitud Cristian sugirió hacer no una sino varias mediciones. Aquellas mediciones en la cuales T_1 - T_0 supere un cierto umbral son descartadas por ser víctimas de la congestión de la red y por lo tanto poco confiables.

Las restantes pueden ser promediadas para obtener un mejor valor.

Alternativamente el valor que llega más rápido puede ser tomado como el más exacto.

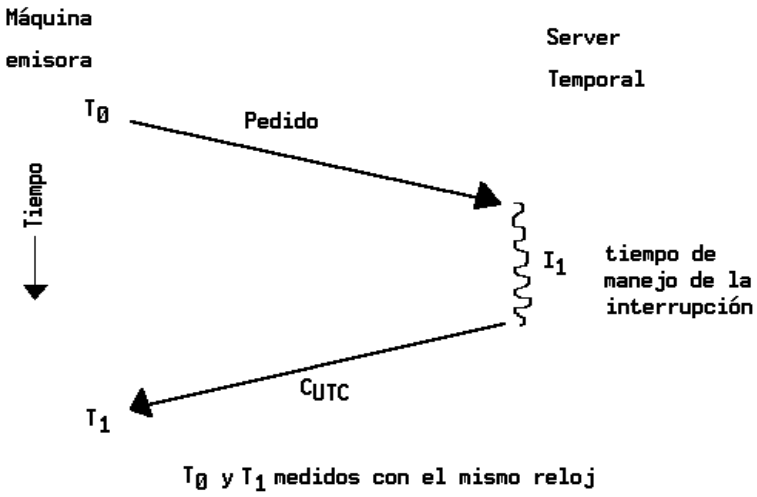


Fig. 23.2. - Obteniendo la hora actual del servidor.

23.1.3.2. - **El algoritmo Berkeley**

En el algoritmo de Cristian el servidor es pasivo. En el UNIX Berkeley ocurre exactamente lo opuesto (Gusella y Zatti 1989).

Aquí el servidor de tiempo (en realidad un demonio temporal) es activo y polea cada máquina periódicamente para preguntarles que hora tienen.

Promedia lo obtenido y de acuerdo a ello les dice que adelanten o atrasen sus relojes hasta alcanzar un cierto valor.

Este método es apto para sistemas en los que no existe ninguna máquina WWV.

La hora del demonio temporal debe ser seteada por el operador periódicamente.

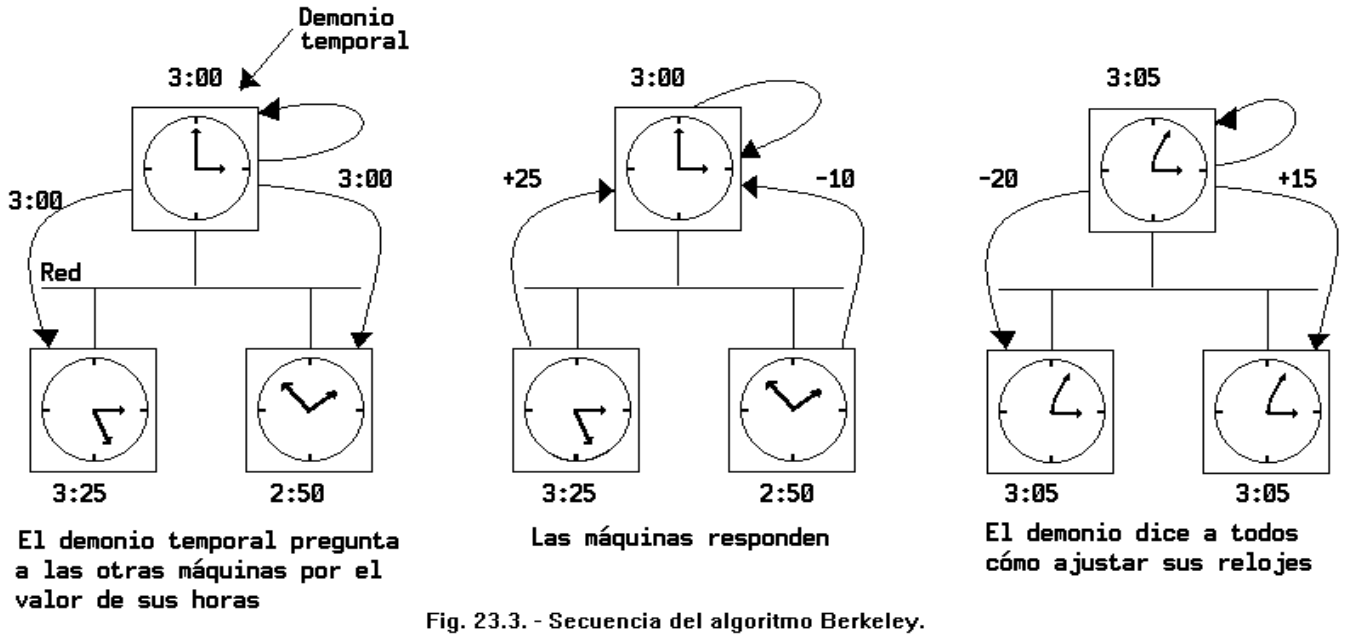


Fig. 23.3. - Secuencia del algoritmo Berkeley.

23.1.3.3. - **Algoritmos de Promedios**

Los dos métodos anteriores son centralizados con los problemas que ya conocemos.

Un algoritmo descentralizado trabaja dividiendo la hora en intervalos fijos de resincronización.
El i-ésimo intervalo empieza en el momento

$T_0 + i\ R$

y corre hasta

$T_0 + (i + 1)\ R$

donde

T_0 es un momento de acuerdo en el pasado
R es un parámetro del sistema

Al comienzo de cada intervalo cada máquina hace broadcast de su hora. Como los relojes locales difieren este broadcast no ocurre simultáneamente.

Luego de que cada máquina envía su hora empieza a correr un timer para recolectar todos los otros broadcast que lleguen durante un cierto intervalo S. Cuando todos llegan se ejecuta un algoritmo para calcular la nueva hora.

El algoritmo puede ser promediarlos a todos, que es el más simple,

También pueden descartarse los m mayores valores y los m menores y promediar el resto. Esto último sirve para liberarse de relojes que funcionen mal y pueden provocar horas sin sentido.

Otra forma es corregir cada hora que llega con un valor estimado de la propagación desde la fuente

23.1.3.4. - **Múltiples fuentes externas de horario**

Este es el caso de sistemas que necesitan exacta sincronización y cuentan con varias máquinas con hora WWV.

Cada máquina con hora UTC broadcastea su hora periódicamente, por ejemplo a cada comienzo de un minuto UTC.

Existen aquí demoras en cuanto a propagación por el medio, colisiones, congestiones de tráfico en la red, atención del receptor, etc.

Veremos por ejemplo cómo maneja esto la Open Software Foundation en su Distributed Computing Environment.

Cuando un procesador recibe todos los rangos UTC primero mira si alguno difiere de los otros. De ser así descarta a estos.

Luego hace la intersección de los valores restantes y luego calcula el punto medio y a ese valor coloca su reloj.

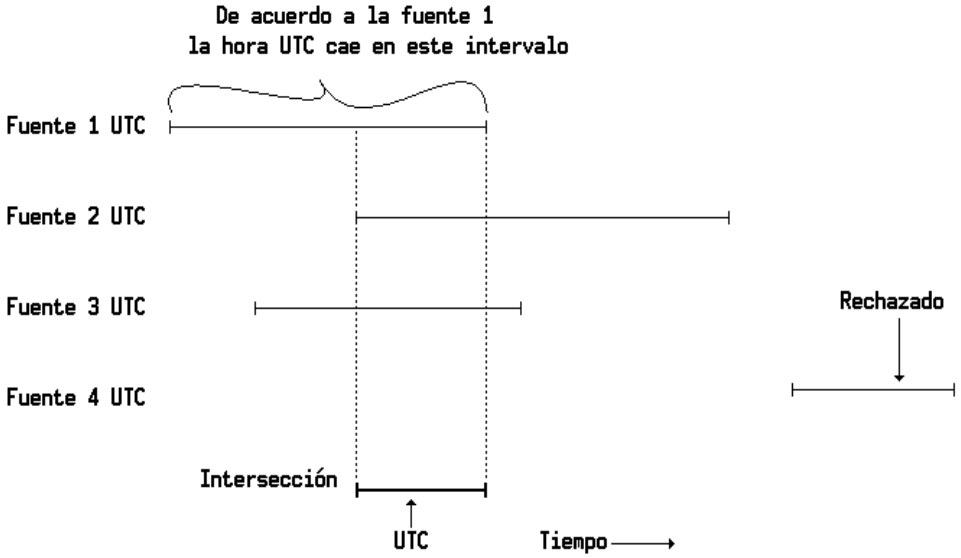


Fig. 23.4. - Calculando UTC desde múltiples fuentes externas.

Este método de sincronización es utilizado por DTS (Distributed Time Services) en DCE (Distributed Computing Environment).

DCE fue definido por la Open Software Foundation (OSF) que es un consorcio de vendedores de computadoras entre los cuales se encuentran IBM, DEC y Hewlett-Packard. No es un sistema operativo ni tampoco una aplicación, es un conjunto integrado de herramientas y servicios que pueden instalarse sobre un sistema operativo y utilizarse como plataforma para construir y ejecutar aplicaciones distribuidas. Ejemplos de sistemas operativos sobre los que puede correr DCE son AIX, SunOS, UNIX System V, HP-UX, Windows y OS/2.

En DCE los usuarios, máquinas y otros recursos se agrupan en **celdas**. En las celdas existe un servidor de tiempos que se sincroniza utilizando el algoritmo de múltiples fuentes externas. El servicio que ejecuta DCE para lograr esto se denomina DTS. Los servidores de tiempo se utilizan para sincronizar las maquinas de las celdas y también para sincronizar los servidores de tiempos entre celdas.

Una característica de DCE es que el tiempo en realidad es un intervalo de tiempo que contiene la hora exacta. Para lograr estas intersecciones de tiempo según el algoritmo se recomienda por ejemplo que cada LAN en una celda DCE tenga por lo menos 3 servidores de tiempo.

23.1.3.5. - **Otras técnicas**

Existe una forma de suavizar las diferencias de hora en forma local realizando promedios.

Supongamos un sistema distribuido en anillo o en grilla. Cada nodo puede arreglar su hora trabajando con promedios sobre las horas de sus vecinos.

23.2. - **EXCLUSIÓN MUTUA**

Para alcanzar la exclusión mutua usualmente se utilizan regiones críticas que suelen protegerse utilizando semáforos, monitores o construcciones similares.

Veremos cómo se logra la exclusión mutua y las regiones críticas en sistemas distribuidos.

23.2.1. - Un algoritmo centralizado

Se elige un proceso como coordinador. Cuando alguno desea entrar a una región crítica manda un mensaje al coordinador con su requerimiento y especificando a cuál región crítica quiere entrar.

Si nadie la está usando el coordinador responde con un mensaje de permiso.

Si alguien más desea esa región el coordinador puede no responder o responder denegando y encola el pedido.

Cuando se libera la región crítica el que la usó manda un mensaje al coordinador quien toma el primero encolado y le da el permiso.

Asegura la exclusión mutua.

Es justo ya que se atienden los pedidos en el orden que llegaron.

Nadie espera para siempre (no hay inanición, el manejo es FIFO).

El algoritmo solo requiere 3 mensajes: pedido, permiso, liberación.

Problemas:

- el coordinador es un punto de falla que haría caer al sistema
- si los procesos se bloquean luego de que les deniegan el permiso no pueden darse cuenta de si el coordinador se murió.
- el coordinador es un cuello de botella

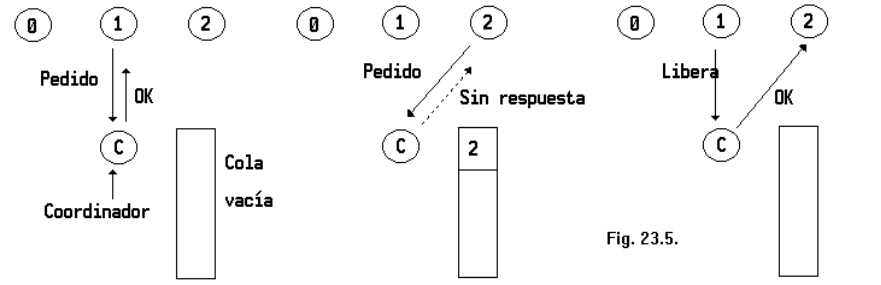


Fig. 23.5.

Proceso 1 pide al coordinador entrar a la zona crítica y se le otorga permiso

Proceso 2 pide luego entrar a la misma zona. El coordinador no responde

Cuando 1 termina y libera avisa al coordinador quien responde ahora al proc. 2

23.2.2. - Un algoritmo distribuido

Luego del trabajo de Lamport (1978), Ricart y Agrawala (1981) presentaron un trabajo más eficiente.

Su algoritmo requiere que todos los eventos en el sistema estén ordenados.

Cuando un proceso desea usar una zona crítica construye un mensaje formado por :

- la región que quiere usar
- su número de proceso
- su hora

Y lo envía a todos los procesos incluso él mismo.

Se supone que el envío de mensajes es confiable o sea que todos llegan.

Cuando le llega a un proceso, depende lo que hace según el estado en que se encuentre respecto de la región crítica solicitada :

1- si el receptor no está en esa zona crítica ni desea usarla le envía un mensaje de OK

2- si el receptor está usando esa zona pero todavía no responde y encola el pedido.

3- si el receptor desea usar la zona crítica pero todavía no entró compara la hora del mensaje con la hora de su propio mensaje enviado. La menor hora gana. Si el que llegó es menor el receptor le manda un OK, caso contrario no responde y encola el pedido.

El emisor una vez que todos le dieron permiso entra a la zona crítica y cuando la libera manda un OK a todos.

Si dos procesos envían un pedido en forma simultánea (iguales valores de LC) toma el recurso el que tenga el "orden" más bajo (ver fig. 23.6).

Este algoritmo asegura la exclusión mutua y no tiene inanición.

Se requieren 2 (n - 1) mensajes enviados siendo n la cantidad de procesos en el sistema.

No existe un único punto de falla.

Lamentablemente el único punto de falla fue reemplazado por n puntos de falla ya que si un proceso se cae no responderá a los requerimientos y parará al sistema.

Esto se arregla si se obliga al receptor a responder siempre ya sea permitiendo o denegando.

El emisor puede, al perder una respuesta, esperar un time out, repetirla y llegar a la conclusión de que un receptor está muerto.

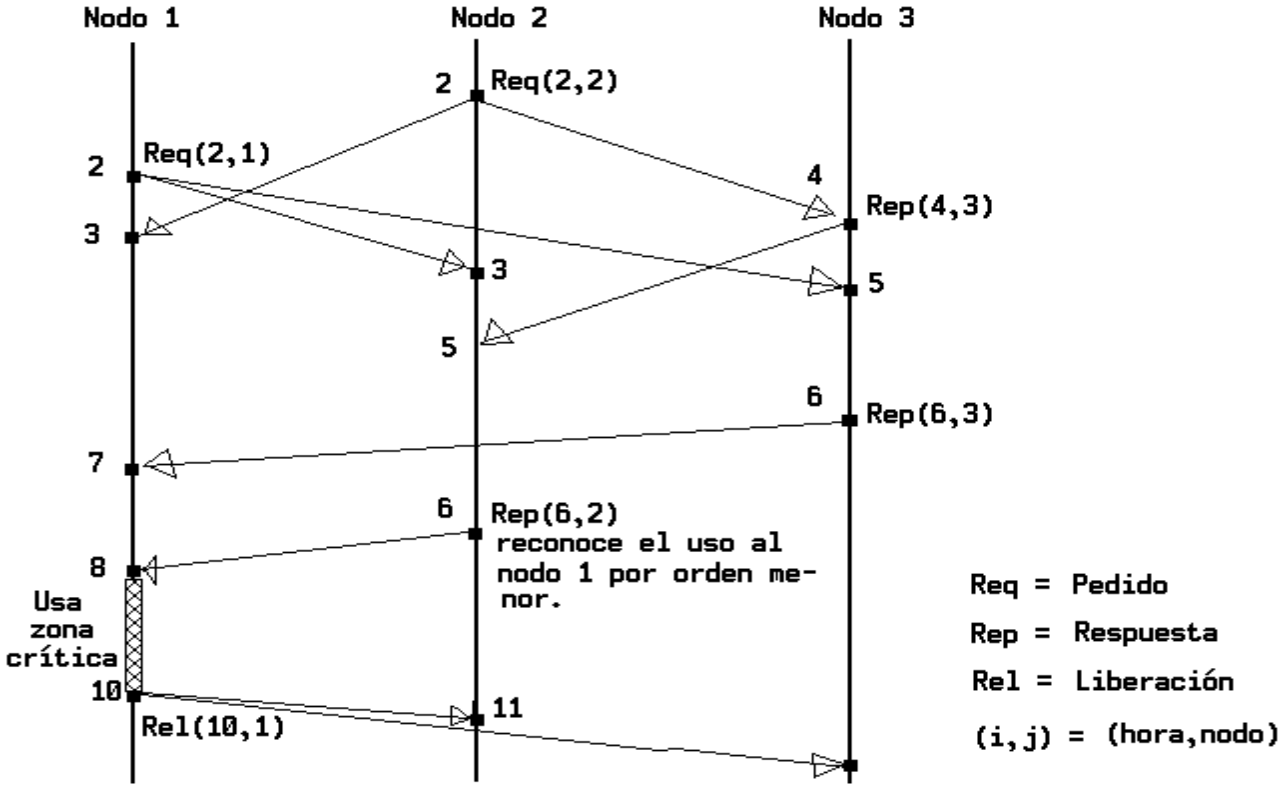


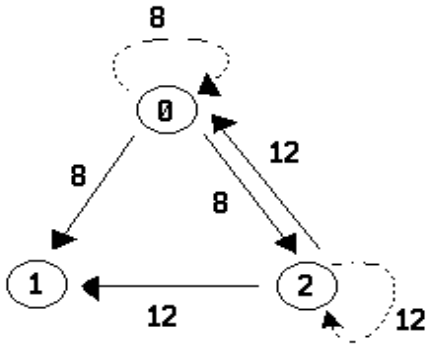
Fig. 23.6. - Sincronización distribuida.

Otro problema es que o se utilizan primitivas de comunicación de grupos o cada proceso debe llevar la lista de los miembros del grupo incluyendo los nuevos procesos o los que se van o se caen.

En oposición al cuello de botella del algoritmo centralizado, aquí <u>todos</u> los procesos intervienen en <u>todas</u> las decisiones que implican uso de zonas críticas.

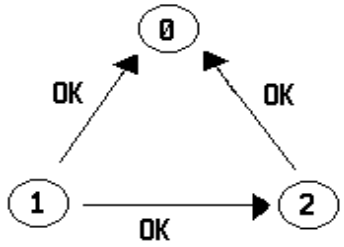
Se pueden hacer algunas mejoras, por ejemplo, que un proceso entre a zona crítica cuando recolectó permiso de la simple mayoría de procesos y no de todos. En este caso el que dio permiso a uno no puede darlo a otro hasta que el primero haya liberado.

23.2.3. - <u>Un algoritmo token ring</u>

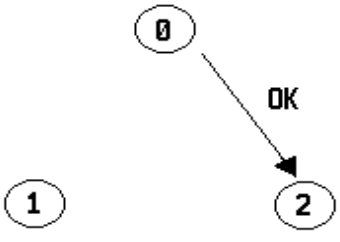


Dos procesos quieren la misma región crítica al mismo tiempo

Fig. 23.7.



El proceso 0 tiene la menor marca temporal y gana



Cuando 0 termina envía un OK entonces el 2 puede entrar a la zona crítica

Cada proceso tiene una posición asociada en el anillo y sabe cuál es su sucesor y su antecesor.

Pasa un token de uno a otro. Cuando un proceso adquiere el token entra a la región crítica, hace su trabajo, sale y restaura el token en el anillo.

Asegura exclusión mutua, no hay inanición.

Problemas :

*) pérdida del token, que deberá ser regenerado. La detección de esta pérdida no es fácil ya que si hace una hora que no aparece puede ser que lo esté usando alguien.
*) si un proceso se cae. Es más fácil de recuperar que en los casos anteriores ya que el vecino que trata de pasarle el token puede detectar que está muerto. Todo esto requiere que cada proceso mantenga la configuración actual del anillo.

23.2.4. - Comparación de los tres algoritmos

ALGORITMO	**Mensajes por entrada/salida**	**Demora antes de entrar (en cantidad de mensajes)**	**PROBLEMAS**
Centralizado	3	2	Caída del coordinador
Distribuido	2 (n - 1)	2 (n - 1)	Caída de cualquier proceso
Token Ring	1 a ∞	0 a n - 1	Pérdida del token, proceso caído

Es irónico que el algoritmo distribuido es aún más sensitivo a las caídas que el centralizado.

23.2.5. – Usos de relojes: Consistencia de cache

Es deseable en un sistema distribuido que los clientes puedan ocultar (caching) archivos en sus estaciones de trabajo. Esto provoca un problema de inconsistencia si dos clientes actualizan el archivo al mismo tiempo.

La solución usual es diferenciar si el archivo se oculta para lectura o para escritura, no obstante si un cliente desea escribir un archivo que ya fue obtenido anteriormente para lectura por otro cliente, el servidor debe invalidar la copia de lectura aún cuando sea una copia realizada hace varias horas.

Esto se puede solucionar utilzando relojes. Cuando el cliente obtiene un archivo se le otorga una especie de **renta** en la cual se especifica el tiempo de validez de la copia. Cuando la renta está a punto de expirar el cliente puede solicitar su revalidación. Si la renta expira la copia no puede utilizarse pero si el cliente aún desea usarla puede solicitar al servidor una nueva renta con lo cual el servidor (si el archivo es aún válido) genera la nueva renta y se la envía al cliente sin tener que retransmitir el archivo.

Si otro cliente desea utilizar un archivo para escritura el servidor debe notificar a los clientes que lo tienen en lectura y cuya renta no haya expirado, que la invaliden. Si un cliente está caído, o el link está caído a ese cliente, el servidor puede simplemente esperar hasta que la renta expire y otorgarle la escritura al que la requirió.

En el esquema tradicional sin relojes en donde el permiso debe devolverse explícitamente al servidor, el problema de la caída del cliente es complicado para el servidor ya que no sabe si el cliente se cayó o si está lento o si se cayó el link de conexión.

23.3. - **DETECCION DE FALLAS - ALGORITMOS DE ELECCIÓN**

En sistemas distribuidos es difícil diferenciar una falla de conexión, de procesador o pérdida de mensaje.

Si ambos nodos tienen una conexión física directa es posible determinar si la falla de conexión o caída de procesador ha ocurrido enviando un mensaje y no recibiendo respuesta y además alcanzando un "time-out".

Algunas de las fallas típicas que pueden producirse son :

1)- caída de un proceso en un nodo
2)- conexión física (si existe) caída
3)- caminos alternativos caídos
4)- mensaje perdido (regeneración de tokens en algunos protocolos)
5)- duplicidad de tokens
6) fracaso al transferir el token

Si se ha producido una falla y ésta fue detectada puede hacerse:

- buscar caminos alternativos
- si el nodo caído era un coordinador hay que buscar uno nuevo

La búsqueda de un nuevo coordinador requiere de algún algoritmo.

Este coordinador debe su existencia a la necesidad de que un proceso actúe como iniciador, secuenciador o para hacer algo en especial. No importa cuál pero alguien debe hacerlo.

Veremos algoritmos de elección de este coordinador. Si todos los procesos son iguales, sin características distintivas no hay forma de seleccionar uno en especial.

Asumimos que cada proceso tiene un único número, por ejemplo, dirección del nodo en la red, y que existe un único proceso por máquina.

Los algoritmos tratan de ubicar el proceso con mayor número y designarlo como coordinador. Difieren en cómo lo hacen.

Se asume que cada proceso conoce el número de los otros procesos, lo que no sabe es si están o no activos.

El objetivo es que cuando se inicia el algoritmo el mismo termina con un acuerdo generalizado sobre quién va a ser el nuevo coordinador.

23.3.1. - Algoritmo Bully (García - Molina 1982)

Cuando el coordinador no está respondiendo a los requerimientos, un proceso P hace :

1) P envía un mensaje de ELECCIÓN a cada proceso con número mayor a él
2) si nadie responde, P gana la elección y se convierte en el coordinador
3) si alguien responde éste toma el control, el trabajo de P termina

Un proceso puede recibir un mensaje de ELECCIÓN de los procesos de menor número que él, entonces le envía al emisor un OK para indicar que está vivo y se hará cargo.

A la larga solo queda uno que no recibe el OK y se autoelige coordinador y envía una señal de COORDINADOR a los otros nodos.

Si un coordinador caído se restaura envía un mensaje de COORDINADOR a los otros nodos obligándolos a someterse a él.

23.3.2. - Un algoritmo anillo

Este es otro algoritmo basado en un anillo pero a diferencia del anterior no existe un token.

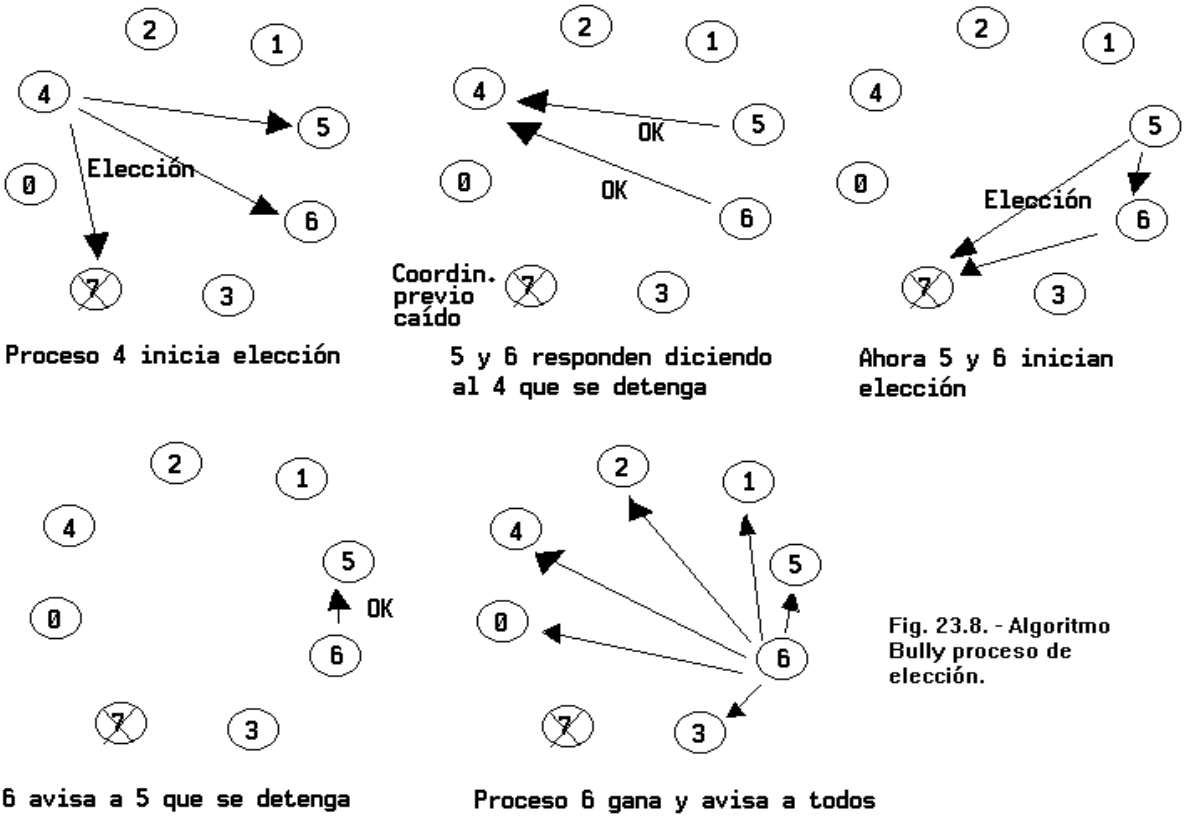
Cuando uno detecta que el coordinador ya no funciona construye un mensaje de ELECCIÓN con su número de proceso y enviándoselo a su sucesor, si el sucesor está caído el emisor lo saltea al próximo.

Por cada proceso que pasa éste le agrega su número al mensaje. Eventualmente vuelve al emisor original el que lo detecta por que ve su propio número en el mensaje.

Entonces cambia el mensaje por un mensaje de COORDINADOR y lo hace circular de nuevo para informar :

- quién es el nuevo coordinador (el proceso de mayor número)
- quienes son los miembros del nuevo anillo.

Después de la vuelta de este mensaje todos vuelven a su trabajo.



Proceso 4 inicia elección

5 y 6 responden diciendo al 4 que se detenga

Ahora 5 y 6 inician elección

6 avisa a 5 que se detenga

Proceso 6 gana y avisa a todos

Fig. 23.8. - Algoritmo Bully proceso de elección.

23.3.3. - Tokens duplicados

En este caso si un nodo que posee el token escucha en la línea que algún otro lo posee automáticamente se inhibe cediendo el derecho al otro. Si el otro nodo a su vez también se inhibe algún nodo regenerará nuevamente el token.

23.3.4. - Falla al transferir el token

Uno de los casos de falla de transferencia del token es cuando el nodo luego de transmitir escucha solamente su propia trama y luego silencio. Primero reintenta la transmisión y si la situación se repite supone en primera instancia que el receptor se ha caído y envía por lo tanto un mensaje pidiendo quién es el nodo siguiente del anillo. Si recibe respuesta del sucesor actualiza su información y transmite. Si por último no recibe respuesta envía un mensaje solicitando su sucesor en el anillo a lo cual todos los nodos están obligados a responder y el nodo emisor resuelve entonces cuál es el nodo al cual deberá transmitir. Si esto último falla entonces todos los nodos han dejado el anillo o falla la conexión o la recepción del nodo emisor falla en cualquiera de los casos el nodo emisor pasa a un estado inactivo y transmitirá solamente cuando vuelva a recibir un nuevo token.

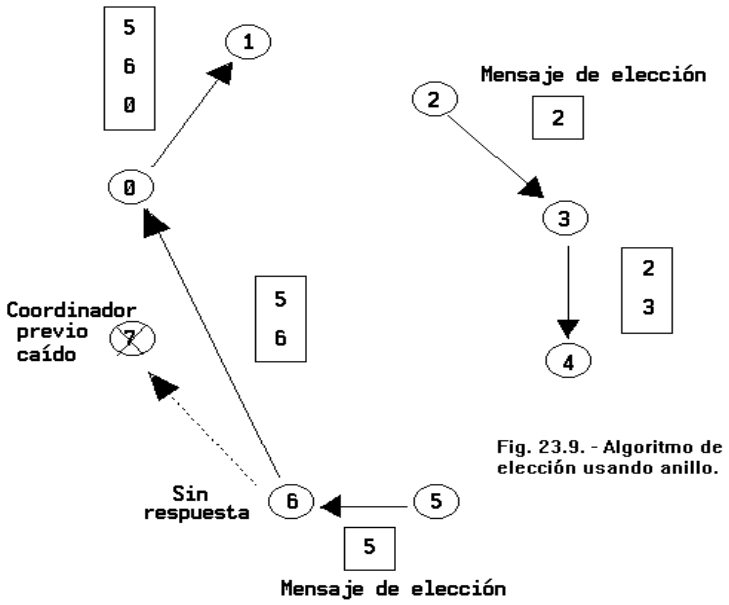


Fig. 23.9. - Algoritmo de elección usando anillo.

23.4. - **TRANSACCIONES ATÓMICAS**

Las técnicas de sincronización utilizadas son esencialmente de bajo nivel, como los semáforos.

Para esto hay que conocer detalles de la exclusión mutua, administración de regiones críticas, prevención de abrazo mortal y recuperación de caídas.

Deseamos un mayor nivel de abstracción el cual denominaremos "transacciones atómicas" o simplemente "transacciones"

23.4.1. - Introducción a transacciones atómicas

El modelo original proviene del mundo de los negocios en el cual para cerrar un trato es necesario una serie de acuerdos entre los negociantes. En computación sucede algo similar.

Un proceso anuncia que desea iniciar una transacción con uno o más otros procesos. Se pueden negociar varias opciones por ejemplo, crear y borrar objetos, y hacer algunas operaciones mientras tanto.

Luego el iniciador anuncia que desea que todos los otros completen su trabajo iniciado antes. Si todos ellos están de acuerdo, los resultados quedan en forma permanente. Si uno o más rehusa (o se cae antes de estar de acuerdo), la situación se revierte al estado exacto anterior al inicio de la transacción con todos los efectos laterales sobre objetos, archivos, bases de datos, etc., eliminados mágicamente.

Esta propiedad de todo-o-nada facilita la tarea del programador.

Este concepto proviene de los viejos sistemas de los años 60 en los cuales todavía no existían discos y lo que se hacía con cintas era por ejemplo lo que se ve en la figura 23-10.

Estos viejos sistemas tenían la propiedad de todo-o-nada de las transacciones atómicas.

Hoy en día por ejemplo una operación de transferencia de fondos de una cuenta bancaria a otra consta de 2 operaciones :

1 - extracción de cuenta A por x $

2 - depósito en cuenta B de x $

Si se corta la operatoria entre 1 y 2 los saldos no serían correctos a menos que se pudiera revertir la operación 1.

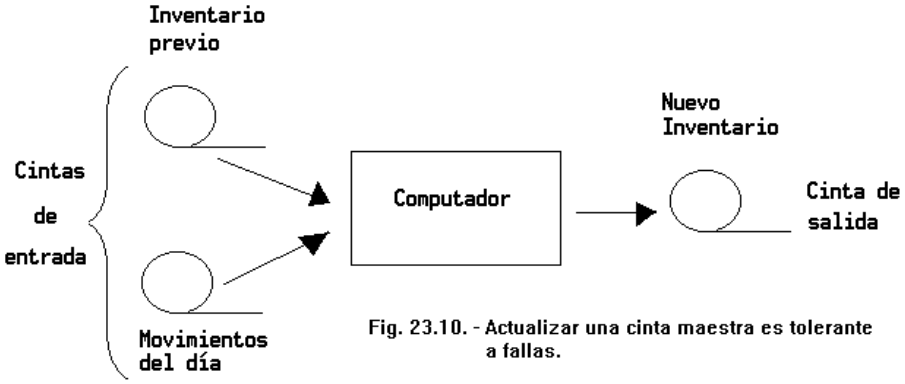


Fig. 23.10. - Actualizar una cinta maestra es tolerante a fallas.

23.4.2. - El modelo de transacción

Vamos a desarrollar un modelo de cómo es una transacción y qué propiedades posee. Asumimos lo siguiente :

- existen procesos independientes y cualquiera de ellos puede fallar
- la comunicación no es confiable en el sentido de que pueden perderse mensajes pero a niveles inferiores se usa time out y protocolo de retransmisión para recuperar estas fallas.
- asumimos que los errores de comunicación son manejados en forma transparente por el software subyacente

23.4.2.1. - **Almacenamiento estable**

La RAM pierde información cuando se corta la corriente. El disco sobrevive pero si aterrizan las cabezas, chau.

El almacenamiento estable está diseñado para sobrevivir a casi todo excepto catástrofes mayores como terremotos, etc.

Se implementa con un par de discos, uno de ellos es el reflejo exacto del contenido en el otro.

Sean los discos 1 y 2.

Primero se graba un bloque en el disco 1 y se lo verifica y luego se graba el bloque en el disco 2.

Si el sistema se cae antes de grabar el disco 2, se comparan ambos discos (al reiniciar actividad) y en las diferencias se da por bueno el disco 1 y se actualiza así el 2.

Si un bloque se daña por causas naturales (por ejemplo polvo) se lo puede regenerar a partir del mismo bloque en el otro disco.

Este tipo de almacenamiento lo hace sumamente útil para las transacciones atómicas ya que se baja a posibilidades muy pequeñas la probabilidad de errores.

23.4.2.2. - **Primitivas de Transacción**

Estas primitivas deberían estar provistas por el sistema operativo que trabaje con transacciones. He aquí algunos ejemplos básicos :

BEGIN_TRANSACTION : los comandos que siguen forman la transacción
END_TRANSACTION : fin de la transacción e intento de completar el trabajo (COMMIT)
ABORT_TRANSACTION : matar la transacción y restaurar al estado previo
READ : leer datos de un archivo u otro objeto
WRITE : grabar datos en un archivo u otro objeto.

La lista varía según la clase de objetos que se utilicen en una transacción. Por ejemplo si es un sistema de correo electrónico existirán primitivas para enviar (SEND) o recibir (RECEIVE) mensajes.

El BEGIN y el END se usan para delimitar el alcance de la transacción. Por ejemplo :

```
BEGIN_TRANSACTION
        reservar BUE - MIA
        reservar MIA - ORL
        reservar ORL _ DALLAS ----------> si este falla toda la transacción se aborta
END_TRANSACTION
```

23.4.2.3. - **Propiedades de las transacciones**

Tienen 4 propiedades esenciales

1)- SERIALICIDAD : las transacciones concurrentes no interfieren entre sí

2)- ATOMICIDAD : para el mundo exterior la transacción aparece como indivisible

3)- PERMANENCIA : una vez que una transacción hizo COMMIT los cambios son permanentes

4)- CONSISTENTES : la transacción mantiene los invariantes del sistema.

La primer propiedad asegura que si o más transacciones se ejecutan al mismo tiempo para cada una de ellas o para las otras el resultado final puede verse como si todas ellas se hubieran corrido secuencialmente en algún orden (ver tabla).

BEGIN_TRANSACTION	BEGIN_TRANSACTION	BEGIN_TRANSACTION
x = 0	x = 0	x = 0
x = x + 1	x = x + 2	x = x + 3
END_TRANSACTION	END_TRANSACTION	END_TRANSACTION

Itinerario 1	x = 0	x=x+1	x = 0	x=x+2	x = 0	x=x+3	Legal
Itinerario 2	x = 0	x = 0	x=x+1	x=x+2	x = 0	x=x+3	Legal
Itinerario 3	x = 0	x = 0	x=x+1	x = 0	x=x+2	x=x+3	Ilegal

Es parte del sistema el asegurar que las operaciones individuales son correctamente intercaladas.

La segunda propiedad es la que ya vimos del todo-o-nada. Cuando una transacción se está ejecutando el resto de los procesos del sistema sean o no transacciones no puede ver nada de los estados intermedios de la transacción.

La tercer propiedad se refiere al hecho de que una vez que se hizo el COMMIT no hay forma de volver las cosas atrás.

La cuarta propiedad puede verse claramente en los sistemas electrónicos de transferencias de fondos en los que se mantiene la conservación del dinero.

23.4.2.4. - **Transacciones anidadas**

Las transacciones pueden contener subtransacciones llamadas transacciones anidadas. La transacción madre puede hacer fork de hijos para que corran en paralelo y así siguiendo.

Existe un problema importante. Si existen subtransacciones en paralelo y una de ellas hace COMMIT y su padre aborta volviendo todo a su estado anterior esto iría en contra de la propiedad de permanencia. Sin embargo el significado es claro.

Cada subtransacción que se inicia obtiene su propia copia a su espacio de trabajo privado para hacer lo que desea. Si ella hace COMMIT su universo reemplaza al del padre. Si luego inicia otra subtransacción ve los resultados de la primera. Es decir que el COMMIT de la subtransacción solo afecta el universo del ancestro y no el mundo real.

23.4.3. - Implementación

Esta claro que si la transacción actualiza todo tipo de objetos esto no puede hacerse atómicamente y la restauración tampoco es simple. Veremos dos métodos de implementación.

23.4.3.1. - **Espacio de trabajo privado**

La transacción obtiene una copia de toda la información que necesita y trabaja sobre ella en un espacio propio privado.

Problema: costo del copiado, aunque se pueden hacer algunas mejoras. Por ejemplo en los casos en que solamente lee no se realiza copia de esta información en el espacio privado (amenos que haya cambiado desde el inicio de la TR).

Por ejemplo cuando empieza un proceso se le asigna un espacio privado vacío, cuando abre un archivo para leer el puntero lleva al archivo real. Si se trata de una subtransacción el puntero lleva al espacio del padre y de

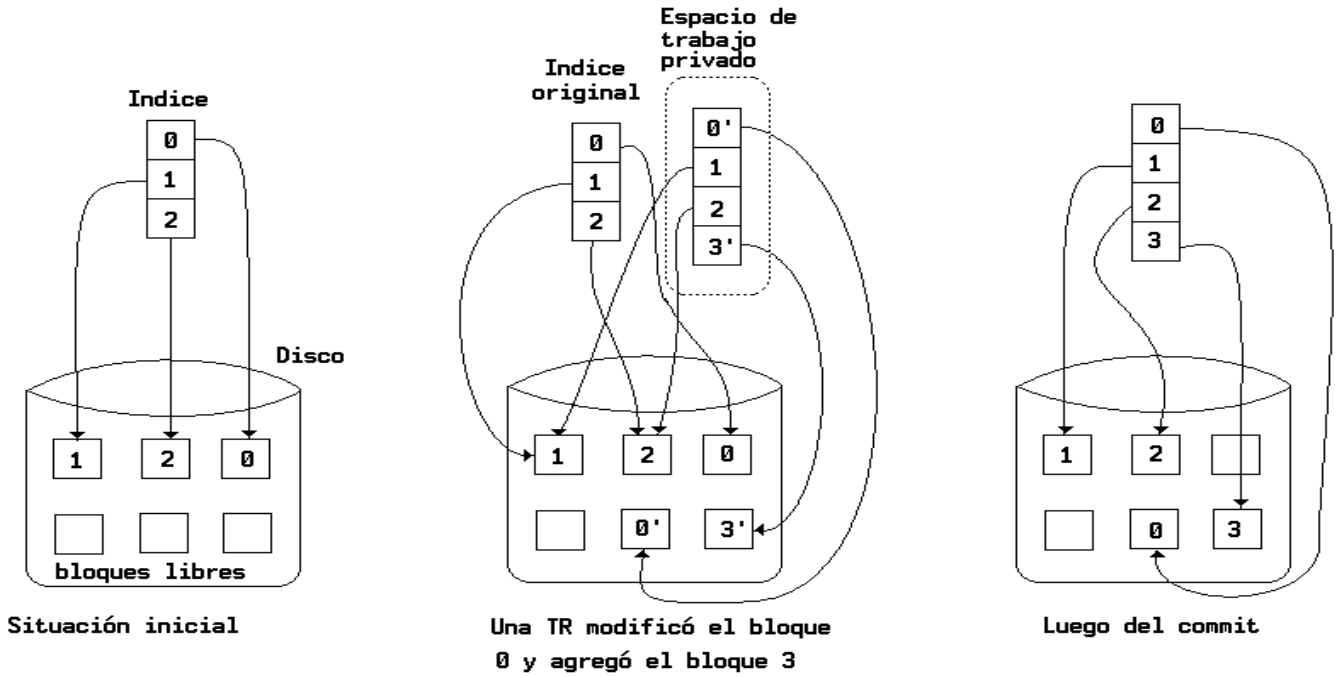


Fig. 23.11. - Actualización en espacio privado de trabajo.

allí al archivo real.

Cuando se abre un archivo para grabar los punteros desde las subtransacciones llevan al espacio privado de la TR madre y entonces allí sí se copia la información.

Una segunda mejora introduce que en lugar de copiarse el archivo entero solo se copia el archivo de índices del archivo.

En la figura 23.11 vemos por ejemplo como la TR ve solo lo que ella está modificando.

Cuando la TR aborta, su espacio privado se deletea y se devuelven los bloques libres a la lista.

Si se cumple, se reemplaza el índice en forma atómica y se actualiza la lista de espacios libres.

23.4.3.2. - **Log de Grabación Adelantada**

Es otro método también llamado Lista de Intenciones. Con este método el archivo es grabado en su lugar original pero antes de cambiar cualquier bloque se graba un registro en este log en almacenamiento estable que indica :

- qué TR está haciendo el cambio
- qué archivo y bloque se cambia
- cuál es el viejo valor
- cuál es el nuevo valor

x = 0 ; y = 0 ;			
BEGIN_TRANSACTION	Log	Log	Log
x = x + 1 ;	x = 0 / 1	x = 0 / 1	x = 0 / 1
y = y + 2		y = 0 / 2	y = 0 / 2
x = y * y			x = 1 / 4
END_TRANSACTION			
La transacción	Resultado en el log después de ejecutar cada sentencia		

Una vez que el log se grabó entonces se graba el archivo.

Si la TR hace COMMIT entonces se graba un registro en el log y no es necesario tocar los archivos porque ya están actualizados.

Si la TR aborta él los usa para retrotraer todo al estado original. Esta acción se llama ROLLBACK.

El log sirve también para recupero en casos de caídas ya que permite retroceder en la TR o verificar que lo último realizado por la TR efectivamente llegó a grabarse en el archivo.

23.4.3.3. - **Protocolo commit de dos fases**

En un sistema distribuido el COMMIT puede requerir de la cooperación de múltiples procesos en distintas máquinas.

Este protocolo fue ideado por Gray en 1978, si bien no es el único es el más usado en la actualidad.

Uno de los procesos involucrados funciona como coordinador, usualmente el que está ejecutando la TR.

El protocolo se inicia cuando el coordinador graba una entrada en el log diciendo que está empezando el protocolo COMMIT y enviando a los otros procesos un mensaje de que se preparen para el COMMIT.

Cuando un subordinado recibe el mensaje mira a ver si está listo para el COMMIT, hace una entrada en el log y devuelve su decisión al coordinador.

Cuando el coordinador recibió todas las respuestas sabe si puede o no hacer el COMMIT.

Si uno o más no pueden hacer COMMIT o no responden la TR se aborta.

El coordinador entonces graba su decisión en el log y se la informa a sus subordinados.

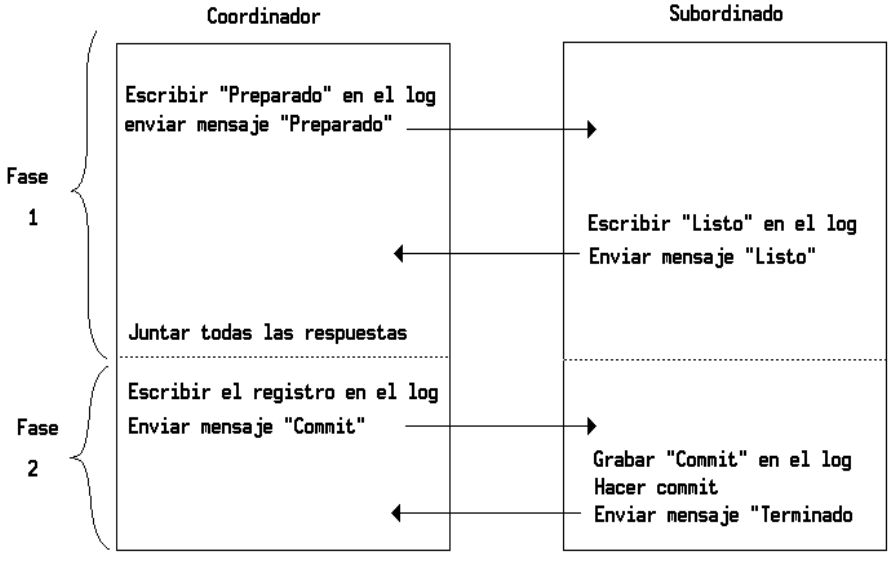


Fig. 23.12. - Protocolo Two-Phase Commit.

Esta grabación en el log es la que vale no importa lo que pase después.

Ya que se usa almacenamiento estable para este log este protocolo es muy flexible a múltiples caídas.

Si el coordinador se cae al mandar el primer mensaje cuando se recupera sigue donde estaba y vuelve a repetir el mensaje.

Si el coordinador se cae después de grabar el resultado de todas las respuestas cuando se recupera solo tiene que reinformar a sus subordinados.

Si el subordinado se cae antes de responder el primer mensaje el coordinador le enviará mensajes hasta que se recupere.

Si se cae después, cuando se recupere verá en el log que es lo que sucedió y actuará según ello.

23.4.4. - Control de concurrencia

Necesitamos mecanismos que controlen el acceso concurrente. Veremos tres algoritmos.

23.4.4.1. - **Bloqueo (locking)**

Cuando un proceso quiere usar un archivo lo bloquea. Puede existir utilizando un administrador centralizado de bloqueos o administradores locales de bloqueo para los archivos locales.

El bloqueo se adquiere y libera a través del sistema de transacciones, no se necesita ninguna acción del programador.

Pueden existir bloqueos de :

- lectura : permiten otras lecturas (compartidos)
- grabación : no permiten otras lecturas (excluyentes)

El bloqueo puede ser a más bajo nivel, por ejemplo un registro, o a mayor nivel, por ejemplo una base de datos.

Esta cuestión del grado del bloqueo se llama GRANULARIDAD DEL BLOQUEO.

Cuanto más bajo el grado, mayor paralelismo se puede lograr, pero se necesitan más indicadores de bloqueo, es más caro y puede llevar más a abrazos mortales.

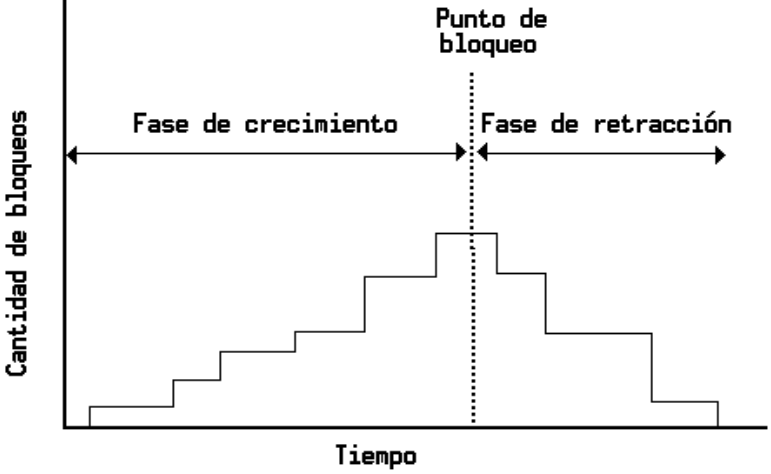
El adquirir el bloqueo solo cuando es necesario y liberarlo ni bien no se lo necesita puede llevar a inconsistencias y abrazo mortal.

Por otra parte muchas TR están implementadas con un uso de bloqueo que se llama BLOQUEO DE DOS ETAPAS.

En la primer etapa la TR adquiere todos los bloqueos (fase de crecimiento) y luego los libera durante la fase de tracción (segunda etapa).

Si el proceso no llega a actualizar algún archivo antes de llegar a la fase de retracción entonces se puede manejar la falla de adquirir algún bloqueo simplemente liberándolos a todos esperando un poco y empezando todo de nuevo.

Fig. 23.13. - Bloqueo de dos etapas.



Más aún, se prueba que si todas las transacciones utilizan este mecanismo de dos etapas todas las formas posibles de intercalar las transacciones son serializables.

La fase de retracción en muchos sistemas ocurre cuando la TR terminó bien (COMMIT) o mal (ABORT), esto se llama BLOQUEO DE DOS ETAPAS ESTRICTO, y tiene dos ventajas :

1) una TR siempre lee un dato generado por una TR que ya terminó con lo cual los cálculos propios no van a fallar por no poder leer un dato.

2) las adquisiciones y liberaciones pueden ser manejadas por el sistema sin que la TR se preocupe por ello. Esta estrategia hace desaparecer la eliminación en cascada (cascaded aborts) que se produce cuando hay que deshacer una TR completada (COMMIT) debido a que vio un archivo que no debería haber visto.

Puede haber abrazo mortal si dos TR piden recursos en orden opuesto. Se usan técnicas clásicas para evitar esto, como ser que se pidan los bloqueos en un cierto orden canónico preestablecido (evita hold-&-wait).

También se pueden detectar ciclos teniendo la información de qué procesos tienen cuáles bloqueos y cuáles desean adquirir.

También puede existir un timeout sobre el tiempo que un proceso puede mantener un cierto bloqueo.

23.4.4.2. - **Control de concurrencia optimista** (Kung y Robinson 1981)

La idea es hacer lo que se desea y no poner atención a lo que los otros hacen. Si existe algún problema ocúpese de él después.

Lo que hace este esquema es llevar un registro de cuáles archivos han sido leídos y escritos y cuando una TR pide COMMIT revisa si sus archivos fueron cambiados por algún otro, de ser así aborta la TR sino la cumple.

Esta técnica anda bien en las implementaciones basadas en espacios privados de trabajo donde cada TR altera sus archivos en forma privada.

Ventajas: está libre de deadlock y permite máximo paralelismo ya que nadie espera para realizar un bloqueo.

Desventaja: es que al fallar la TR debe reejecutarse y si existe mucha carga la probabilidad de fallas crece haciendo que este esquema resulte una pobre elección.

23.4.4.3. - **Sellos Temporales** (timestamps Reed 1983)

Se asigna a cada TR un sello de tiempo en el momento en que se realiza el BEGIN_TRANSACTION.

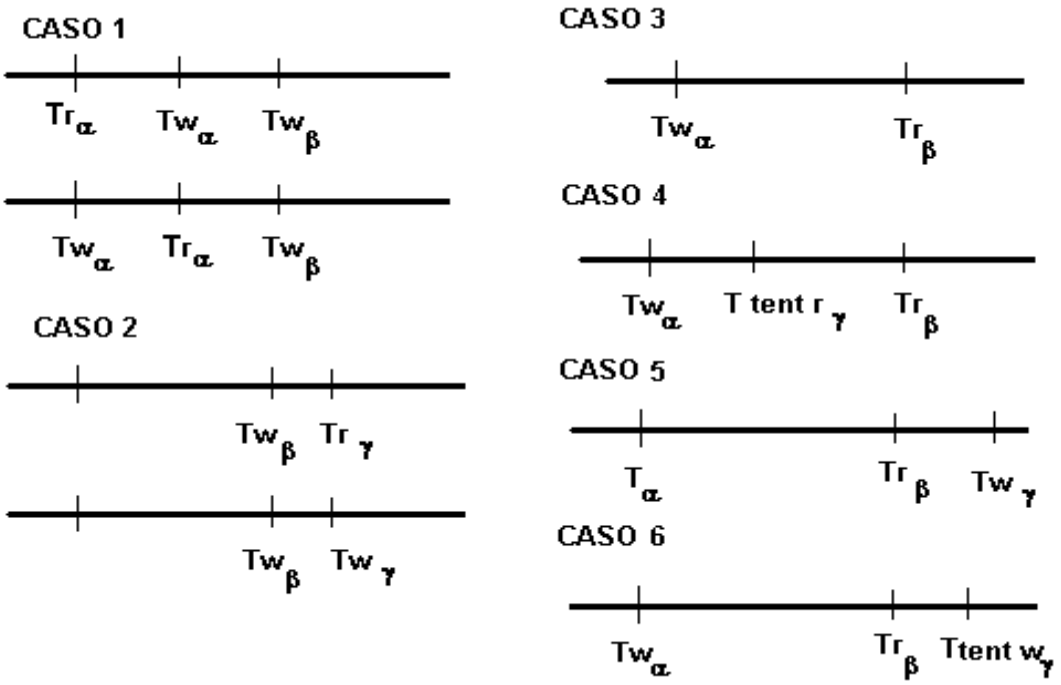
Se utiliza el algoritmo de Lamport para asegurar que esta hora es única.

Cada archivo tiene asociado un sello de lectura y uno de escritura que indican qué TR cumplida (que hizo COMMIT) es la última que leyó o actualizó.

Las propiedades de los sellos temporales respecto del bloqueo son que cuando la TR encuentra un sello mayor aborta en tanto que en el bloqueo puede esperar o continuar. Los sellos temporales están libres de deadlock.

Esta técnica asegura un ordenamiento correcto en el tiempo. Cuando el orden es incorrecto significa que una TR que empezó después que la actual manejó el archivo e hizo COMMIT con lo cual la TR actual es tardía y debe ser abortada.

Veamos algunos ejemplos, para ello asumamos tres transacción: α, β y γ, de las cuales α es la transacción vieja que accedió al recurso y ya hizo COMMIT. Las transacciones α y β se inician concurrentemente y el sello temporal de β es < que el sello temporal de γ.



Veamos primero el caso de la grabación.

En los casos 1) se acepta porque Twβ > Twα y se registra Twβ como tentativa de escritura hasta que la transacción realice commit.

En el caso 2) si la transacción γ lee o graba el archivo luego de que la transacción β lo accedió con intención de grabar y la transacción γ hace commit, entonces la transacción β debe ser cancelada.

Veamos ahora ejemplos respecto de la lectura.

En el caso 3) no hay conflicto por lo tanto la lectura procede.

En el caso 4) como la transacción γ intenta leer en una marca temporal menor que el intento de lectura de β, entonces β debe esperar a que γ haga commit y luego leerá.

En el caso 5) la transacción γ modificó el archivo e hizo commit, por lo tanto β debe ser cancelada.

En el caso 6) la transacción γ modifica tentativamente el archivo y si bien aún no hizo commit, β está atrasada y también debe ser cancelada.

La técnica de sellos temporales está libre de Deadlock aunque su implementación es bastante compleja.

23.5. - **ABRAZO MORTAL EN SISTEMAS DISTRIBUIDOS**

El abrazo mortal en sistemas distribuidos es peor, más difícil de evitar, prevenir e incluso de detectar.

Es también difícil de arreglar por que la información está desparramada entre distintas máquinas.

Alguno autores distinguen entre dos tipos de abrazos mortales en sistemas distribuidos, a saber :

- *deadlock de comunicaciones*

Estos son del estilo de :
A manda mensaje a B
B manda mensaje a C
C manda mensaje a A

En ciertas circunstancias esto puede llevar a un deadlock.

- *deadlock de recursos*

Cuando dos procesos pelean por el acceso exclusivo a dispositivos de E/S, archivos, bloqueos u otros recursos.

No vamos a usar esta distinción ya que los recursos de comunicación son también recursos e igual se pueden modelizar.

Además las esperas circulares como la descripta no son frecuentes, por ejemplo en el modelo cliente-servidor es raro que el servidor desee enviar un mensaje al cliente que lo llamó actuando él como un cliente y esperando que el cliente actúe como servidor con lo cual la espera circular es extraño que ocurra como consecuencia de solamente la comunicación en sí.

Se utilizan cuatro estrategias para manejar abrazos mortales :

1) - El algoritmo del avestruz (ignorar el problema)
2) - Detección (dejar que ocurra, detectarlo y tratar de recuperar)
3) - Prevención (hacer que estáticamente el abrazo mortal sea imposible desde el punto de vista de la estructura)
4) - Evitar (evitar el abrazo alocando recursos cuidadosamente)

El algoritmo 1) es tan usado como en los sistemas monoprocesadores. En instalaciones de oficina, programación, control de procesos, y muchas otras no existe ningún algoritmo de prevención. No obstante ello aplicaciones de bases de datos pueden implementar su propio algoritmo si lo necesitan.

El algoritmo 2) es muy popular porque los otros dos son muy difíciles.

La prevención (3) es también posible pero aún más difícil de implementar que con un solo procesador. Sin embargo en presencia de las transacciones atómicas algunas nuevas opciones están disponibles y que no las teníamos antes.

El último algoritmo (4) no se usa nunca en sistemas distribuidos. Si no se usa casi en sistemas monoprocesadores ¿porqué usarlo aquí en donde es aún más dificultoso ? El problema es que algoritmos como el del Banquero necesitan conocer de antemano cuantas instancias de los recursos se utilizarán y esta información no está disponible en los sistemas distribuidos.

Veremos solo dos de las técnicas : detección y prevención.

23.5.1. - Detección de abrazo mortal

Cuando se detecta un abrazo mortal en un sistema monoprocesador la solución es matar uno o más procesos lo que deriva en uno o más usuarios infelices. Pero en un sistema distribuido gracias a las propiedades y por el diseño de las transacciones atómicas el matar una o más transacciones no acarrea mayor dificultad. Luego las consecuencias son menos severas al matar un proceso si se usan las transacciones atómicas que si no se usan.

23.5.1.1. - Detección centralizada de Abrazo Mortal

Esto es parecido al caso monoprocesador. Si bien cada máquina tiene los grafos de alocación de sus propios procesos y recursos existe un coordinador centralizado que mantiene el grafo de recursos de todo el sistema.

Cuando el coordinador detecta un ciclo mata un proceso para romper el abrazo.

Este grafo debe actualizarse constantemente.

Una forma es que cuando un arco de agrega o elimina se debe enviar un mensaje al coordinador.

Otra es que cada x tiempo cada proceso envía una lista de arcos a agregar o eliminar (requiere menos mensajes que el anterior).

Otra es que el coordinador pida información cuando la necesita.

Lamentablemente ninguno de estos métodos trabaja bien.

Supongamos la situación (Fig. 23-14) en la máquina 0 en la cual el proceso A tiene el recurso S y espera el recurso R que está asignado el proceso B.

En la máquina 1 el proceso C tiene T y espera por S.

La situación vista por el coordinador es correcta. Tan pronto como B termine a podrá obtener R y terminar liberando S para C.

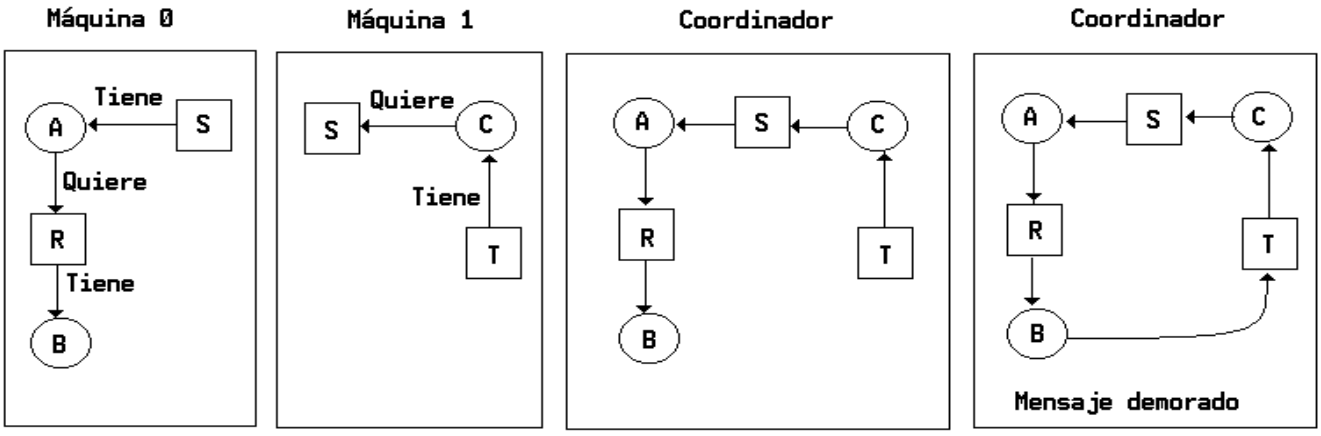


Fig.23.14. - Detección centralizada de Abrazo Mortal.

Luego de un tiempo B libera R y solicita T, y la máquina 1 avisa que B está esperando por su recurso T. Pero el mensaje de la máquina 1 llega primero al coordinador el cual al agregar el arco detecta la existencia de un abrazo mortal llegando a la conclusión de que algún proceso debe morir.

Esta situación es llamada un **falso abrazo mortal** o falso ciclo. Muchos algoritmos en sistemas distribuidos producen falsos abrazos mortales debido a información incompleta o demorada.

Una forma de solución es utilizar el algoritmo de Lamport para proveer un tiempo global. El mensaje de la máquina 1 se dispara luego del requerimiento de la máquina 0 por lo tanto deberá tener un sello temporal mayor.

Cuando el coordinador recibe el mensaje de la máquina 1 que le lleva a sospechar la existencia de un abrazo mortal deberá enviar a todas las máquinas un mensaje que diga : "Acabo de recibir un mensaje con sello temporal T que me provoca abrazo mortal. Alguien tiene un mensaje para mí con menor sello temporal ? Por favor, envíenmelo inmediatamente".

A pesar de que este método elimina los falsos abrazos mortales requiere de tiempo global y es caro. Más aún, existen otras situaciones en las que la eliminación del falso abrazo mortal es aún más difícil.

Puede suceder también que existan rollbacks innecesarios porque mientras el coordinador envió un rollback por alguna razón otro de los procesos envueltos en el deadlock dejó de ejecutar.

23.5.1.2. - Detección jerárquica de abrazo mortal

En este esquema en los nodos superiores se va guardando la información referente a los nodos inferiores, cada uno de los cuales guarda su propio grafo. El ejemplo puede visualizarse en la Fig. 23.15.

23.5.1.3. - Detección distribuida de abrazo mortal

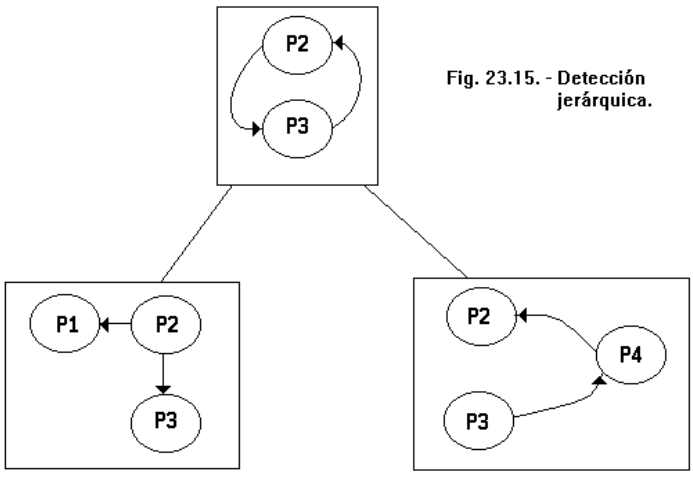


Fig. 23.15. - Detección jerárquica.

Veamos el algoritmo de Chandy-Misra-Haas (1983), en este algoritmo los procesos pueden pedir múltiples recursos de a una vez en lugar de a uno por vez.

Esto permite acelerar considerablemente la fase de crecimiento de una transacción. La consecuencia es que un proceso puede ahora esperar por do o más recursos simultáneamente.

En el dibujo (Fig. 23-16) eliminamos las flechas de los recursos e indicamos solo el encadenamiento entre los procesos. El proceso 3 en la máquina 1 está esperando dos recursos, uno retenido por el proceso 4 y el otro por el proceso 5. Algunos procesos están esperando recursos locales (proceso 1) y otros esperan recursos de otras máquinas (proceso 2).

Estos últimos arcos son los que hacen dificultoso ver los ciclos.

Se invoca al algoritmo cuando algún proceso tiene que esperar por algún recurso (por ej. el proceso 0 espera por el proceso 1).

En este punto se envía un mensaje de **prueba** a los procesos que retienen el recurso. Este mensaje consta de tres números: el proceso que se está bloqueando, el proceso que envía el mensaje y el proceso que retiene el recurso, en este caso el mensaje contiene la terna (0, 0, 1).

Cuando el mensaje llega el receptor chequea para ver si él mismo está esperando a algún proceso. De ser así actualiza el mensaje manteniendo el primer campo pero reemplazando el segundo por sí mismo y el tercero por el número del proceso al cual está esperando. El mensaje es luego ruteado al proceso sobre el cual él se encuentra bloqueado. Si está bloqueado sobre varios a todos ellos se les envía mensaje (diferente).

El algoritmo es igual aunque el recurso sea local o remoto.

En la figura podemos ver los mensajes (0,2,3), (0,4,6), (0,5,7) y (0,8,0).

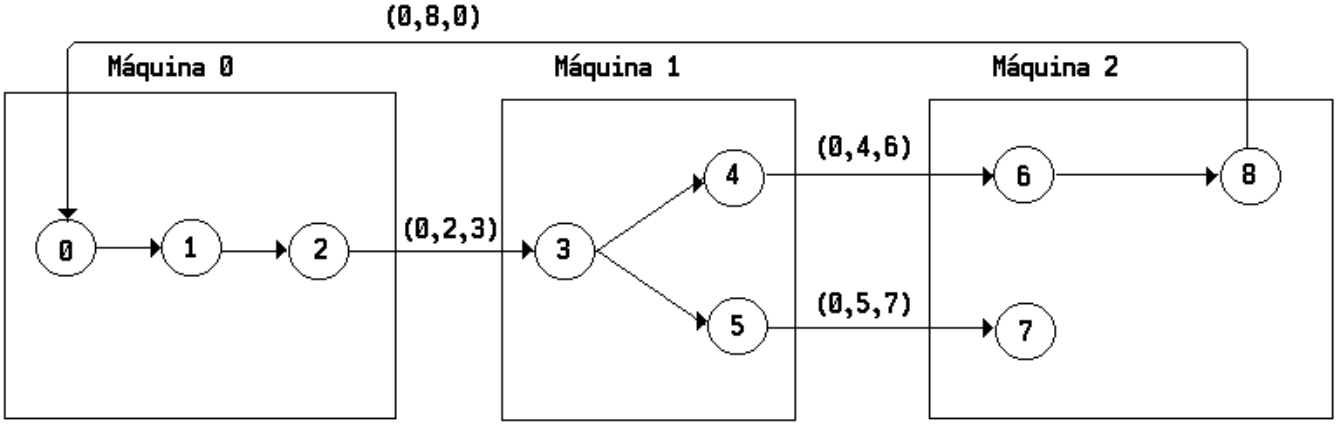


Fig. 23.16. - Detección distribuida.

Si el mensaje da toda la vuelta y vuelve al emisor original entonces existe un abrazo mortal.

Este abrazo se puede romper de varias formas. Por ejemplo el primer emisor del mensaje puede cometer suicidio.

No obstante este método tiene problemas si varios procesos invocan simultáneamente al algoritmo. Supongamos que tanto el proceso 0 como el proceso 6 se bloquean al mismo tiempo y ambos inician mensajes de prueba.

Cada uno descubrirá el abrazo mortal y se suicidará. Esto provoca que mueran demasiados procesos cuando la muerte de uno de ellos bastaba.

Una alternativa sugiere agregar la propia identidad de cada proceso al final del mensaje de manera tal que cuando vuelve al emisor inicial podría listarse el ciclo completo. El emisor podría entonces ver cual proceso tiene el mayor número y enviarle un mensaje solicitándole que se mate. De esta forma varios procesos que descubren el ciclo elegirían la misma víctima.

En el campo de los sistemas distribuidos la teoría difiere aún más de la realidad. Por ejemplo enviar un mensaje de prueba cuando se está bloqueado es algo no trivial.

Existe mucha investigación en este campo pero una vez que hay un método nuevo alguien muestra un contraejemplo.

En conclusión, tenemos mucha investigación, modelos que no se ajustan bien a la realidad, las soluciones halladas son generalmente impracticables, los análisis de performance dados son poco significativos y los resultados probados son a menudo incorrectos. Como algo positivo esta es un área que ofrece grandes oportunidades de mejoras.

23.5.2. Prevención distribuida de abrazo mortal

La prevención consiste en diseñar cuidadosamente el sistema de manera tal que los deadlocks sea estructuralmente imposibles.

Muchas técnicas incluyen :

- los procesos sólo pueden retener de a un recurso por vez,
- los procesos deben requerir todos sus recursos al inicio,
- los procesos deben liberar todos sus recursos al requerir uno nuevo.

Estas prácticas son engorrosas en la práctica.

Otro método es el ordenamiento de los recursos y exigir que los requerimientos de los procesos se realicen estrictamente en ese orden (vimos que está libre de deadlocks).

No obstante en un sistema distribuido con tiempos globales y transacciones atómicas existen otros dos métodos que son posibles.

Ambos se basan en asignarle a cada transacción un sello temporal en el momento en que la misma se inicia. Es esencial que dos transacciones no puedan tener nunca el mismo sello temporal para ello se puede utilizar el algoritmo de Lamport.

La idea es que cuando un proceso se va a bloquear en espera de un recurso que está utilizando otro proceso se chequea cuál de los dos tiene un mayor sello temporal (cuál es el más joven). Se puede permitir entonces el bloqueo si el que se bloquea tiene un menor sello temporal, es decir es más viejo que el que posee el recurso en ese momento.

De esta forma siguiendo una cadena de procesos en espera el sello temporal de ellos siempre va in crescendo con lo cual la creación de ciclos es imposible.

También es factible permitir el bloqueo si el que se quiere bloquear es más joven que el que retiene el recurso (es más joven) en cuyo caso la cadena de procesos en espera tiene sellos temporales que van decreciendo.

Si bien ambos métodos previenen el abrazo mortal es sensato otorgar prioridad a los procesos más viejos ya que han procesado por más tiempo, el sistema invirtió mucho ya en ellos y es probable que retengan más recursos. Además un proceso joven que se suicida eventualmente envejece al punto tal en que es el más viejo y de esta forma se elimina la inanición.

Como vimos antes matar una transacción es relativamente inocuo ya que por definición se la puede restaurar en forma segura más tarde.

Para aclarar consideremos la situación de la figura 23-17. En (a) un proceso viejo desea un recurso retenido por uno más joven. En (b) un proceso joven desea un recurso retenido por uno más viejo.

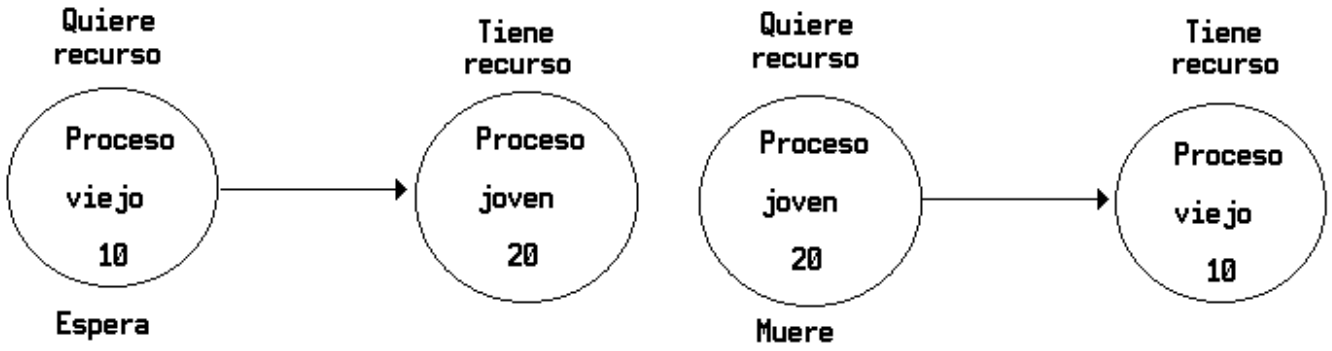


Fig. 23.17. - El algoritmo Wait-Die.

En un caso permitiremos la espera en tanto que en otro matamos al proceso. Supongamos que designamos que en (a) muere y en (B) espera. Lo que estamos haciendo es matar un proceso más viejo que desea un recurso retenido por uno más joven, esto es ineficiente, entonces tenemos que designarlos de otra manera que es como se ve en la figura.

Bajo estas condiciones las flechas apuntan siempre en una dirección creciente de número de transacción haciendo que los ciclos sean imposibles de ocurrir. Este algoritmo se denomina espera-muerte (wait-die).

Una vez que asumimos la existencia de transacciones podemos hacer algo que estaba prohibido anteriormente: alejar los recursos de los procesos que se ejecutan.

Estamos diciendo que cuando existe un conflicto tanto podemos matar al proceso que realiza el requerimiento como al propietario del recurso.

Sin transacciones matar un proceso puede tener consecuencias severas en cuanto a lo que el proceso modificó, etc. Con transacciones estos efectos desaparecen mágicamente cuando la transacción muere.

Consideremos la situación de la figura 23-18 en la cual vamos a permitir el desalojo. Ya que nuestro sistema cree en el trabajo de los ancestros no deseamos que un joven mequetrefe desaloje a un sabio anciano, y por tal motivo rotulamos la parte a) como desalojo y la parte b) como espera.

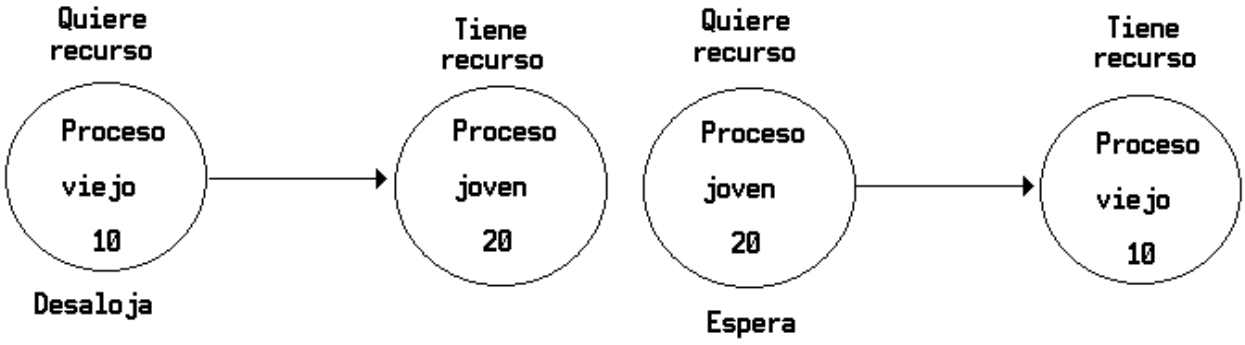


Fig. 23.18. - El algoritmo Wound-Wait.

Este algoritmo se denomina golpear-esperar (wound-wait) debido a que una transacción supuestamente es golpeada o herida (y en realidad muere) y la otra espera.

En el caso a) el pedido de la transacción más vieja provoca la muerte de la transacción más joven desalojándola. El joven comenzará de nuevo y pedirá el recurso otra vez lo que lo colocará en espera como se ve en la parte b).

En el método espera-muere si un proceso viejo quiere un recurso retenido por un joven espera. En cambio si es un joven el que quiere el recurso retenido por un viejo el joven muere, comenzará de nuevo y pedirá el recur-

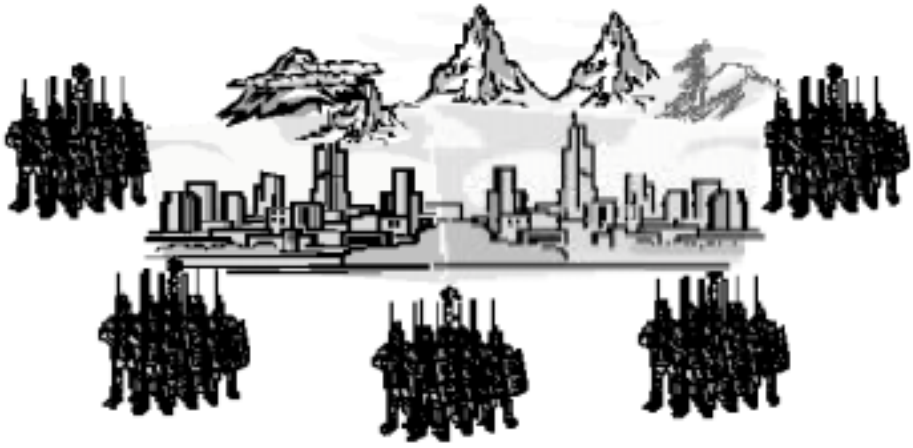
so muriendo nuevamente, y así siguiendo hasta que el proceso viejo libere el recurso. Esta característica desagradable no existe en el método golpear-esperar.

Ambos esquemas permiten la inanición pues las "horas" (que son dadas a cada proceso en el momento de su creación) no son actualizadas.

Para evitar la inanición en ambos esquemas es importante que a una nueva transacción no le asigna un nuevo sello temporal cuando se la reinicia luego de haber sido abortada (una transacción joven de esta forma se volverá más vieja con el transcurso del tiempo y no será cancelada).

23.6. CONSENSO EN PRESENCIA DE FALLAS - El problema de los Generales Bizantinos

Hasta ahora no hemos visto como lograr que un grupo de nodos llegue a un acuerdo considerando la existencia de fallas. Este problema suele mencionarse en la bibliografía como *el problema de los Generales Bizantinos* planteado originalmente por Lamport , Shostak y Peace en una publicación de la ACM en Julio de 1982.



Un grupo de generales sitia una ciudad y deben ponerse de acuerdo en un plan de ataque, ya sea atacar o retirarse, independientemente de que existan generales traidores. Los generales solo se comunican a través de mensajes a los otros generales.

Esta traición puede verse de dos formas:

- los mensajes pueden no llegar, o dicho de otra forma, las comunicaciones son no confiables
- un general traidor puede mentir, o sea, un nodo puede fallar de manera impredecible.

23.6.1. – **Definición del problema**

Un general comandante debe enviar una orden a sus n-1 tenientes generales de manera tal que (Dada una red con n procesos que se comunican entre sí solo a través del pasaje de mensajes sobre canales bidireccionales, asegurar que un proceso envía un ítem a los otros n-1 procesos de manera tal que) :

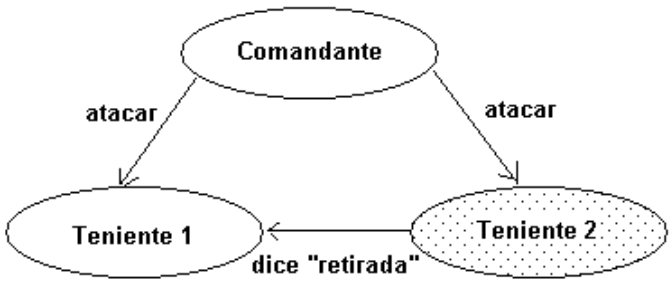
Premisa 1) Todos los tenientes leales obedecen la misma orden (los procesos confiables reciben el mismo ítem)

Premisa 2) Si el general comandante es leal, entonces todos los tenientes leales obedecen la orden que el envió (si el proceso que envía el mensaje es confiable entonces el ítem recibido es igual al ítem enviado)

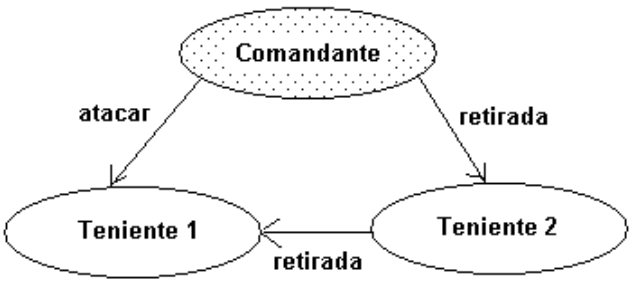
Las premisas 1) y 2) son conocidas como las condiciones de **consistencia interactivas**. Obsérvese que si el comandante es leal la condición P1) se deriva de la condición P2). Sin embargo el comandante puede ser un traidor.

23.6.2. – **Resultados de imposibilidad**

Este problema no es siempre soluble, imagínese el caso de 3 generales :



Caso A - El teniente 2 es un traidor



Caso B - El comandante es un traidor.

En el caso A) para satisfacer P2) el teniente 1 debería atacar. En el caso B) si el teniente 1 ataca viola P1) El teniente 1 no puede distinguir entre ambos casos con la información de que dispone.

No existe solución para el caso de 3 generales que operan en presencia de un traidor.
Generalizando: No existe solución con menos de **3 m + 1** generales cuando existen **m** traidores.

23.6.3. – **Una solución con mensajes sin firma (MSF)**.

Se realizan las siguientes presunciones a priori :

A1)- cada mensaje que se envía se entrega correctamente
A2)- El receptor de un mensaje sabe quién lo envió
A3)- La ausencia de un mensaje puede ser detectada

Las presunciones A1 y A2 evitan que un traidor interfiera las comunicaciones entre dos generales, ya que por A1 no puede interferir con los mensajes que se envían y por A2 no puede provocar confusión introduciendo mensajes espurios. A3 impide al traidor que otro general tome una decisión por la sencilla razón de no haber recibido un mensaje.

En un sistema distribuido las presunciones A1 y A2 indican que en la comunicación la falla del link de conexión se considera como una más de las fallas **m** (un traidor) ya que no se puede distinguir entre una falla de conexión o un proceso que falle en un nodo. La presunción A3 requiere que el emisor y receptor tengan una mecanismo de sincronización de relojes y además que se conozca el tiempo máximo de generación y transmisión de un mensaje.

23.6.3.1. – **El algoritmo MSF**

Este algoritmo vale para **n** generales y **m** traidores con **n > 3 m**.

Se requiere de un valor por defecto $\mathbf{v_{def}}$ por si el general traidor no envía un mensaje (por ejemplo RETIRADA)

Se define una función **mayoría(v_1,.....,v_{n-1}) = v** si la mayoría de los valores de $\mathbf{v_j = v}$.

Algoritmo MSF(n, 0) No hay traidores

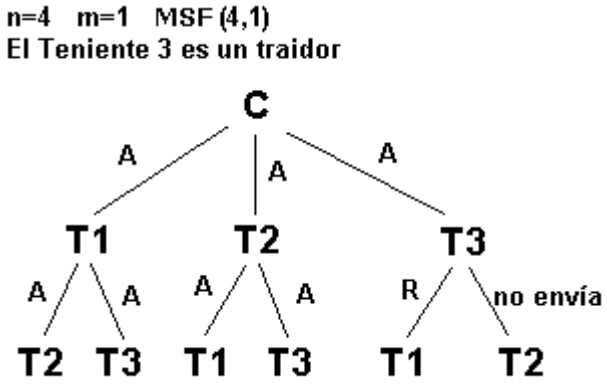
(1) El general comandante envía **v** a cada teniente
(2) Cada teniente utiliza el valor recibido del comandante o $\mathbf{v_{def}}$ si no recibe un valor

Algoritmo MSF(n, m) Hay **m** traidores

(1) El general comandante envía **v** a cada teniente
(2) Para cada **Teniente** $_i$
    Sea $\mathbf{v_i}$ el valor recibido del comandante o $\mathbf{v_{def}}$ si no recibe un valor
    Enviar $\mathbf{v_i}$ a los n-2 otros tenientes utilizando **MSF(n-1, m-1)**
(3) Para cada **i** y cada **z** con **z ≠ i**
    Sea $\mathbf{v_j}$ = el valor que el Teniente$_i$ recibió del Teniente$_z$ en el paso (2) o $\mathbf{v_{def}}$ si no recibe un valor
    El Teniente$_i$ utiliza el valor de la función **mayoría(v_1,.....,v_{n-1})**

Veamos este algoritmo con un par de ejemplos. Supongamos que el mensaje enviado por el Comandante es ATACAR (o A abreviadamente) y el valor por defecto es RETIRADA (o R abreviadamente).

1er Caso: El Teniente es traidor



Al final del paso 1 tenemos que:
L1 : v1 = A
L2 : v2 = A
L3 : v3 = A
Al finalizar el paso 3 se tiene que:
L1 : v1 = A v2 = A y v3 = R
L2 : v1 = A v2 = A y v3 = R (valor por defecto)
L3 : v1 = A v2 = A y v3 = A
Al final de esta etapa cada Teniente tiene un conjunto de valores y arriba a la misma decisión (respeta P1) y el valor enviado por C es el valor de la función **mayoría** (respeta P2).

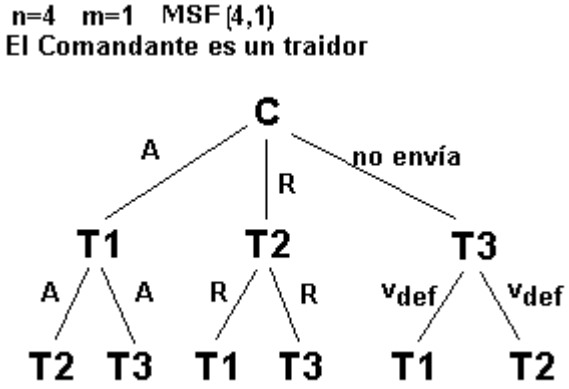
2do Caso: El Comandante es traidor

Al final del paso 1 tenemos que:

L1 : v1 = A
L2 : v2 = R
L3 : v3 = R (valor por defecto)

Al finalizar el paso 3 se tiene que:

L1 : v1 = A v2 = R y v3 = R (valor por defecto)
L2 : v1 = A v2 = R y v3 = R (valor por defecto)
L3 : v1 = A v2 = R y v3 = R (valor por defecto)



Los tres tenientes leales reciben el mismo valor de la función mayoría y las premisas P1 y P2 se respetan. Nótese en este caso que como el Comandante **no** es leal su orden no es acatada aunque todos los tenientes actúan coordinadamente.

Un lema que se desprende de este algoritmo indica que : Para cualquier **m** y **k** el algoritmo **MSF(m)** satisface la Premisa 2 si hay al menos **2k+m** generales y a lo sumo **m** traidores.

Por otra parte para cualquier **m** el algoritmo **MSF(m)** satisface las premisas 1 y 2 si hay más de **3m** generales y a lo sumo **m** traidores.

26.6.3.1. – **Complejidad de MSF(n,m)**

Al aplicar **MSF(n,m)** primero se envía **n-1** mensajes. Cada mensaje invoca **MSF(n-1, m-1)** lo que provoca que se envíen **n-2** mensajes, etc.

Etapa 1 : **(n-1)** mensajes
Etapa 2 : **(n-1)(n-2)** mensajes
.....
Etapa m+1 : **(n-1)(n-2).....(n-(m+1))** mensajes

Luego, el total de mensajes es $\mathbf{O(n^{m+1})}$

Nótese que las **m+1** etapas de intercambio de mensajes es una característica fundamental de los algoritmos en los que se logra consenso en presencia de **m** posibles procesos que fallen.

23.6.4. – **Una solución con mensajes firmados MF**

Lo que se busca es restringir la posibilidad de que el traidor mienta permitiendo a los generales enviar mensajes firmados que no pueden ser alterados.

Se agrega una presunción a las hechas anteriormente:

A4): (a) La firma de un general leal no puede ser alterada y cualquier alteración del contenido de sus mensajes firmados puede ser detectado.
(b) Cualquiera puede verificar la autenticidad de la firma del general

Adoptaremos la siguiente notación:

V:j:i – valor **v** firmado por **j** y luego el valor **v:j** ha sido firmado por **i**. El general 0 es el comandante.

Se necesita una función **elección(v)** la cual selecciona el valor de **v** de un conjunto de valores de **v** de forma tal que :

Elección ({v}) = v;
Elección ($\{v_{def}\}$) = v_{def};

Esta función es utilizada para obtener el valor del consenso y no necesariamente es el valor de la mayoría ni un promedio.

23.6.4.1. – **El algoritmo MF**

En este algoritmo cada teniente **i** mantiene un conjunto $\mathbf{V_i}$ de ordenes correctas firmadas que ya ha recibido. Con un comandante leal el conjunto no contiene más de un elemento.

Algoritmo MF(m) Inicialmente $\mathbf{V_i = \{\}}$

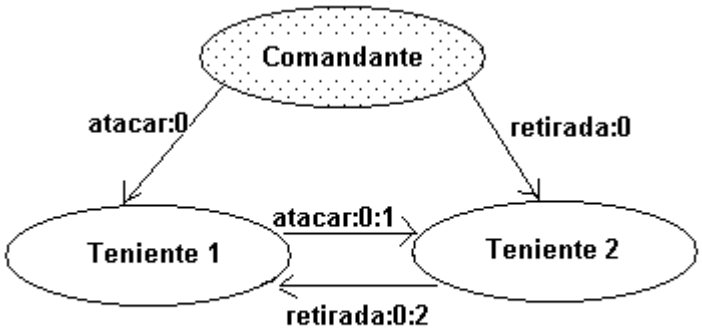
(1) El comandante envía su valor firmado a cada teniente
(2) Para cada **i** :
  (a) Si el teniente **i** recibe un mensaje **v:0** y aún no ha recibido una orden, entonces:
    (i) establece $\mathbf{V_i = \{v\}}$
    (ii) envía **v:0:i** a los otros tenientes
  (b) Si el teniente **i** recibe un mensaje de la forma $\mathbf{v:0:j_1: ... :j_k}$ y **v** no está en el conjunto $\mathbf{V_i}$ entonces:

(i) $V_i := V_i + \{v\}$

(ii) Si **k < m** entonces envía el mensaje **v:0:j_1: ... :j_k:i** a cada teniente de **j_1: ... :j_k**

(3) Cuando no se reciben más mensajes el teniente **i** obedece la orden que surge de **elección(V_i)**

Veamos un ejemplo:



v1 = v2 = elección({atacar,retirada})

Nótese que con mensajes firmados los tenientes pueden detectar que el comandante es un traidor debido a que su firma aparece en dos órdenes opuestas y por el supuesto A4 solo él pudo haberlas firmado.

En este caso siempre es posible lograr un acuerdo en tanto y en cuanto existan por lo menos dos generales leales.

26.3.4.2.- **Complejidad**

La cantidad de mensajes es del orden de **$O(n^{m+1})$**

La cantidad de etapas : **m + 1**

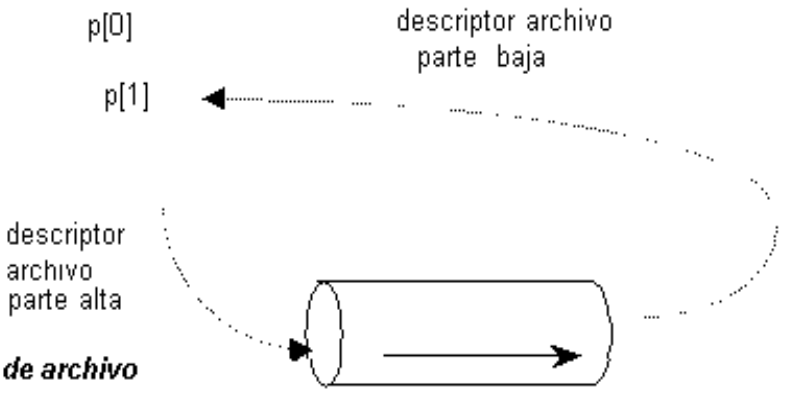
23.7. COMUNICACIÓN ENTRE PROCESOS - INTER PROCESS COMMUNICATION (IPC)

Hasta ahora hemos visto mecanismos de comunicación entre procesos con uso de memoria común, como semáforos, regiones críticas, monitores, etc. y en forma incipiente, mecanismos que no utilizan memoria común, por medio de primitivas send y receive, el modelo cliente/servidor y una introducción al RPC.

A continuación, haremos una breve introducción a un mecanismo de comunicación que utiliza archivos especiales, los pipes, para luego extender el concepto a mecanismos de comunicación que utilizan un principio similar, los sockets, pero que permiten la comunicación de procesos que residen en nodos (computadoras) ligados por algún medio de comunicación.

23.7.1. PIPES

int p(2);
pipe(p);



descriptor de archivo parte alta

Los pipes o tubos son archivos que permiten un mecanismo de comunicación entre procesos que cuenta de dos partes:

Alta: por la cual es posible escribir

Baja: desde la cual se puede leer

Se trata de un mecanismo que maneja una estructura de cola (FIFO) en el cual la información transmitida por un lado es recibida y puede ser retirada por el otro lado.

La comunicación es unidireccional, sincrónica y unioperacional, pues no es posible escribir y leer al mismo tiempo. O sea que mientras un proceso realiza una de las dos operaciones permitidas (en realidad la que se le permite) el otro espera su finalización para comenzar la suya.

Al estar compartiendo un mismo recurso, el pipe, este debe ser heredado de otro proceso que lo haya generado, o sea son utilizados para establecer comunicación entre dos procesos que posean el mismo padre.

Se crean a partir de una llamada al sistema pipe() (ver figura)

Los pipes tienen una capacidad finita.

Si el proceso de la parte alta del pipe escribe en él y el pipe está lleno, el proceso se bloqueará hasta que la parte baja libere datos.

Si el proceso de la parte baja del pipe realiza una lectura y el pipe está vacío el proceso se bloquea.

Por ejemplo si se tiene:

While ((count = read(fd, buffer, 100)) > 0 { procesa }

Si suponemos que el buffer tiene 230 bytes, luego de dos lecturas, quedan 30, que será lo que leerá en la tercera, en su cuarto intento se bloqueará.

Si un proceso intenta leer la parte baja del pipe, que no tiene abierta su parte alta, la lectura regresará un EOF (fin de archivo).

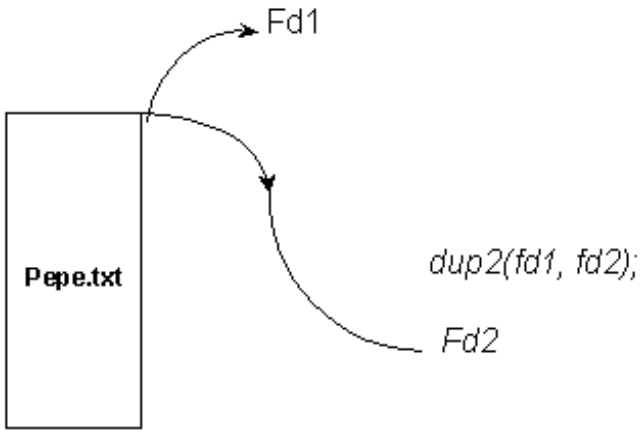
Como los pipes son archivos, es necesario generar sus descriptores, en general se reutiliza un archivo ya abierto por el padre, en cuyo caso se utiliza una llamada al sistema que duplica un descriptor de un archivo ya abierto (**dup2()**), de la manera que se aprecia en la figura.

#include <unistd.h>

int dup2(int fd1, int fd2);

Cuyo resultado es:

- El descriptor de archivos fd2 va a apuntar al mismo archivo que fd1.
- Fd1 es un descriptor de archivos que hace referencia a un archivo abierto.
- Fd2 es un entero no negativo menor al número de descriptores de archivos abiertos permitidos.
- Si fd2 apuntaba a un archivo abierto, diferente de fd1, primero cierra el archivo en cuestión y luego apunta al de fd1.



23.7.2. **Ejemplos de uso de pipes**

Veamos un sencillo ejemplo en el que se utilizan dos comandos UNIX, **who I more**

```
/* Archivo del programa whomore.c  */
main()
{
      int fds[2]

      pipe(fds);
      /* Hijo1 reconecta stdin a  parte baja del pipe y cierra alta */
      if (fork() == 0)  {
        dup2(fds[0], 0);
        close(fds[1]);
        execlp("more","more",0);
      }
      else {
       /* Hijo2 reconecta stdout  a  parte alta del pipe y cierra baja */
        if ( fork() == 0 )      {
           dup2(fds[1], 1);
           close(fds[0]);
           execlp("who", "who", 0);
        }
        else {       /* padre cierra ambas partes y espera a los hijos */
           close(fds[0]);
           close(fds[1]);
           wait(0);
           wait(0):
        }
}
```

Veamos otro ejemplo en que dos "hermanos" mantienen comunicación por **pipes**

```
#include <stdio.h>
#define BLKSIZE 80

main()
   int n;
   char msg[80];
   int fd[2];

   if  ( pipe(fd) < 0 ) {
     fprintf(stderr,"Error en la creación del pipe \n");
     exit(1);
   }
   if ( fork() == 0) {
       close(fd[0]);
       sprintf(msg, "Mensaje envía por %d",getpid());
       write(fd[1], msg, BLKSIZE)
   }
   else
       if ( fork() == 0) {
          close(fd[1]);
          read(fd[0], msg, BLKSIZE);
          printf("Proceso %d recibió mensaje: %s \n",getpid(), msg);
       }
       else {
          close(fd[0]);  close(fd[1]);
          wait(0);   wait(0);
          printf("Los hijos terminaron de comunicarse \n");
       }
}
```

Las limitaciones de los pipes residen en:

1. Pipes son unidireccionales. La solución para lograr comunicación en dos sentidos es crear dos pipes.
2. Pipes no pueden autentificar al proceso con el que mantiene comunicación.
3. Pipes deben de ser pre-arreglados. Dos procesos no relacionados no pueden conectarse vía pipes, deben tener un ancestro común que cree el pipe y se los herede.
4. Pipes no trabajan a través de una red. Los dos procesos deben encontrarse en la misma máquina.

23.8. **SOCKETS**

Los sockets no son mas que puntos o mecanismos de comunicación entre procesos que permiten que un proceso hable (emita o reciba información) con otro proceso incluso estando estos procesos en distintas máquinas. Esta característica de interconectividad entre máquinas hace que el concepto de socket nos sirva de gran utilidad. Esta interfaz de comunicaciones es una de las contribuciones Berkeley al sistema UNIX, implementándose las utilidades de interconectividad de este Sistema Operativo (rlogin, telnet, ftp,...) usando sockets.

La forma de referenciar un socket por los procesos implicados es mediante un descriptor del mismo tipo que el utilizado para referenciar archivos. De hecho muchos desarrolladores dicen que un socket no es más que un pipe con un protocolo asociado y bidireccional.

Debido a esta característica, se podrá realizar redirecciones de los archivos de E/S estándar (descriptores 0,1 y 2) a los sockets y así combinar entre ellos aplicaciones de la red. Todo nuevo proceso creado heredará, por tanto, los descriptores de sockets de su padre.

La comunicación entre procesos a través de sockets se basa en la filosofía CLIENTE/SERVIDOR: un proceso en esta comunicación actuará de proceso servidor creando un socket cuyo nombre conocerá el proceso cliente, el cual podrá "hablar" con el proceso servidor a través de la conexión con dicho socket nombrado.

El proceso crea un socket sin nombre cuyo valor de retorno es un descriptor sobre el que se leerá o escribirá, permitiéndose una comunicación bidireccional, característica propia de los sockets y que los diferencia de los pipes, o canales de comunicación unidireccional entre procesos de una misma máquina. El mecanismo de comunicación vía sockets tiene los siguientes pasos:

1º) El proceso servidor crea un socket con nombre y espera la conexión.
2º) El proceso cliente crea un socket sin nombre.
3º) El proceso cliente realiza una petición de conexión al socket servidor.
4º) El cliente realiza la conexión a través de su socket mientras el proceso servidor mantiene el socket servidor original con nombre.

Es muy común en este tipo de comunicación lanzar un proceso hijo, una vez realizada la conexión, que se ocupe del intercambio de información con el proceso cliente mientras el proceso padre servidor sigue aceptando conexiones. Para eliminar esta característica se cerrará el descriptor del socket servidor con nombre en cuanto realice una conexión con un proceso socket cliente.

Antes de seguir avanzando repasemos nuestros conocimientos sobre redes y en particular sobre el modelo TCP/IP

23.8.1. **El modelo TCP/IP**.

Los cinco niveles del modelo TCP/IP son:

	TCP/IP	OSI
(NFS)		7. Aplicación
(XDR)	5. Aplicación	6. Presentación
(RPC)		5. Sesión
(TCP/UDP)	4. Transporte	4. Transporte
(IP/ICMP)	3. Internet	3. Red
Trama Ethernet	2. Interfaz de Red	2. Enlace de Datos
Red Ethernet	1.Hardware	1. Físico

Nivel Interfaz de Red y Hardware:

Agrupa los bits en tramas para el manejo de la información y se ocupa de las características técnicas de la red (voltajes, pines, ...)

Nivel Internet:

Se corresponde con el nivel de Red OSI y controla el direccionamiento de la información. El IP se trata de un protocolo para el intercambio de datagramas en modo no conectado. Esto no garantiza la llegada de mensajes (cosa que se hará con el TCP). El algoritmo de direccionamiento de Internet se basa en tablas de direccionamiento de los datagramas difundidos por los gateways.

Nivel de Transporte:

Se corresponde con el de Transporte OSI, garantizando la seguridad de la conexión y el control del flujo. Incluye el Protocolo de Control de Transmisión (TCP) y el Protocolo de Datagrama de Usuario (UDP).

* El TCP es un protocolo orientado a conexión que transporta de forma segura grupos de octetos (segmentos) modo dúplex (en los dos sentidos).
* Utiliza el mecanismo de puerto (al igual que el protocolo de transporte UDP, pero que actúa en modo datagrama no conectado).

Este mecanismo se basa en la asignación para cada uno de los protocolos del nivel de transporte (TCP o UDP) de un conjunto de puertos de E/S identificados mediante un número entero. Así TCP y UDP multiplexarán las conexiones por medio de los números de los puertos. Existen una serie de puertos reservados a aplicaciones estándares Internet (ECHO - puerto 7, FTP - puerto 21, TELNET - puerto 23). El archivo **/etc/services** contiene la lista de los puertos estándar. En UNIX están reservados los números de puerto inferiores a 1024. El resto pueden ser utilizados, cualidad fundamental que aprovechan los programas definidos por el usuario para el establecimiento de comunicaciones entre nodos.

Nivel de Aplicación:

Incluye los niveles OSI de sesión, presentación y aplicación. Ejemplos de estos niveles son el telnet, ftp o el sistema de archivos de red NFS para el nivel de aplicación, el lenguaje de descripción de información XDR para el nivel de presentación o la interfaz de llamada a procedimientos remotos RPC.

Por ejemplo, el formato de una trama telnet sería con sus archivos UNIX asociados:

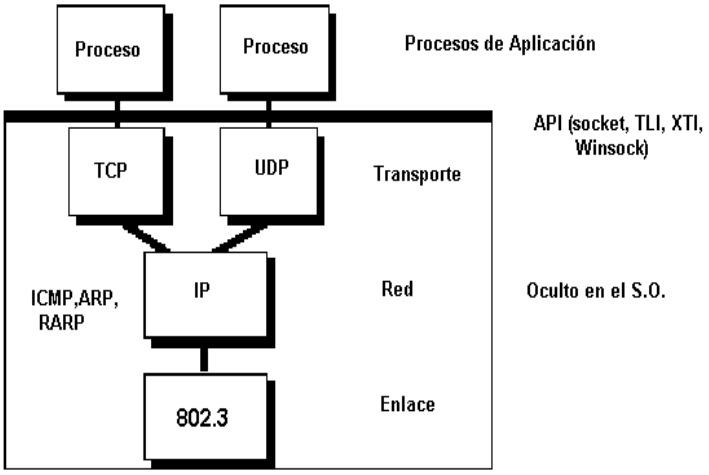
/etc/services

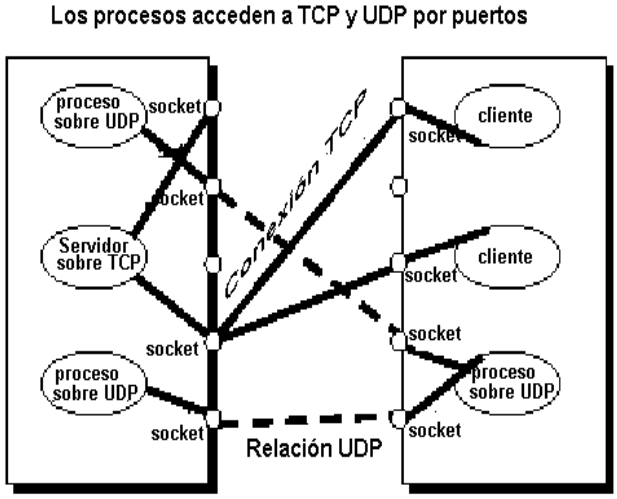
Dirección Ethernet	IP	TCP	telnetd

/etc/host /etc/protocols inetd.conf

En forma gráfica sería como puede visualizarse en la figura.

23.8.2. **Primera aproximación a los sockets**





Todo socket viene definido por dos características fundamentales:

- El **tipo del socket**, que indica la naturaleza del mismo, el tipo de comunicación que puede generarse entre los sockets.
- El **dominio del socke**t especifica el conjunto de sockets que pueden establecer una comunicación con el mismo.

Vamos a estudiar con más detalle estos dos aspectos:

23.8.2.1. **Tipos de sockets.**

Define las propiedades de las comunicaciones en las que se ve envuelto un socket, esto es, el tipo de comunicación que se puede dar entre cliente y servidor. Estas pueden ser:

- Fiabilidad de transmisión.
- Mantenimiento del orden de los datos.
- No duplicación de los datos.
- El "Modo Conectado" en la comunicación.
- Envío de mensajes urgentes.

Los tipos disponibles son los siguientes:

* Tipo SOCK_DGRAM: sockets para comunicaciones en modo no conectado, con envío de datagramas de tamaño limitado (tipo telegrama).
En dominios Internet como la que nos ocupa el protocolo del nivel de transporte sobre el que se basa es el UDP.

* Tipo SOCK_STREAM: para comunicaciones confiables en modo conectado, de dos vías y con tamaño variable de los mensajes de datos. Por debajo, en dominios Internet, subyace el protocolo TCP.

* Tipo SOCK_RAW: permite el acceso a protocolos de más bajo nivel como el IP (nivel de red)

* Tipo SOCK_SEQPACKET: tiene las características del SOCK_STREAM pero además el tamaño de los mensajes es fijo.

23.8.2.2. **El dominio de un socket.**

Indica el formato de las direcciones que podrán tomar los sockets y los protocolos que soportarán dichos sockets.

La estructura genérica es

```
struct sockaddr {
        u__short    sa__family;       /* familia */
        char        sa__data[14];     /* dirección */
    };
```

Pueden ser:

* Dominio AF_UNIX (Address Family UNIX):

El cliente y el servidor deben estar en la misma máquina. Debe incluirse el archivo cabecera /usr/include/sys/un.h. La estructura de una dirección en este dominio es:

```
struct sockaddr__un {
short         sun__family;  /* en este caso AF_UNIX */
char          sun__data[108]; /* dirección */
    };
```

* Dominio AF_INET (Address Family INET):

El cliente y el servidor pueden estar en cualquier máquina de la red Internet. Deben incluirse los archivos cabecera /usr/include/netinet/in.h, /usr/include/arpa/inet.h, /usr/include/netdb.h. La estructura de una dirección en este dominio es:

```
struct in__addr {
    u__long     s__addr;
};

struct sockaddr__in {
short          sin_family;  /* en este caso AF_INET */
u__short     sin_port;   /* numero del puerto */
struct in__addr          sin__addr; /* direcc Internet */
char          sin_zero[8];    /* campo de 8 ceros */
    };
```

Estos dominios van a ser los utilizados en xshine. Pero existen otros como:

* Dominio AF_NS:
  Servidor y cliente deben estar en una red XEROX.
* Dominio AF_CCITT:
  Para protocolos CCITT, protocolos X25, ...

23.8.3. **FILOSOFIA CLIENTE-SERVIDOR:**

23.8.3.1. **El Servidor.**

Vamos a explicar el proceso de comunicación servidor-cliente en modo conectado, modo utilizado por las aplicaciones estándar de Internet (telnet, ftp). El servidor es el proceso que crea el socket no nombrado y acepta las conexiones a él. El orden de las llamadas al sistema para la realización de esta función es:

1º) int socket (int dominio, int tipo, int protocolo)
  crea un socket sin nombre de un dominio, tipo y protocolo específico
  dominio : AF_INET, AF_UNIX
  tipo : SOCK__DGRAM, SOCK__STREAM
  protocolo : 0 (protocolo por defecto)

2º) int bind (int dfServer, struct sockaddr* direccServer, int longDirecc)

nombra un socket: asocia el socket no nombrado de descriptor dfServer con la dirección del socket almacenado en direccServer. La dirección depende de si estamos en un dominio AF_UNIX o AF_INET.

3º) int listen (int dfServer, int longCola)

especifica el máximo número de peticiones de conexión pendientes.

4º) int accept (int dfServer, struct sockaddr* direccCliente, int* longDireccCli)

escucha al socket nombrado servidor dfServer y espera hasta que se reciba la petición de la conexión de un cliente. Al ocurrir esta incidencia,
crea un socket sin nombre con las mismas características que el socket servidor original, lo conecta al socket cliente y devuelve un descriptor de archivo que puede ser utilizado para la comunicación con el cliente.

23.8.3.2. **El Cliente**

Es el proceso encargado de crear un socket sin nombre y posteriormente enlazarlo con el socket servidor nombrado. O sea, es el proceso que demanda una conexión al servidor. La secuencia de llamadas al sistema es:

1º) int socket (int dominio, int tipo, int protocolo)

crea un socket sin nombre de un dominio, tipo y protocolo específico
dominio : AF_INET, AF_UNIX
tipo : SOCK__DGRAM, SOCK__STREAM
protocolo : 0 (protocolo por defecto)

2º) int connect (int dfCliente, struct sockaddr* direccServer, int longDirecc)

intenta conectar con un socket servidor cuya dirección se encuentra incluida en la estructura apuntada por direccServer. El descriptor dfCliente se utilizará para comunicar con el socket servidor. El tipo de estructura dependerá del dominio en que nos encontremos.

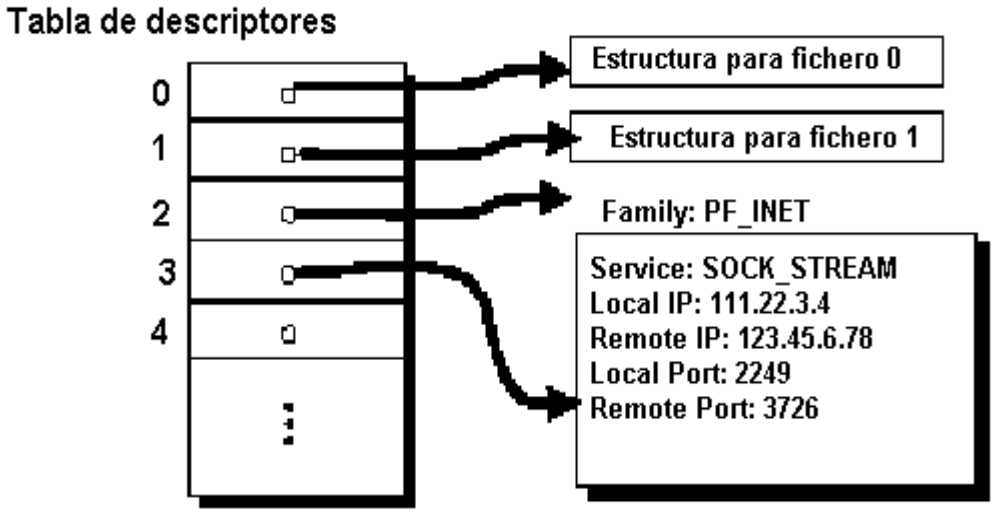
Una vez establecida la comunicación, los descriptores de archivos serán utilizados para almacenar la información a leer o escribir.

En forma esquemática se puede ver así:

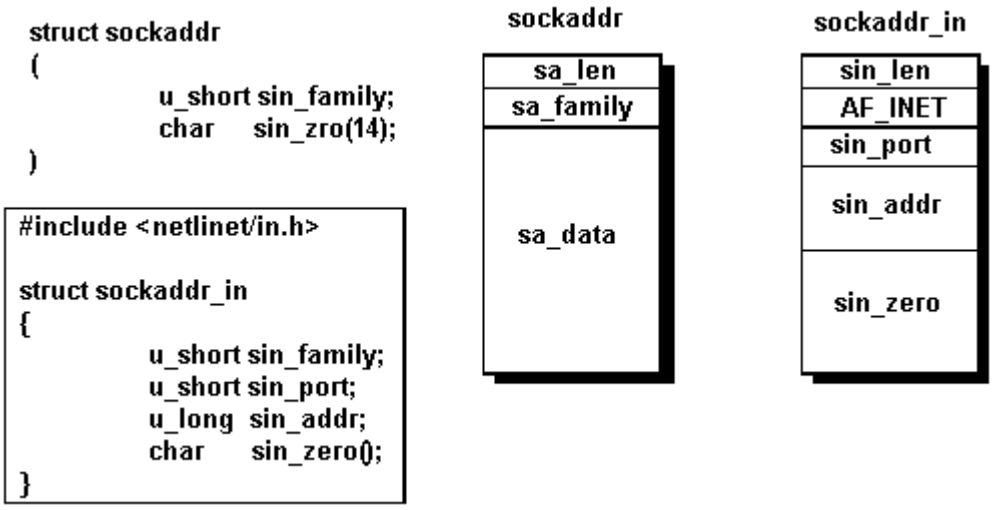
SERVIDOR	CLIENTE
DescrServer = **socket** (dominio, SOCK_STREAM,PROTOCOLO)	descrClient = **socket** (dominio, SOCK_STREAM,PROTOCOLO)
bind (descrServer, PuntSockServer,longServer)	
	do {
listen (descrServer, longCola)	
DescrClient = **accept** (descrServer,PuntSockClient,longClient)	result = **connect** (descrClient, PuntSockServer,longserver)
[close (descrServer)]	} while (result == -1)
< DIALOGO >	< DIALOGO >
close (descrClient)	**close** (descrClient)

23.8.4. **Estructuras de los sockets**

Una descripción gráfica de un socket como descriptor similar a la de archivos



Una descripción gráfica de las estructuras los sockets, (la estructura depende de la familia de protocolos):



Las estructuras son usadas en la programación de sockets para almacenar información sobre direcciones. La primera de ellas es struct sockaddr, la cual contiene información del socket.

```
struct sockaddr
{
  unsigned short sa_family; /* familia de la dirección */
  char sa_data[14]; /* 14 bytes de la dirección del protocolo */
};
```

Pero, existe otra estructura, struct sockaddr_in, la cual nos ayuda a hacer referencia a los elementos del socket.

```
struct sockaddr_in
{
  short int sin_family; /* Familia de la Dirección */
  unsigned short int sin_port; /* Puerto */
  struct in_addr sin_addr; /* Dirección de Internet */
  unsigned char sin_zero[8];
  /* Del mismo tamaño que struct sockaddr */
};
```

Nota. sin_zero puede ser seteada con ceros usando las funciones memset() o bzero() (Ver los ejemplos).

La siguiente estructura no es muy usada pero está definida como una unión.

Como se puede ver en los dos ejemplos de abajo (ver la sección de nombre Un ejemplo de Servidor de Flujos y la sección de nombre Un ejemplo de Cliente de Flujos), cuando se declara, por ejemplo "client" para que sea del tipo sockaddr_in, luego se hace client.sin_addr = (...).

De todos modos, aquí está la estructura:

```
struct in_addr
{
  unsigned   long s_addr;
};
```

Finalmente, creemos que es mejor hablar sobre la estructura hostent. En el ejemplo de Cliente de Flujos (ver la sección de nombre Un ejemplo de Cliente de Flujos), se puede ver cómo se usa esta estructura, con la cual obtenemos información del nodo remoto.

Aquí se puede ver su definición:

```
struct hostent
{
  char *h_name; /* El nombre oficial del nodo.  */
  char **h_aliases; /* Lista de Alias.  */
  int h_addrtype; /* Tipo de dirección del nodo.  */
  int h_length; /* Largo de la dirección. */
```

```
    char **h_addr_list; /* Lista de direcciones del nombre del servidor. */
    #define h_addr  h_addr_list[0]  /* Dirección, para la compatibilidad con anteriores. */
};
```

Esta estructura está definida en el archivo netdb.h.

Al principio, es posible que estas estructuras nos confundan mucho. Sin embargo, luego de empezar a escribir algunas líneas de código, y luego de ver los ejemplos que se incluyen, será mucho más fácil entenderlas. Para ver cómo se pueden usar estas estructuras, recomiendo ver los ejemplos de la sección de nombre Un ejemplo de Servidor de Flujos y la sección de nombre Un ejemplo de Cliente de Flujos.

23.8.5-**Conversiones**

Existen dos tipos de ordenamiento de bytes: bytes más significativos, y bytes menos significativos. Este es llamado “Ordenamiento de Bytes para Redes'', y hasta algunas máquinas utilizan este tipo de ordenamiento para guardar sus datos, internamente.

Existen dos tipos a los cuales seremos capaces de convertir: short y long. Imaginémonos que se quiere convertir una variable larga de Ordenación de Bytes para Nodos a una de Ordenación de Bytes para Redes. ¿Qué haríamos? Existe una función llamada **htonl()** que hará exactamente esta conversión. Las siguientes funciones son análogas a esta y se encargan de hacer este tipo de conversiones:

htons() -> ``Nodo a variable corta de Red''

htonl() -> ``Nodo a variable larga de Red''

ntohs() -> ``Red a variable corta de Nodo''

ntohl() -> ``Red a variable larga de Nodo''

Una cosa importante, es que sin_addr y sin_port, de la estructura sockaddr_in, deben ser del tipo Ordenación de Bytes para Redes. Se verá, en los ejemplos, las funciones que aquí se describen para realizar estas conversiones, y a ese punto se entenderán mucho mejor.

23.8.6. **Direcciones IP**

En C, existen algunas funciones que nos ayudarán a manipular direcciones IP. En esta sección se hablará de las funciones inet_addr() y inet_ntoa().

Por un lado, la función inet_addr() convierte una dirección IP en un entero largo sin signo (unsigned long int), por ejemplo:

```
(...)

dest.sin_addr.s_addr = inet_addr("195.65.36.12");

(...)

/*Recordar que esto sería así, siempre que tengamos una
estructura "dest" del tipo sockaddr_in*/
```

Por otro lado, inet_ntoa() convierte a una cadena que contiene una dirección IP en un entero largo. Por ejemplo:

```
(...)

char *ip;

ip=inet_ntoa(dest.sin_addr);

printf("Address is: %s\n",ip);

(...)
```

Se deberá recordar también que la función inet_addr() devuelve la dirección en formato de Ordenación de Bytes para Redes por lo que no necesitaremos llamar a htonl().

23.8.7. **Funciones Importantes**

En esta sección, (en la cual se nombrarán algunas de las funciones más utilizadas para la programación en C de sockets), se mostrará la sintaxis de la función, las bibliotecas necesarias a incluir para realizar las llamadas, y algunos pequeños comentarios. Además de las que se mencionan aquí, existen muchas funciones más.

23.8.7.1. **socket()**

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain,int type,int protocol);
```

Analicemos los argumentos:

domain. Se podrá establecer como AF_INET (para usar los protocolos ARPA de Internet), o como AF_UNIX (si se desea crear sockets para la comunicación interna del sistema). Estas son las más usadas, pero no las únicas.

type. Aquí se debe especificar la clase de socket que queremos usar (de Flujos o de Datagramas). Las variables que deben aparecer son SOCK_STREAM o SOCK_DGRAM según queramos usar sockets de Flujo o de Datagramas, respectivamente.

protocol. Aquí, simplemente se puede establecer el protocolo a 0.

La función socket() nos devuelve un descriptor de socket, el cual podremos usar luego para llamadas al sistema. Si nos devuelve -1, se ha producido un error (obsérvese que esto puede resultar útil para rutinas de chequeo de errores).

23.8.7.2. **bind()**

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int fd, struct sockaddr *my_addr,int addrlen);
```

Analicemos los argumentos:

fd. Es el descriptor de archivo socket devuelto por la llamada a socket().

my_addr. es un puntero a una estructura sockaddr

addrlen. contiene la longitud de la estructura sockaddr a la cual apunta el puntero my_addr. Se debería establecerla como sizeof(struct sockaddr).

La llamada bind() es usada cuando los puertos locales de nuestra máquina están en nuestros planes (usualmente cuando utilizamos la llamada listen()). Su función esencial es asociar un socket con un puerto (de nuestra máquina). Análogamente socket(), devolverá -1 en caso de error.

Por otro lado podremos permitir que nuestra dirección IP y puerto sean elegidos automáticamente:

```
server.sin_port = 0;  /* bind() will choose a random port*/

server.sin_addr.s_addr = INADDR_ANY; /* puts server's IP automatically */
```

Un aspecto importante sobre los puertos y la llamada bind() es que todos los puertos menores que 1024 son reservados. Se podrá establecer un puerto, siempre que esté entre 1024 y 65535 (y siempre que no estén siendo usados por otros programas).

23.8.7.3. **connect()**

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

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

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

Analicemos los argumentos:

fd. Debería setearse como el archivo descriptor del socket, el cual fue devuelto por la llamada a socket().

serv_addr. Es un puntero a la estructura sockaddr la cual contiene la dirección IP destino y el puerto.

addrlen. Análogamente de lo que pasaba con bind(), este argumento debería establecerse como sizeof(struct sockaddr).

La función connect() es usada para conectarse a un puerto definido en una dirección IP. Devolverá -1 si ocurre algún error.

23.8.7.4. **listen()**

```
#include <sys/types.h>
#include <sys/socket.h>

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

Veamos los argumentos de listen():

fd. Es el archivo descriptor del socket, el cual fue devuelto por la llamada a socket()

backlog. Es el número de conexiones permitidas.

La función listen() se usa si se está esperando conexiones entrantes, lo cual significa, si se quiere, alguien que se quiera conectar a nuestra máquina.

Luego de llamar a listen(), se deberá llamar a accept(), para así aceptar las conexiones entrantes. La secuencia resumida de llamadas al sistema es:

1.socket()

2.bind()

3.listen()

4.accept()

Como todas las funciones descriptas arriba, listen() devolverá -1 en caso de error.

23.8.7.5. **accept()**

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int fd, void *addr, int *addrlen);
```

Veamos los argumentos de la función:

fd. Es el archivo descriptor del socket, el cual fue devuelto por la llamada a listen().

addr. Es un puntero a una estructura sockaddr_in en la cual se pueda determinar qué nodo nos está contactando y desde qué puerto.

addrlen. Es la longitud de la estructura a la que apunta el argumento addr, por lo que conviene establecerlo como sizeof(struct sockaddr_in), antes de que su dirección sea pasada a accept().

Cuando alguien intenta conectarse a nuestra computadora, se debe usar accept() para conseguir la conexión. Es muy fácil de entender: alguien sólo podrá conectarse (asóciese con connect()) a nuestra máquina, si nosotros aceptamos (asóciese con accept()) ;-)

A continuación, Se dará un pequeño ejemplo del uso de accept() para obtener la conexión, ya que esta llamada es un poco diferente de las demás.

```
(...)

sin_size=sizeof(struct sockaddr_in);
/* En la siguiente línea se llama a accept() */
if ((fd2 = accept(fd,(struct sockaddr *)&client,&sin_size))==-1){
printf("accept() error\n");
exit(-1);
}

(...)
```

A este punto se usará la variable fd2 para añadir las llamadas send() y recv().

23.8.7.6. **send()**

```
#include <sys/types.h>
 #include <sys/socket.h>

 int send(int fd,const void *msg,int len,int flags);
```

Y sobre los argumentos de esta llamada:

fd. Es el archivo descriptor del socket, con el cual se desea enviar datos.

msg. Es un puntero apuntando al dato que se quiere enviar.

len. es la longitud del dato que se quiere enviar (en bytes).

flags. deberá ser es establecido a 0.

El propósito de esta función es enviar datos usando sockets de flujo o sockets conectados de datagramas.

Si se desea enviar datos usando sockets no conectados de datagramas debe usarse la llamada sendto(). Al igual que todas las demás llamadas que aquí se vieron, send() devuelve -1 en caso de error, o el número de bytes enviados en caso de éxito.

23.8.7.7. **recv()**

```
 #include <sys/types.h>
 #include <sys/socket.h>

 int recv(int fd, void *buf, int len, unsigned int flags);
```

Veamos los argumentos:

fd. Es el descriptor del socket por el cual se leerán datos.

buf. Es el buffer en el cual se guardará la información a recibir.

len. Es la longitud máxima que podrá tener el buffer.

flags. Por ahora, se deberá establecer como 0.

Al igual de lo que se dijo para send(), esta función es usada con datos en sockets de flujo o sockets conectados de datagramas. Si se deseara enviar, o en este caso, recibir datos usando sockets desconectados de Datagramas, se debe usar la llamada recvfrom(). Análogamente a send(), recv() devuelve el número de bytes leídos en el buffer, o -1 si se produjo un error.

23.8.7.8. **recvfrom()**

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

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

int recvfrom(int fd,void *buf, int len, unsigned int flags
        struct sockaddr *from, int *fromlen);
```

Veamos los argumentos:

fd. Lo mismo que para recv()

buf. Lo mismo que para recv()

len. Lo mismo que para recv()

flags. Lo mismo que para recv()

from. Es un puntero a la estructura sockaddr.

fromlen. Es un puntero a un entero local que debería ser inicializado a sizeof(struct sockaddr).

Análogamente a lo que pasaba con recv(), recvfrom() devuelve el número de bytes recibidos, o -1 en caso de error.

23.8.7.9. **close()**

```
#include <unistd.h>

close(fd);
```

La función close() es usada para cerrar la conexión de nuestro descriptor de socket. Si llamamos a close() no se podrá escribir o leer usando ese socket, y se alguien trata de hacerlo recibirá un mensaje de error.

23.8.7.10. **shutdown()**

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

int shutdown(int fd, int how);
```

Veamos los argumentos:

fd. Es el archivo descriptor del socket al que queremos aplicar esta llamada.

how. Sólo se podrá establecer uno de estos nombres:

0. Prohibido recibir.
1. Prohibido enviar.
2. Prohibido recibir y enviar.

Es lo mismo llamar a close() que establecer how a 2. shutdown() devolverá 0 si todo ocurre bien, o -1 en caso de error.

23.8.7.11. **gethostname()**

```
#include <unistd.h>

int gethostname(char *hostname, size_t size);
```

Veamos de qué se tratan los argumentos:

hostname. Es un puntero a un vector que contiene el nombre del nodo actual.

size. La longitud del vector que contiene al nombre del nodo (en bytes).

La función gethostname() es usada para adquirir el nombre de la máquina local.

23.8.8. **Algunas palabras sobre DNS**

DNS son las siglas de ``Domain Name Service [9]'' y, básicamente es usado para conseguir direcciones IP. Por ejemplo, necesito saber la dirección IP del servidor queima.ptlink.net y usando el DNS puedo obtener la dirección IP 212.13.37.13.

Esto es importante en la medida de que las funciones que ya vimos (como bind() y connect()) son capaces de trabajar con direcciones IP.

Para mostrar cómo se puede obtener la dirección IP de un servidor, por ejemplo de queima.ptlink.net, utilizando C, el autor ha realizado un pequeño

ejemplo:

```
 /* <---- EL CÓDIGO FUENTE EMPIEZA AQUÍ ----> */

 #include <stdio.h>
 #include <netdb.h>   /* Este es el archivo de cabecera necesitado por gethostbyname() */
 #include <sys/types.h>
 #include <sys/socket.h>
 #include <netinet/in.h>

int main(int argc, char *argv[])
 {

   struct hostent *he;

   if (argc!=2) {
     printf("Usage: %s <hostname>\n",argv[0]);
     exit(-1);
   }

   if ((he=gethostbyname(argv[1]))==NULL) {
     printf("gethostbyname() error\n");
     exit(-1);
   }

   printf("Hostname : %s\n",he->h_name);
     /* muestra el nombre del nodo */
   printf("IP Address: %s\n",
         inet_ntoa(*((struct in_addr *)he->h_addr)));
     /* muestra la dirección IP */

 }

 /* <---- CÓDIGO FUENTE TERMINA AQUÍ ---> */
```

23.8.9. **Un ejemplo de Servidor de Flujos**

En esta sección, se describirá un bonito ejemplo de un servidor de flujos. El código fuente tiene muchos comentarios para que así, al leerlo, no nos queden dudas.

Empecemos:

```
 /* <---- El CÓDIGO FUENTE COMIENZA AQUÍ ----> */

 /* Estos son los ficheros de cabecera usuales */
 #include <stdio.h>
 #include <sys/types.h>
 #include <sys/socket.h>
 #include <netinet/in.h>

 #define PORT 3550 /* El puerto que será abierto */
 #define BACKLOG 2 /* El número de conexiones permitidas */

 main()
```

```
{

   int fd, fd2; /* los archivos descriptores */

   struct sockaddr_in server;
   /* para la información de la dirección del servidor */

   struct sockaddr_in client;
   /* para la información de la dirección del cliente */

   int sin_size;

   /* A continuación la llamada a socket() */
   if ((fd=socket(AF_INET, SOCK_STREAM, 0)) == -1 ) {
      printf("error en socket()\n");
      exit(-1);
   }

   server.sin_family = AF_INET;

   server.sin_port = htons(PORT);
   /* ¿Recuerdas a htons() de la sección "Conversiones"? =) */

   server.sin_addr.s_addr = INADDR_ANY;
   /* INADDR_ANY coloca nuestra dirección IP automáticamente */

   bzero(&(server.sin_zero),8);
   /* escribimos ceros en el reto de la estructura */


   /* A continuación la llamada a bind() */
   if(bind(fd,(struct sockaddr*)&server,
           sizeof(struct sockaddr))==-1) {
      printf("error en bind() \n");
      exit(-1);
   }

   if(listen(fd,BACKLOG) == -1) {  /* llamada a listen() */
      printf("error en listen()\n");
      exit(-1);
   }

   while(1) {
      sin_size=sizeof(struct sockaddr_in);
      /* A continuación la llamada a accept() */
      if ((fd2 = accept(fd,(struct sockaddr *)&client,
                        &sin_size))==-1) {
         printf("error en accept()\n");
         exit(-1);
      }

      printf("You got a connection from %s\n",
             inet_ntoa(client.sin_addr) );
      /* que mostrará la IP del cliente */

      send(fd2,"Bienvenido a mi servidor.\n",22,0);
      /* que enviará el mensaje de bienvenida al cliente */

      close(fd2); /* cierraclose a fd2 */
   }
}

/* <---- EL CÓDIGO FUENTE TERMINA AQUÍ ----> */
```

23.8.10. **Un ejemplo de Cliente de Flujos**

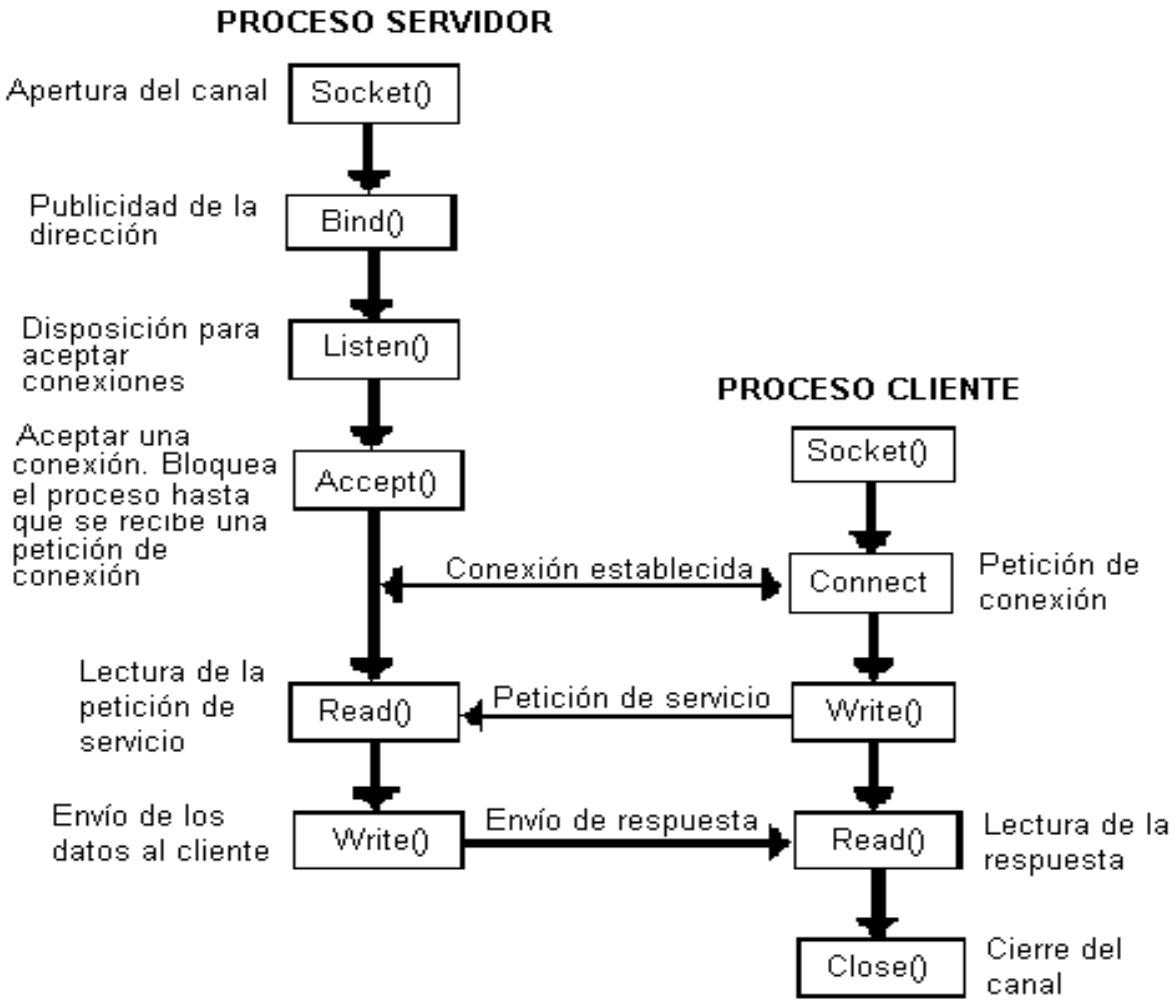
Todo será análogo a lo visto en la sección anterior.

```
/* <---- El CÓDIGO FUENTE COMIENZA AQUÍ ----> */

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
/* netbd.h es necesitada por la estructura hostent ;-) */

#define PORT 3550
/* El Puerto Abierto del nodo remoto */

#define MAXDATASIZE 100
/* El número máximo de datos en bytes */

int main(int argc, char *argv[])
{
  int fd, numbytes;
  /* archivos descriptores */

  char buf[MAXDATASIZE];
  /* en donde es almacenará el texto recibido */

  struct hostent *he;
  /* estructura que recibirá información sobre el nodo remoto */

  struct sockaddr_in server;
  /* información sobre la dirección del servidor */

  if (argc !=2) {
    /* esto es porque nuestro programa sólo necesitará un
    argumento, (la IP) */
    printf("Usage: %s <IP Address>\n",argv[0]);
    exit(-1);
  }

  if ((he=gethostbyname(argv[1]))==NULL){
    /* llamada a gethostbyname() */
    printf("gethostbyname() error\n");
    exit(-1);
  }

  if ((fd=socket(AF_INET, SOCK_STREAM, 0))==-1){
    /* llamada a socket() */
    printf("socket() error\n");
    exit(-1);
  }

  server.sin_family = AF_INET;
  server.sin_port = htons(PORT);
  /* htons() es necesaria nuevamente ;-o */
  server.sin_addr = *((struct in_addr *)he->h_addr);
  /*he->h_addr pasa la información de ``*he'' a "h_addr" */
  bzero(&(server.sin_zero),8);

  if(connect(fd, (struct sockaddr *)&server,
    sizeof(struct sockaddr))==-1){
    /* llamada a connect() */
    printf("connect() error\n");
```

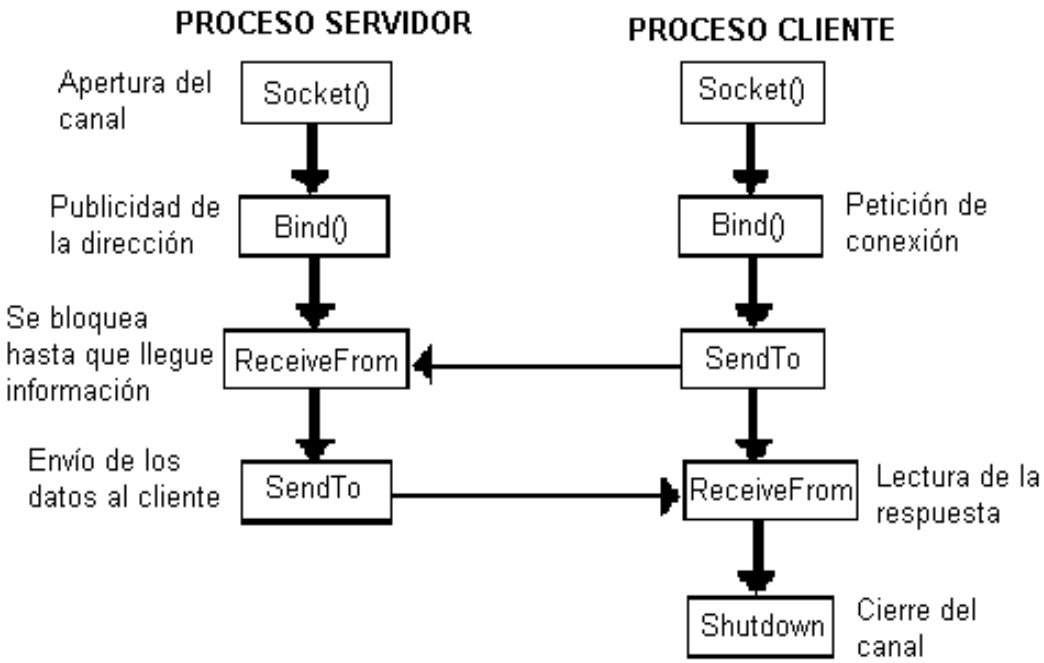
```
    exit(-1);
  }

  if ((numbytes=recv(fd,buf,MAXDATASIZE,0)) == -1){
    /* llamada a recv() */
    printf("recv() error\n");
    exit(-1);
  }

  buf[numbytes]='\0';

  printf("Server Message: %s\n",buf);
  /* muestra el mensaje de bienvenida del servidor =) */

  close(fd);   /* cerramos fd =) */

}

/* <---- EL CÓDIGO FUENTE TERMINA AQUÍ ----> */
```

En resumen podemos esquematizar la comunicación entre sockets, en modelo cliente/servidor de la siguiente manera:

Orientado a conexión:



Orientado a datagramas, o sea sin conexión:



23.9. - **INTERCAMBIO DE MENSAJES**.

La implementación de los mecanismos de comunicación y sincronización a través del intercambio de mensajes se da por el envío (**send**) y recepción (**receive**) de mensajes, en vez de la lectura o escritura de una variable compartida.

La comunicación ocurre porque una tarea al recibir un mensaje obtiene datos enviados por otra tarea, en tanto que la sincronización se da porque un mensaje sólo puede ser recibido después de haber sido enviado, lo que restringe el orden en el cual estos eventos deben ocurrir.

Ejemplo: **Send** mensaje **to** destino
Receive mensaje **from** origen

Asumimos como hipótesis que luego de recibido el mensaje, éste se destruye.

Estudiaremos el intercambio de mensajes por :
- Tipos de sincronización.
- Especificación de los canales de comunicación.
- Tipos de mensajes.
- Tratamiento y recuperación de errores.

Para este fin utilizaremos las siguientes primitivas:

Send sincrónico : la tarea emisora queda bloqueada hasta que el mensaje sea recibido por la tarea destino.

Send asincrónico : la tarea emisora luego de emitir el mensaje continua con su procesamiento. De esta forma la concurrencia entre las tareas que se comunican es maximizada.

Send condicional: la tarea emisora luego de emitir el mensaje continúa con su procesamiento, pero el mensaje solamente será recibido por la tarea receptora si es que ésta se encuentra bloqueada en el momento de emisión del mensaje. En caso contrario la tarea emisora recibe un código de retorno que le indica que el mensaje no ha llegado a destino. La utilización de esta primitiva implica que la tarea receptora utilice un receive bloqueante.

Receive (incondicional o bloqueante): la tarea receptora queda bloqueada hasta recibir un mensaje.

Receive condicional (polling): la tarea receptora pregunta a los canales de comunicación si existe un mensaje. En caso negativo, la primitiva receive devuelve un código de retorno que indica que no hay mensajes en ese canal de comunicación. Esta capacidad de consulta (polling) permite que la tarea receptora controle el nivel de concurrencia.

23.9.1. - **TIPOS DE SINCRONIZACION.**

Las primitivas de comunicación pueden clasificarse en sincrónicas y asincrónicas según bloqueen o no a las tareas que ejecutan la emisión o recepción de un mensaje.

23.9.1.1. - **Comunicación Sincrónica.**

Los mecanismos de intercambio de mensajes basados en el uso de primitivas de comunicación y sincronización sincrónica se clasifican en tres categorías discutidas a continuación:

a) Rendez-Vous

La tarea emisora es bloqueada hasta que la tarea receptora esté lista para recibir el mensaje. Cuando la tarea receptora ejecuta un receive, si no se encuentra disponible un mensaje entonces queda bloqueada hasta la llegada del mismo.

Una vez efectuado el intercambio de mensajes ambas tareas continúan su ejecución en forma concurrente.

```
tarea Productor;
        begin
        <Producir un msg>;
        Send msg to Consumidor
        end;
tarea Consumidor;
        begin
        Receive msg from Productor;
        <Consumir msg>
        end;
```

Una abstracción utilizando una implementación con semáforos sería:

```
V(x)
P(y)
SEND

P(x)
V(y)
RECEIVE
```

Obsérvese que en este caso las velocidades de producción y consumo de mensajes son equivalentes debido a la sincronización explícita de las tareas.

b) Rendez-Vous extendido

Como su nombre lo indica, esta forma de interacción entre tareas es una extensión del mecanismo anterior, con la diferencia de que la tarea receptora solamente envía una respuesta a la tarea emisora después de la ejecución de un cuerpo de comandos que operan sobre el mensaje recibido. Esta respuesta puede poseer parámetros que contengan el resultado de los cálculos efectuados por la tarea receptora. Obsérvese también que la tarea emisora queda bloqueada hasta el término del cuerpo de comandos en la tarea receptora, que es cuando el rendez-vous se completa.

Con semáforos se resume en:

```
V(x)
SEND
P(y)

P(x)
RECEIVE
If ok Then V(y)
```

c) Rendez-Vous asimétrico

Es una variante del anterior. Aquí solamente el emisor (cliente) nombra a la tarea receptora (server) (ADA).

La primitiva receive es reemplazada por el comando **accept**. Ambas tareas quedan en rendez-vous hasta que se ejecute todo el cuerpo del comando accept. El accept no nombra al emisor pues la comunicación ya está establecida.

Los parámetros (ya que es como llamar a un monitor) pueden ser de input, de output o de input / output.

```
Tarea T1;
        begin
        Send x to T2;
        end;
```

```
Tarea T2;
        begin
        Accept Send (x);
        y := x;
        end;
```

23.9.1.2. - **Comunicación Asincrónica.**

Las primitivas de comunicación asincrónica se caracterizan por no bloquear a las tareas que las ejecutan.

Una tarea emisora al realizar un send asincrónico continua su ejecución sin bloquearse.

En el caso de receive asincrónico el receptor continúa su ejecución aunque no haya llegado nada. Depende de la implementación si los mensajes siguientes serán o no tomados en cuenta (Ej. Spool de VM).

Este tipo de envío no bloqueante permite la múltiple difusión de mensajes (**broadcasting**), o sea, que un mismo mensaje es enviado a varios destinatarios simultáneamente.

La principal ventaja de la comunicación asincrónica es que maximiza el paralelismo en la ejecución de las tareas.

23.9.1.3. - **Comunicación Semi-Sincrónica.**

Una variante a los esquemas de comunicación sincrónica y asincrónica es la comunicación semi-sincrónica que usa **send** no bloqueantes y **receive** bloqueantes. Sin embargo en esta implementación se corre el peligro de tener largas colas de mensajes.

23.9.2. - **Especificaciones de los Canales de Comunicación**

Definir Origen y Destino de los mensajes define un canal de comunicación.

Existen dos tipos de especificaciones, a saber:

a) Comunicación directa

Cada proceso que desea enviar/recibir un mensaje debe nombrar explícitamente el receptor/emisor del mensaje en cuestión.

Un proceso para comunicarse con otro sólo debe conocer la identidad del otro proceso.

Cada enlace de comunicación de este tipo vincula a dos procesos y es bidireccional.

El lenguaje CSP utiliza los comandos:

- de entrada P1 ? A
- de salida P2 ! B

siendo P1 y P2 las tareas.

Este esquema se usa cuando la salida de una tarea es entrada de otra (Pipeline (Unix)).

Sin embargo, este mecanismo no resuelve el problema del server que atiende a más de un cliente, o el cliente que llame a más de un server.

b) Comunicación indirecta

Este mecanismo soluciona el problema anterior mediante el uso de nombres globales (Mailboxes).

Los procesos envían y reciben mensajes desde mailboxes. Un mailbox puede ser visto como un objeto en el que los mensajes son depositados y retirados por los procesos. Cada mailbox tiene una identificación que lo determina unívocamente.

Dos procesos pueden comunicarse sólo si comparten un mailbox.

La comunicación entre procesos puede vincular a más de dos procesos y dos procesos se pueden comunicar a través de varios mailboxes. La comunicación entre procesos puede ser unidireccional o bidireccional.

Esta mecánica funciona cuando, por ejemplo, se comparte memoria, pero no funciona para sistemas distribuidos a menos que se le atribuya al mailbox un mecanismo de red. Esto implica que se deben colocar estos nombres globales en todos los puntos de la red donde ese mensaje podría ser usado.

Un caso especial es aquel que se establece cuando una única tarea está autorizada a hacer receives en un mailbox pero varias tareas pueden hacer sends a dicho mailbox. Este tipo de mailbox se denomina **puerta** (port). Este esquema es más fácil de implementar porque todos los receives que pueden referenciar a una misma puerta se hallan dentro de una única tarea.

23.9.3. - **Direccionamiento**

En resumen los direccionamientos pueden ser:

Directo: Dos tareas sobre nodos distintos mantienen comunicación directa entre sí.

Indirecto: La comunicación se mantiene a través de una estructura de datos compartida (llamada mailbox)
Ventaja: desacopla al emisor y al receptor, permitiendo mayor flexibilidad en el uso de los mensajes.
Desventaja: Centraliza.

La utilización del mailbox es posible en las modalidades:

- Uno a uno, también llamado enlace privado.
- Muchos a uno, que corresponde al modelo cliente/servidor.
- Uno a muchos, también llamado broadcast.
- Muchos a muchos, que se puede visualizar como la posibilidad de tener muchos servidores

Por lo general los mailbox manejan disciplinas de colas en modalidad FIFO, pero también es posible el manejo por prioridades.

23.9.4. - **Ejemplo de Modelos Clásicos con mailbox**

23.9.4.1. - **Exclusión Mutua**

Se utilizan **receive bloqueantes y send no bloqueantes**.
El **mailbox** se lo utiliza como contenedor de un **token.**

```
/* programa exclusion-mutua */
int  n=   /* número de procesos */

void p(int i)
{
        mensaje msj;
        while (cierto)
        {
                receive (exmut, msj);  /*si el mailbox está vacío el proceso se detiene */
                /* sección crítica */
                send (exmut, msj);
                /* resto */
        }
}

void main ()
{
        crear-mailbox (exmut);
        send (exmut, token);
        parbegin (p1, p2, p3, ..., pn);
}
```

23.9.4.2. - **Productor/Consumidor**

Se utilizan **receive bloqueantes y send no bloqueantes**.
Se utilizan dos buzones, puede_consumir y puede_producir

```
Capacidad =     /* capacidad del buffer */;
Int i;

Void productor()
{
        mensaje msjp;
        while (cierto);
        {
                receive (puede_producir, msjp);
                msjp = producir();
                send (puede-consumir, msjp);
        }
}
```

```
void consumidor()
{
        mensaje msjc;
        while (cierto)
        {
                receive (puede_consumir, msjc);
                consumir(msjc);
                send (puede_producir, token);
        }
}

void main()
{
        crear_mailbox (puede_producir);
        crear_mailbox (puede_consumir);
        for (int i = 1, i <= capacidad; i++)
                send (puede_producir, token);
                send (puede_consumir, null);
        parbegin (productor, consumidor) parend;
}
```

23.9.4.3. - **Lectores/Escritores**

Se utilizan **3 mailbox pedir_lectura pedir-escritura** y **terminado.**
Además de los procesos lector y escritor se utiliza uno auxiliar llamado controlador que actúa según el valor de una variable **cont, según lo siguiente:**

Cont > 0 no hay escritores esperando
Cont = 0 pendientes escrituras, esperar terminado
Cont < 0 escritor en espera

Cantidad máxima de lectores = 100

```
void lector(int i)
{
        mensaje msjl;
        while (cierto)
        {
                msjl = i;
                send (pedir-lectura, msjl);
                receive (buzón[i], msjl);
                LEER;
                msjl = i;
                send (terminado, msjl);
        }
}


void escritor(int j)
{
        mensaje msje;
        while (cierto)
        {
                msje=j;
                send (pedir_escritura, msje);
                receive (buzón[j], msje);
                ESCRIBIR;
                msje = j;
                send (terminado, msje)
        }
}
```

```
void controlador()
{
        while(cierto)
        {
                if cont > 0
                {
                        if (!vacío (terminado))
                        {
                                receive (terminado, msj);
                                cont++
                        }
                        else
                        if (!vacío (pedir_escritura))
                        {
                                receive (pedir_escritura, msj);
                                escritor_id = msj-id;
                                cont  = cont – 100;
                        }
                        else
                        if (!vacío (pedir_lectura))
                        {
                                receive (pedir_lectura, msj));
                                cont--;
                                send(msj_id, “OK”);
                        }
                }
                if (cont == 0)
                {
                        send (escritor_id, “OK”);
                        receive (terminado, msj);
                        cont = 100;
                }
                while (cont < 0)
                {
                        receive (terminado, msj);
                        cont ++;
                }
        }
}
```

23.9.5. - **Tipos de Mensajes**.

Los mensajes enviados por un proceso pueden ser de tres tipos: tamaño fijo, tamaño variable o mensajes tipificados.

Si solamente se pueden enviar mensajes de tamaño fijo, la implementación física es bastante sencilla. Esta restricción, sin embargo, hace a la tarea de programación más difícil.

Por otro lado, los mensajes de longitud variable requieren una implementación física más compleja pero facilitan la programación.

El último caso consiste en asociar a cada mailbox un tipo de mensaje, por lo que este esquema es sólo aplicable cuando se usa comunicación indirecta. Esta asociación permite detectar, en tiempo de compilación, la posibilidad del intercambio de mensajes de diferente tipo brindando un alto grado de seguridad sin provocar sobrecarga en tiempo de ejecución.

23.9.6. - **Tratamiento y Recuperación de Errores.**

Existen dos tipos de fallas que merecen un tratamiento adecuado para garantizar, que cuando ocurra alguna de ellas, sea posible preservar la integridad del sistema.

El primer tipo de falla se denomina falla lógica y se da cuando se produce un deadlock o cuando una tarea envía o espera recibir un mensaje de otra tarea que ya no existe (destrucción prematura de tareas).

El otro tipo de falla se denomina falla física y se da cuando trabajamos en un sistema distribuido y la falla se produce en el hardware de comunicación.

Para ambos tipos de errores es necesario contar con un mecanismo de time-out para evitar el problema del bloqueo perpetuo.