Sistemas distribuidos

Esteban Ruiz
eruiz0@fceia.unr.edu.ar

Sistemas Operativos II - 2021
Departamento de Ciencias de la Computación
FCEIA-UNR

Índice

1. Conceptos generales

1.1. Introducción

Por ahora hablaremos de sistemas distribuidos en general y no estrictamente de sistemas operativos distribuidos. LLamaremos sistemas distribuidos a un sistema conformado por un gran número de CPUs interconectadas por una red de alta velocidad.

Discusión: ¿cuántas son "un gran número de CPUs"? ¿cuáles son redes de alta velocidad? La definición puede parecer un poco ambigua o vaga (y en algunos casos la distinción no es para nada clara).

En general usamos sistemas distribuidos para distinguirlo de sistemas centralizados consistentes en una única CPU, memoria, periféricos y alguna terminal. Aquí quizás convenga considerar que nos preocuparemos por el software (más que por el hardware) que no siempre podrá confiar en tener una memoria compartida ni en que los mensajes llegarán siempre y en orden. También habrá que tener en cuenta que la red puede caerse total o parcialmente y lo mismo puede suceder con algunos nodos (o CPUs) que incluso pueden volver a aparecer en un tiempo.

1.2. ¿Por qué se usan sistemas distribuidos? Ventajas

Actualmente las ventajas son muchas:

- Precio/rendimiento: el poder de cómputo de 200 procesadores económicos interconectados puede ser mucho mayor que el del mejor procesador disponible.
- Distribución necesaria o eficiente: sistema de reserva de vuelos, nodos cercanos brindarán una respuesta rápida.
- Seguridad (alta disponibilidad): tener muchas copias de datos y servicios permite brindar más confiabilidad en el caso de que un nodo deje de funcionar.
- Crecimiento exponencial: un sistema que inicialmente atenderá poco pedidos puede comenzar a trabajar con un pocos nodos y a medida que el negocio crece pueden ir agregándose más nodos para poder seguir brindando la misma calidad de servicio.

1.3. Problemas y desventajas

Los sistemas distribuidos añaden complejidad, principalmente:

- Software: el acceso concurrente y la división de tareas ya son conceptos que aparecen en sistemas paralelos o concurrentes.
- Como no hay memoria compartida y se depende de una red que tal vez no sea tan estable los mensajes y nodos pueden perderse y/o llegar desordenados.
- Seguridad: tener los datos y servicios disponibiles en una red involucra nuevos problemas de seguridad.

2. Transparencia

De acuerdo al tipo de sistema puede ser necesario brindar una o más formas de transparencia. Lo óptimo sería combinarlas todas.

- Ubicación: el usuario no puede decir dónde están ubicados los recursos
- Migración: los recursos pueden moverse sin cambiar de nombre
- Replicación: el usuario no puede saber cuántas copias hay de un recurso
- Concurrencia: los recursos pueden compartirse automáticamente con muchos usuarios
- Paralelismo: las acciones pueden realizarse en paralelo sin que el usuario lo sepa

3. Algoritmos distribuidos

En general un algoritmo en un sistema distribuido evitará ser centralizado si es posible: si todos los nodos deben comunicarse con un cierto nodo "distinguido" ese nodo será un punto de falla único: si deja de responder entonces todo el sistema fallará. Además es muy posible que el acceso a ese nodo en sistemas grandes se transforme en un gran cuello de botella.

Los algoritmos distribuidos deberían funcionar razonablemente bien si algunos nodos no están disponibiles (temporaria o definitivamente) y si los mensajes llegan en cualquier orden.

4. Acceso exclusivo

Veremos algoritmos que permiten tomar un lock para acceder a algún recurso compartido en un sistema distribuido.

4.1. Un algoritmo centralizado

Ya vimos que no es lo óptimo tener un único nodo, pero esto nos servirá de base para presentar los siguientes algoritmos y comparar.

En este caso, habrá un nodo coordinador y el mecanismo para adquirir un lock es relativamente simple: un nodo que necesite acceder a un recurso protegido por un lock le enviará un mensaje al cordinador que permitirá el acceso si no hay otro proceso en la sección crítica. Cuando el proceso termina de utilizar el lock enviará otro mensaje al coordinador que le permitirá saber que el lock ha sido liberado.

¿Qué ocurre si otro proceso necesita acceder a la sección crítica cuando el lock está siendo utilizado? El coordinador recibirá el pedido y aquí se pueden implementar dos soluciones

1. El coordinador no responde pero recuerda el pedido para responderlo cuando la sección crítica esté libre.
2. El coordinador responde "acceso denegado"

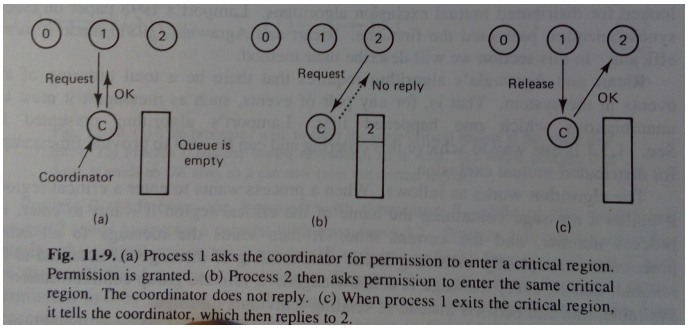


Fig. 11-9. (a) Process 1 asks the coordinator for permission to enter a critical region. Permission is granted. (b) Process 2 then asks permission to enter the same critical region. The coordinator does not reply. (c) When process 1 exits the critical region, it tells the coordinator, which then replies to 2.

Ambas soluciones parecen sencillas pero cada una tiene su desventaja: Si se utiliza la primera y el coordinador queda desconectado o se apaga, entonces el proceso que pidió permiso quedará esperando por siempre.

Si se utiliza la segunda se obliga al proceso que quiere acceder a la sección crítica a reintentar periódicamente, obligándolo a una espera ocupada (busy waiting) y además a generar mayor tráfico de red.

No está de más volver a mencionar que este algoritmo sufre los problemas de los algoritmo centralizado: el coordinador puede volverse un cuello de botella y en el caso de que el coordinador falle todo el sistema fallará.

4.2. Un algoritmo distribuido

El primer algoritmo distribuido fue presentado por Lamport. Veremos una versión más eficiente propuesta por Ricart y Agrawala en 1981. Este algoritmo requiere que todos los eventos en el sistema tengan un orden total (basado en un timestamp): dados dos mensajes y/o eventos debe poder saberse de forma no ambigua cuál de los dos se considera que ocurrió primero (otro algoritmo de Lamport permite garantizar eso). También se asume que la entrega de paquetes es confiable (i.e.: hay un acknowledge o "acuse de recibo").

Ahora para ingresar a una sección crítica un proceso construye un paquete con el pedido que incluye: el nombre de la región crítica, su número de proceso y la hora actual. Este mensaje es enviado a todos los miembros del grupo y se espera a que todos los miembro del grupo respondan OK.

Cuando otro proceso recibe el pedido pueden darse tres casos:

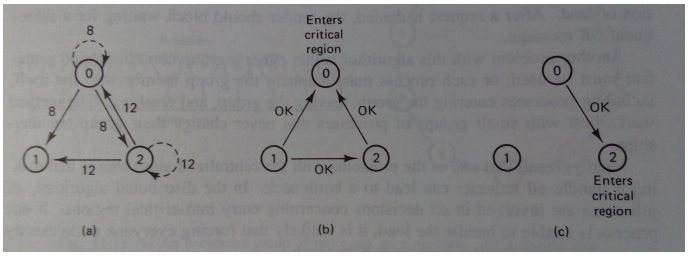
- No tiene intenciones de ingresar en esa región crítica, entonces responde OK.
- Está en la región crítica: encola el pedido para responderlo cuando termine de usar esa región.
- Necesita entrar en la región crítica pero todavía no recibió todas las autorizaciones. En este caso compara el timestamp que le llega con el timestamp que envió en su pedido. El más bajo gana: si el mensaje entrante tiene el timestamp más bajo entonces responde OK (lo que eventualmente habilitará al otro proceso). En caso contrario encola la petición (este proceso entrará antes a la sección crítica).

Cuando un proceso termina de usar la región crítica envía OK a todos los procesos que estaban esperando por ella.

Desventaja: ahora se necesitan $2(n-1)$ paquetes si hay n nodos en la red

Ventaja: No hay un servidor o coordinador, no hay único punto de falla

Desventaja: ¡¡ahora hay n puntos de falla!!



Se puede mejorar un poco: no es necesario esperar que todos respondan, se puede cambiar un poco para que con la respuesta de una mayoría de los procesos ya se tenga suficiente concenso para continuar.

Más desventajas: requiere que se conozca quienes son todos los miembros del grupo, si no se recibe respuesta puede ser que el proceso esté en una sección crítica pero también que el nodo se haya caído.

Nuevamente, se pueden hacer más mejoras, pero de cualquier manera el algoritmo será más lento, más complicado, más caro y menos robusto que el algoritmo centralizado.

En conclusión: si bien existen algoritmos distribuidos para este problema muchas veces tener un proceso coordinador simplifica mucho la cosa. Así, para muchos casos se utiliza un coordinador y se aplican algoritmos de elección de lider para reelegir un nuevo coordinador en caso de que un coordinador deje de estar disponibile.

4.3. Algoritmo basado en anillo

Otra alternativa es utilizar una topología lógica basada en token ring: los procesos (o nodos) estarán ordenados lógicamente en un anillo y un token (un paquete especial) circulará por el anillo: sólo el proceso que tiene el token puede acceder a la sección crítica (sencillamente retendrá el token hasta salir de la sección crítica). Este algoritmo tiene varias ventajas pero también tiene todas las ventajas de una arquitectura basada en anillo.

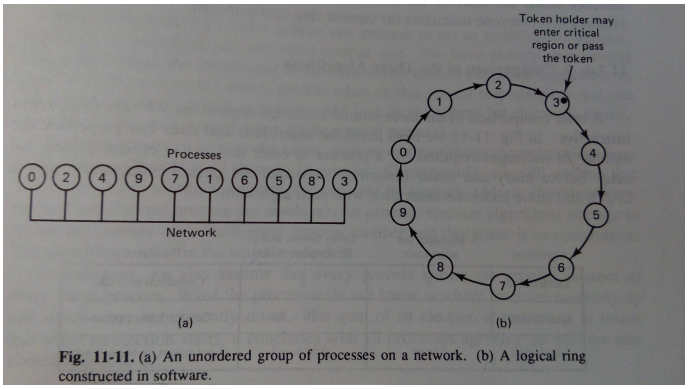


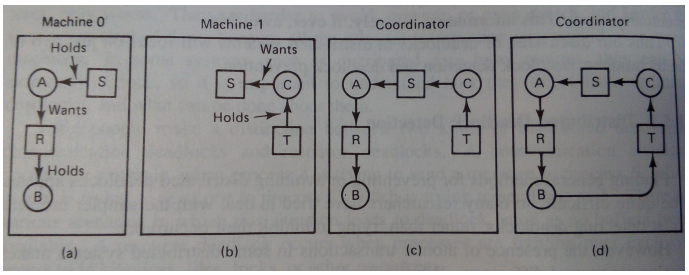
Fig. 11-11. (a) An unordered group of processes on a network. (b) A logical ring constructed in software.

5. Detección de deadlock

Si tenemos locks está la posibilidad de que existan deadlocks, por tanto está bueno poder detectarlos (i.e.: con la intención de terminar un proceso si se detecta deadlock).

5.1. Algoritmo centralizado: intento

Utilizando un proceso coordinador podríamos hacer que el coordinador lleve el estado del grafo pedido-tomado: cada proceso puede avisarle al coordinador cada vez que se necesita agregar o quitar un arco. Si el coordinador detecta que se originó un ciclo entonces avisa el proceso para que aborte.



Lamentablemente dependiendo del orden de los mensajes puede darse un escenario en el que se concluye falsamente que existe un deadlock. Supongamos el estado de la figura: (a) y (b) representan el estado inicial de dos máquinas (máquina 0 y máquina 1 respectivamente) y (c) representa la visión que tiene el coordinador. Ahora si B libera a R y luego necesita acceder a T se originarán dos mensajes: la máquina 0 avisará que se liberó R al coordinador y la máquina 1 avisará al coordinador que se necesita agregar un arco: desde B hacia T. Si este segundo mensaje (desde la máquina 1) llega al coordinador antes que el primer mensaje desde la máquina 0 entonces el coordinador concluirá que hay un deadlock al detectar un ciclo (que en realidad no ocurrió, sólo que el coordinador durante un lapso de tiempo puede tener información inconsistente).

Esto se puede mejorar utilizando timestamps como en el algoritmo presentado anteriormente, pero hace que sea más costoso. Además existen otras situaciones en donde eleiminar el falso deadlock es mucho más difícil.

5.2. Algoritmo distribuido

Hay varias propuestas pero una amplia proporción de los algoritmos publicados son incorrectos. Un algoritmo correcto típico es el de Candy-Misar-Haas (1983).

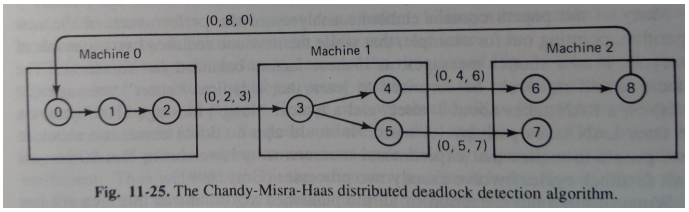


Fig. 11-25. The Chandy-Misra-Haas distributed deadlock detection algorithm.

La idea básica (y muy resumida) es enviar un mensaje que intenta recorrer dinámicamente y en forma distribuida el gráfico de pedidos/tomados: si un proceso A debe esperar por recursos se lo indica a los procesos que los poseen. Al recibir ese paquete un proceso B entonces ese proceso verá si está esperando por un recurso, en ese caso enviará otro paquete (mencionando el proceso original) a todos los procesos que tienen recursos que B espera. Los procesos receptores de este segundo paquete harán lo mismo. Si hay un ciclo eventualmente alguno de los procesos enviará a A un paquete. En ese momento A detectará que hay un deadlock.

Referencias

[1] *A. Tanenbaum, Modern Operating Systems*, 1º ed., Prentice Hall