

Sistemas Distribuidos

1. Introducción

2. La Comunicación

3. Sistemas Operativos Distribuidos

☞ 4. Sincronización y Coordinación

1. Sincronización de Relojes

- Relojes físicos
- Relojes lógicos

2. Coordinación

- Exclusión mutua
- Elección de coordinador
- Consenso distribuido

En este capítulo se presentan algunos conceptos y algoritmos relacionados con la medida del tiempo y la coordinación en el tratamiento de los eventos o sucesos que se producen en un sistema distribuido.

Comenzaremos por examinar la noción de **tiempo** y veremos cómo se pueden sincronizar los relojes de distintos ordenadores, de tal forma que todos tengan la misma medida de cuándo se ha producido un cierto evento. Como veremos, muchas veces no es necesario que todos los equipos del sistema tengan registrada la hora exacta en que se produce cada evento, sino que simplemente es suficiente con tener los eventos ordenados en el tiempo.

La segunda parte del capítulo está dedicado a la **coordinación de procesos**, entendiendo por coordinación el hecho de ponerse de acuerdo entre varios procesos para llevar a cabo alguna acción. Esta acción puede ser el conseguir el derecho a entrar en una región crítica en exclusión mutua, o la elección de un proceso, entre un grupo de procesos, que actúe como coordinador de las actividades del resto de los procesos del grupo. En último lugar comentaremos el consenso distribuido, con algoritmos de coordinación tolerantes a fallos, viendo, como ejemplo, el problema de los generales bizantinos.

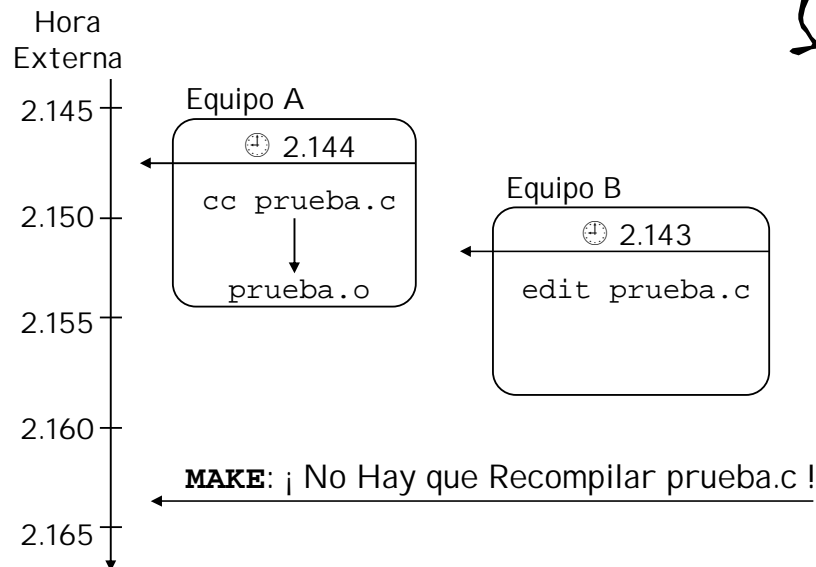
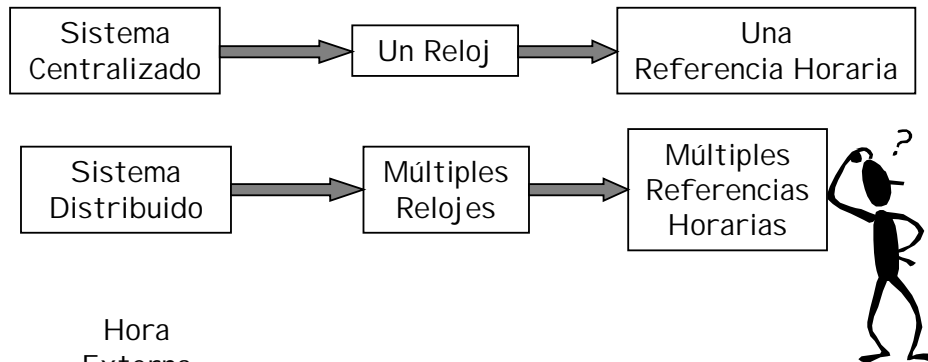
Sincronización de Relojes

Hay que saber en qué momento se produce cada evento en un ordenador

Los Procesos Necesitan Sincronizarse

Sistemas de Tiempo Real

Propósito General: *make*, Kerberos, *backup*, ...



La medida del tiempo es una cuestión muy importante en los sistemas distribuidos. En los sistemas centralizados también lo es, pero no se le da importancia debido a que no es ningún problema conseguir que todos los procesos tengan una misma referencia: el reloj compartido. En cambio en los sistemas distribuidos cada ordenador tiene su propio reloj y, como veremos, no resulta fácil sincronizarlos.

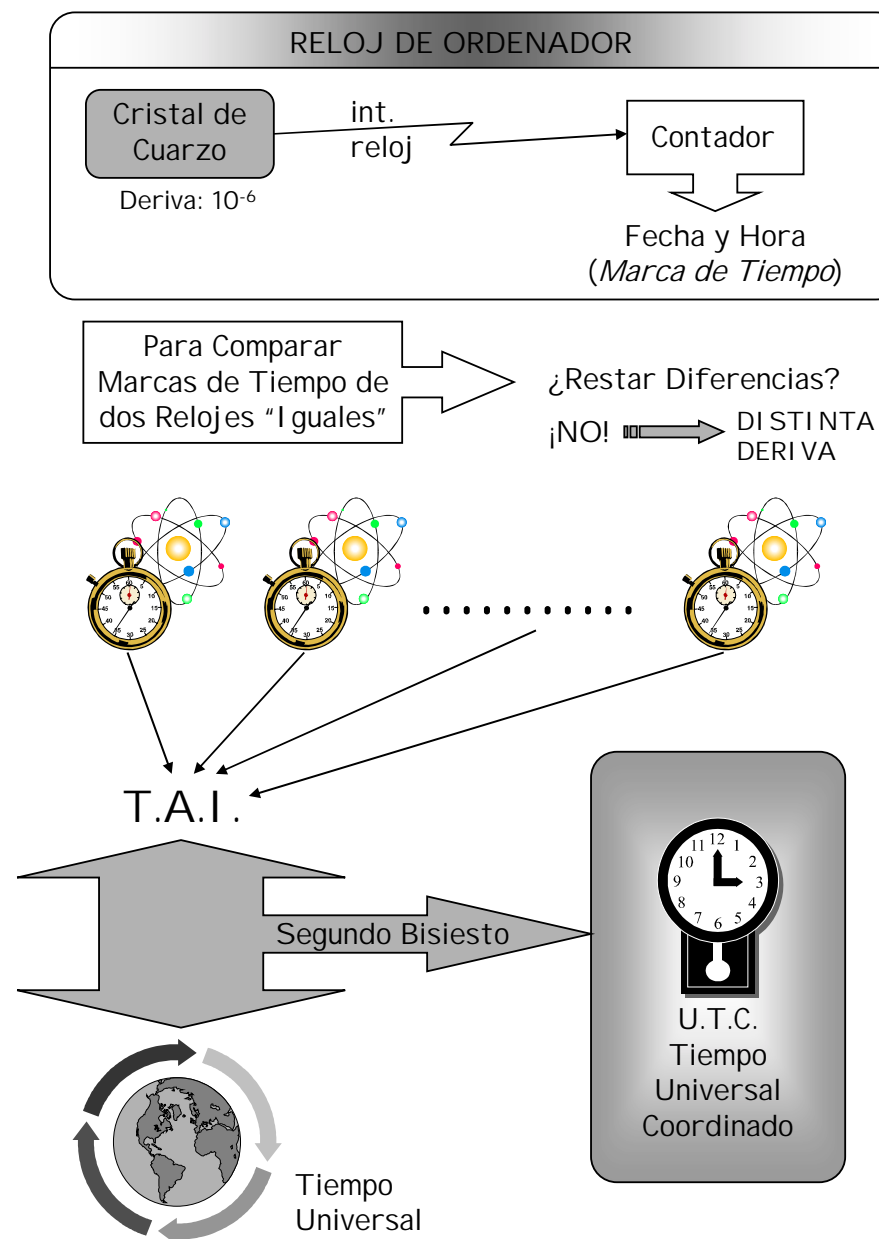
Como hemos dicho, la medida del tiempo es muy importante, pues resulta normal tener que saber a qué hora del día han sucedido los distintos eventos que se producen en un ordenador. La precisión requerida en cada caso varía, siendo los sistemas de tiempo real los que posiblemente requieren una mayor precisión. No obstante, incluso en sistemas de propósito general se hace necesario el disponer de una sincronía horaria.

La herramienta *make* sirve como ejemplo claro de esta necesidad. Supongamos que tenemos un programa compuesto por múltiples módulos que se están escribiendo y actualizando por diversos programadores desde equipos distintos. En cualquier momento, desde cualquier puesto, puede solicitarse la generación de un programa ejecutable mediante el comando *make*. Pero ahora estamos en un entorno distribuido con reparto de carga, de tal manera que la compilación de un fichero no tiene por qué realizarse en la misma estación en la que se editó. El comando *make* comprueba que la hora de un fichero objeto debe ser posterior a la del correspondiente fichero fuente, pues de no ser así significa que no se ha actualizado el fichero objeto, y se genera la compilación del fichero fuente. Pues bien, supongamos que un fichero compilado genera un objeto con la hora 2144. Poco después el encargado de ese fichero fuente lo edita con su hora local 2143 (obviamente, el equipo donde se edita va retrasado respecto a la estación en la que se compila). Cuando el programador ejecuta el comando *make*, éste se encuentra con que el fichero fuente se modificó con la hora 2143, mientras que el objeto lo generó una compilación en el momento 2144, lo que le hace suponer que es resultante de la última actualización del fichero fuente. Claramente se observa que el programa ejecutable resultante estará formado a partir de un módulo objeto obsoleto, con lo que su comportamiento no será el esperado, y el programador no entenderá por qué no funciona.

También se tiene necesidad de una sincronización horaria en algunos programas de autenticación de mensajes, como Kerberos, o sistemas de respaldos de ficheros (*backups*) que se realizan en función de las horas de última modificación de los ficheros.

Ya que la medida del tiempo es tan básica para nuestra forma de pensar, y el efecto de no tener los relojes sincronizados puede ser dramático, comenzaremos por ver cómo podría conseguirse la sincronización de los relojes de todas las estaciones del sistema distribuido o, lo que es lo mismo, que todos tengan la misma hora simultáneamente. A esto se le conoce como la **sincronización de los relojes físicos**.

Relojes Físicos



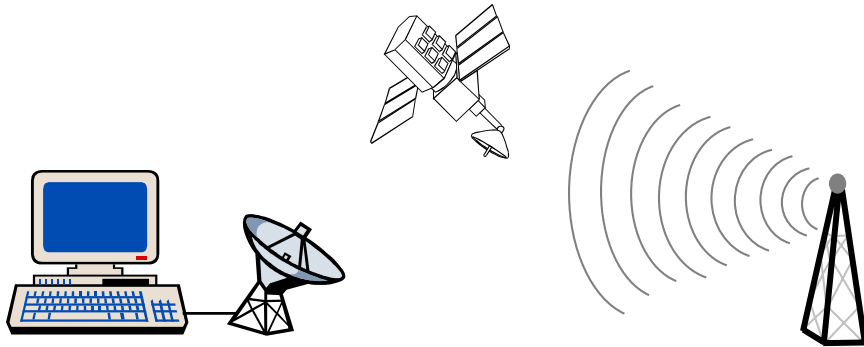
Cada ordenador dispone de su propio reloj físico. Este reloj es un dispositivo electrónico basado en un cristal de cuarzo que oscila a una determinada frecuencia, y que además puede programarse para generar interrupciones a intervalos determinados (cada cierto número de oscilaciones). Sabiendo cada cuanto tiempo se producen estas interrupciones (llamadas **interrupciones de reloj**), pueden aprovecharse para llevar la cuenta del tiempo, y por consiguiente de la fecha y la hora actual. Los sistemas operativos lo hacen mediante un contador que se va incrementando a medida que se producen las interrupciones de reloj.

Esta fecha y hora se suele utilizar para indicar el momento en el que suceden ciertos eventos del sistema, como por ejemplo, la hora de creación o modificación de un fichero de datos. La indicación del momento en que sucede un cierto evento se denomina **marca de tiempo** (*time stamp*). Para comparar marcas de tiempo producidas por ordenadores que cuentan con un mismo modelo de reloj, parece que bastaría con saber la diferencia de los valores que tienen los contadores de sus respectivos sistemas operativos, sin embargo esta suposición está basada en una premisa falsa, pues es prácticamente imposible que dos relojes "iguales" oscilen exactamente a la misma frecuencia, sean o no del mismo tipo.

Los relojes basados en un cristal están sujetos a una desviación o **deriva**, es decir que cuentan o miden el tiempo a velocidades distintas, por lo que progresivamente sus contadores van distanciándose. Por muy pequeño que sea el periodo de oscilación de los dos relojes, la diferencia acumulada después de muchas oscilaciones conduce a una diferencia claramente observable. La velocidad de deriva de un reloj es el cambio por unidad de tiempo en la diferencia de valores entre su contador y el de un reloj perfecto de referencia. Para los relojes de cristal de cuarzo esta velocidad de deriva está alrededor de 10^{-6} , lo que significa una diferencia de un segundo cada 11,6 días.

Los relojes más precisos utilizan osciladores atómicos, cuya precisión es del orden de 10^{14} (un segundo en 1.400.000 años). Diversos laboratorios (con reloj atómico) de todo el mundo le indican el número de ticks (interrupciones de reloj) periódicamente al Bureau Internationale de l'Heure (BIH), el cual calcula su valor medio produciendo el **Tiempo Atómico Internacional** (TAI), teniendo en cuenta que el tiempo se empezó a contar el 1 de enero de 1958, una vez que el segundo se definió como el tiempo que necesita un átomo de Cesio-133 en realizar 9.192.631.770 transiciones. No obstante, las unidades de tiempo que utilizamos tienen un origen astronómico (**tiempo universal**), es decir, están definidas en función de los periodos de rotación y traslación de la Tierra alrededor del Sol, y resulta que este periodo de rotación se está alargando gradualmente, pues nuestro planeta se está frenando, debido, principalmente, a la fricción de las mareas, efectos atmosféricos y a las corrientes de convección en el núcleo de la tierra. Esto quiere decir que el tiempo atómico y el universal tienden a desincronizarse.

La gradual ralentización de la rotación de La Tierra da lugar a que, en la actualidad, un día TAI (86.400 segundos TAI) sea 3 milisegundos menor que el día solar. Por esto, cuando la diferencia llega a los 900 ms, el BIH añade un segundo "bisiesto" al contador TAI, dando lugar al **Tiempo Universal Coordinado** (UTC), el cual es la base de todas las medidas políticas, civiles y militares del tiempo.



Precisión: 0,1 - 10 ms

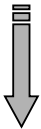
Ordenador a 10 MIPS

100.000 instr.

¡ Varios Mensajes !

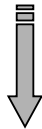
SINCRONIZACIÓN CON HORA UTC

Hora Local < Hora UTC



Avanzar Hora Local

Hora Local > Hora UTC



¿ Retrasar Hora Local ? ¡NO!

Ralentizar Reloj Local

En Sistemas Distribuidos
es Importante
**Mantener Sincronizados
los Equipos**

Dos Equipos con $V_d = \rho$ Opuesta
Después de $\Delta t \Rightarrow$ Dif. hora = $2\rho\Delta t$

Para mantener dif. máxima = δ seg.

Sincronización cada $\delta/2\rho$ seg.

El tiempo UTC está disponible a través de emisoras de radio que emiten señales horarias, mediante satélites geoestacionarios y GPS cuyas precisiones son 0,1 a 10 ms.; 0,1 ms.; y 1 ms. respectivamente. A pesar de esta relativamente buena precisión, nos encontramos con dos problemas. Por una parte tenemos que un receptor de señales horarias es un dispositivo demasiado caro comparado con el precio de una estación de trabajo y, además, en caso de disponer de él, resulta que la medida del tiempo con una precisión de 10 milisegundos puede resultar insuficiente para un ordenador a 10 MIPS, en el que en esos 10 ms. pueden ejecutarse 100.000 instrucciones y pueden transmitirse varios mensajes.

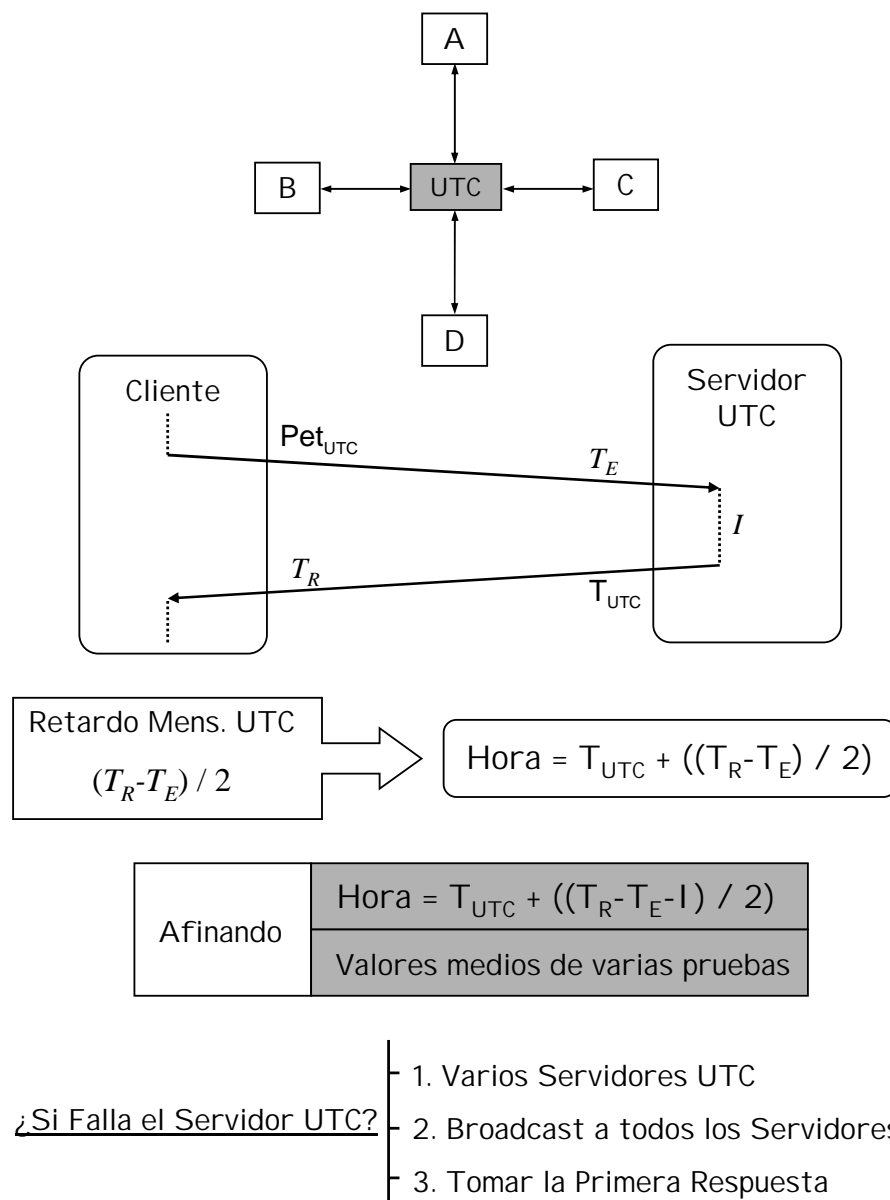
Supongamos que un equipo dispone de un receptor de estas señales horarias. El reloj hardware del ordenador puede ir más despacio o más deprisa que el UTC. ¿Qué hacemos para sincronizarlo con el UTC? Si la hora del reloj local es anterior a la hora UTC, simplemente avanzamos la hora local hasta la hora UTC; lo más que puede suceder es que parezca como si se hubieran perdido algunos ticks de reloj. Pero ¿qué se puede hacer si la hora local es mayor que la hora UTC? ¿Retrasamos la hora local hasta la hora UTC? Esto nunca debe hacerse, pues se puede confundir a las aplicaciones. Pensemos, en nuestro ejemplo del comando `make`, si se atrasase el reloj local, podría suceder que en un mismo equipo, la hora de compilación de un programa fuera anterior a la hora de edición, lo cual confundiría al comando `make`.

La solución para la situación en que la hora local es mayor que la hora UTC no estriba en atrasar la hora, sino en ralentizar la velocidad del reloj local hasta que se sincronice con la hora UTC, y entonces volver a ponerlo a su velocidad. De esta manera no se producen paradojas como la de la compilación del fichero.

En los sistemas distribuidos resulta incluso más importante la sincronización horaria entre los distintos equipos que el mantener mínima la diferencia con la hora UTC. Si la máxima velocidad de deriva de un reloj hardware de un ordenador es ρ , dos relojes cuya divergencia de la hora UTC sea opuesta, en un momento Δt después de haberse sincronizado pueden mostrar una diferencia horaria de $2\rho\Delta t$. Esto quiere decir que si se quiere garantizar que dos ordenadores no difieran más de δ segundos en su hora, deben sincronizarse (su hora software) al menos cada $\delta/2\rho$ segundos.

Veamos a continuación dos métodos para realizar esta sincronización: el algoritmo de Cristian y el algoritmo de Berkeley.

Algoritmo de Cristian



El algoritmo de Cristian [1984] está pensado para entornos en los que un ordenador, al que llamaremos Servidor de Tiempo, está sincronizado con una hora UTC, y el resto de las estaciones quieren sincronizarse con él.

Periódicamente (cada $\delta/2p$ segundos), cada ordenador cliente envía un mensaje al servidor de tiempo preguntándole la hora actual. El servidor de tiempo responde con un mensaje que contiene la hora actual T_{UTC} . Cuando el cliente recibe el mensaje, debe actualizar su reloj. Si su hora local es posterior a T_{UTC} debe ralentizar la velocidad del reloj hasta ajustarse a la hora UTC (si a cada interrupción del reloj normalmente se añaden 10 ms. al contador de tiempo, se empiezan a añadir solamente 9). Si su hora es anterior a la hora recibida, directamente actualiza su reloj, o bien lo acelera hasta igualarlo a la hora UTC.

Ahora hay que considerar el error cometido, pues se ha requerido un tiempo para la transmisión del mensaje con la hora UTC desde el servidor hasta el cliente, y lo que es peor, este retardo es variable, dependiendo de la carga de la red. Veamos cómo el algoritmo de Cristian calcula este retraso. Lo que hace es medir el tiempo que se tarda en recibir la respuesta desde que se envía el mensaje de petición. Para ello, simplemente tiene que anotar la hora de envío T_E y de recepción T_R . (Aunque se utilice el reloj local, dado el corto periodo que transcurre, su precisión es suficiente). En ausencia de otra información, el tiempo estimado de propagación del mensaje será $(T_R - T_E) / 2$. Así, cuando el cliente recibe el mensaje con la hora actual, tiene que añadirle el retardo calculado.

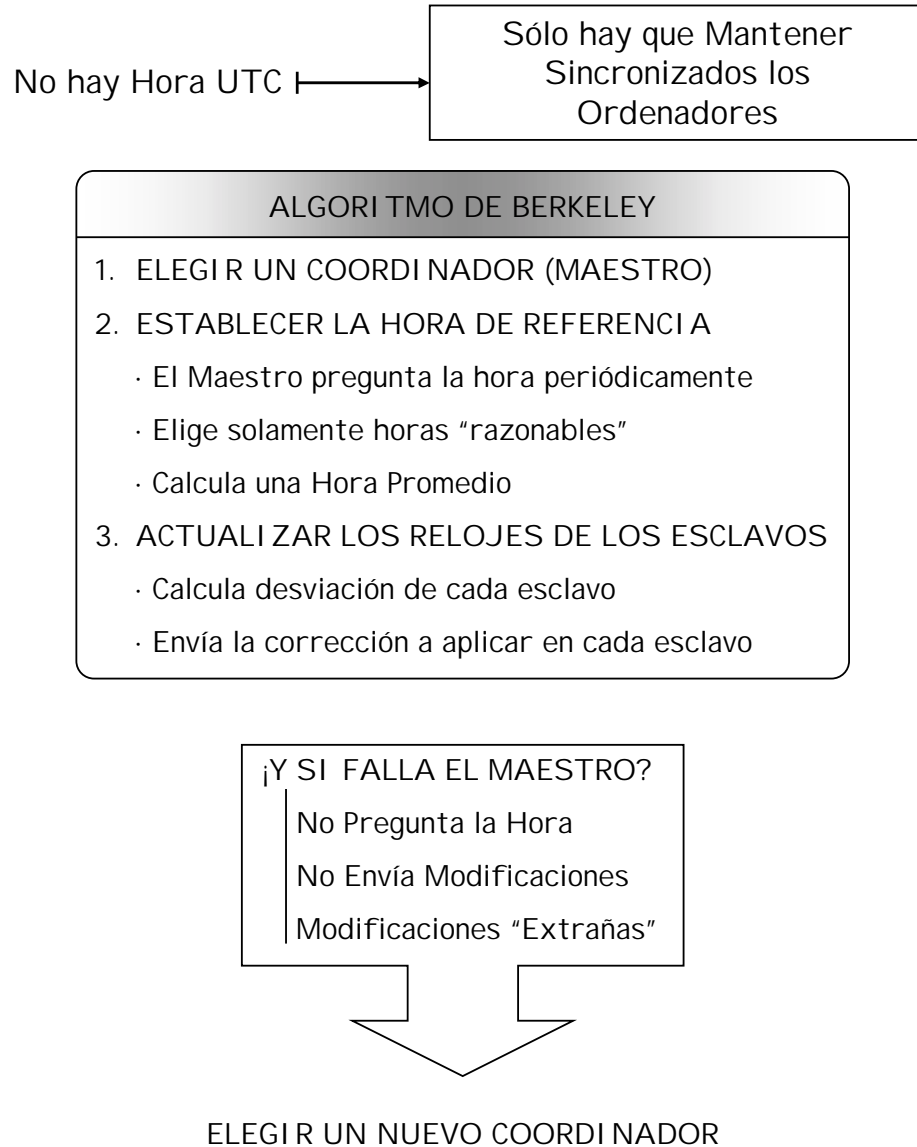
La duración estimada del retardo puede mejorarse si se sabe lo que tarda el servidor de tiempo en tratar la interrupción que genera la llegada del mensaje y atender la petición. Si a este tiempo lo llamamos I , el tiempo estimado de propagación del mensaje será $(T_R - T_E - I) / 2$. Todavía puede mejorarse un poco más esta estimación si se conoce el tiempo mínimo de propagación del mensaje, es decir, cuando no hay ninguna congestión en la red.

Otra posibilidad de mejora consiste en realizar múltiples pruebas de envío y recepción al servidor para calcular el tiempo de retardo. Cualquier medida $T_R - T_E$ que sobrepase cierto umbral se desecha, suponiendo que ha habido un problema de congestión, por lo que su retardo no es significativo. Con las medidas válidas se calcula un valor promedio que se toma como tiempo de propagación del mensaje.

El problema que se presenta es el mismo de todos los servicios ofrecidos por un único servidor: la posibilidad de fallo. Cristian sugiere que la hora debe suministrarse por varios servidores de tiempo sincronizados mediante receptores de tiempo UTC; el cliente envía su petición a todos los servidores y toma la primera respuesta recibida.

Este algoritmo no contempla problemas de malfuncionamiento o fraude por parte del servidor. No obstante, hay algoritmos (Marzullo [1984]) para distinguir los servidores válidos de los que funcionan mal o son impostores.

Algoritmo de Berkeley



El algoritmo de Berkeley (Gusella y Zatti [1989]) está pensado para entornos en los que no se dispone de ningún receptor de tiempo UTC, y lo único que se pretende es que todos los ordenadores se mantengan sincronizados con una misma hora.

En este algoritmo, entre todos los equipos del sistema distribuido eligen un coordinador para que actúe como maestro o servidor de tiempo. Al contrario que en el algoritmo de Cristian, en el que el servidor era pasivo, en este caso el coordinador elegido pregunta la hora, periódicamente, al resto de las estaciones (los esclavos). Cuando los esclavos responden cada uno con su hora local, el maestro estima los retardos de propagación de los mensajes con cada uno de los esclavos (de manera similar a Cristian y eliminando los valores que sobrepasan cierto umbral) y calcula un tiempo promedio con las horas recibidas y la suya propia.

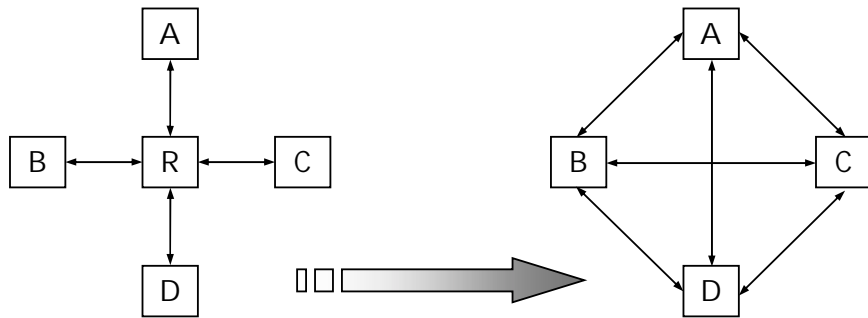
Queda por actualizar los relojes de los esclavos. Ahora el maestro, en lugar de enviar la hora coordinada a todos los esclavos (lo que introduciría un cierto error debido al tiempo de transmisión), envía a cada uno el desfase que tiene con la hora promedio calculada. Recibido este desfase, cada equipo actualiza su reloj adelantándolo directamente, o ralentizándolo temporalmente en caso de ir adelantado.

Podríamos pensar que no se está teniendo en cuenta el tiempo de propagación en el envío del mensaje desde el esclavo hacia el maestro y que, por lo tanto, la hora que se recibe en el maestro llega retrasada. En este caso esto no nos importa, pues lo que se quiere es calcular una hora promedio de entre las recibidas de todos los esclavos. En una red de área local, los tiempos de propagación de los mensajes entre los esclavos y el maestro será similar para todos ellos, con lo que la hora que indican todos los esclavos está tomada por todos ellos en el mismo momento.

El algoritmo realiza un "promedio tolerante a fallos" con los tiempos recibidos de los esclavos, es decir, que considera solamente el subconjunto de relojes que no difieren entre sí más de una cierta cantidad, considerando que los demás no están funcionando bien.

Si el maestro falla (no pregunta, no envía modificaciones de tiempos, o éstas son "demasiado extrañas"), simplemente se elige otro coordinador.

Los autores de este algoritmo realizaron un experimento con 15 ordenadores cuyos relojes tenían una precisión de 2×10^5 , y con un tiempo de propagación de mensajes de unos 10 ms., consiguiendo que sus relojes se sincronizaran con errores de alrededor de 20 milisegundos.



¡Problemático!

ALGORITMO DISTRIBUIDO

1. Se deben conocer los tiempos de propagación
2. Todos los equipos deben sincronizarse periódicamente
3. Al comienzo de cada intervalo, todos difunden su hora local
4. Esperan un tiempo para recibir la hora de los demás equipos
 - Se desechan los valores extremos
 - Con todos los valores recibidos se calcula la desviación media
5. Con la desviación media se actualiza el Tiempo Local.
6. Todos los nodos tienen la misma hora.

Los algoritmos de sincronización de relojes físicos que hemos visto son centralizados, pues se basan en los servicios ofrecidos por un único servidor. No obstante, hay que reconocer que el método de Berkeley es, digamos, “más democrático”, pues entre todos los clientes eligen, mediante un algoritmo distribuido, el que van a utilizar como maestro. Sin embargo, una vez elegido, el escenario de sincronización vuelve a ser centralizado, con sus consiguientes problemas (fallos, cuellos de botella, ...), aunque en caso de caída del equipo maestro, los esclavos pueden volver a elegir otro maestro, cosa que en el algoritmo de Cristian no era posible.

Veamos un ejemplo de **algoritmo totalmente distribuido**. Consiste en que todos los equipos conectados al sistema deben sincronizarse cada cierto intervalo de tiempo I , de tal manera que el n -ésimo intervalo o momento de sincronización tiene lugar en el momento $T_0 + n \cdot I$, donde T_0 es el momento en el que los equipos comienzan a sincronizarse tras su arranque.

En este algoritmo, cada nodo debe conocer cuál es el tiempo de propagación del mensaje desde cada origen a cada destino, para lo cual debe conocerse bien la topología de la red, o calcularlo mediante mensajes (sondas) que se envían (y se devuelven) para calcular simplemente el tiempo de propagación.

Al comienzo de cada intervalo, cada máquina difunde un mensaje a todo el grupo (incluido él mismo) indicando su hora local. Ya que todos los relojes son distintos y no andan a la misma velocidad, la difusión no se produce exactamente de una manera simultánea. Para cada uno de estos mensajes que se recibe, sabiendo el tiempo de propagación desde su nodo y comparando con la hora local, se calcula la desviación propia respecto al nodo del mensaje recibido.

Cuando un equipo ha enviado el mensaje de difusión de su hora, arranca un temporizador y se pone a esperar los mensajes de difusión del resto de los equipos, los cuales deberán llegar dentro de un cierto intervalo de tiempo. Cuando han llegado todos los mensajes del resto de las estaciones o ha vencido la temporización, se calcula el valor medio de las desviaciones y se ajusta el tiempo local (acelerando o frenando el reloj local).

Como todos los nodos reciben el mismo conjunto de mensajes, después de cada sincronización todos deben tener la misma hora.

Pueden eliminarse los valores más extremos recibidos, por suponer que puedan deberse a relojes que están fallando o mintiendo.

El sistema DCE de la OSF utiliza este algoritmo, con ligeras variaciones, para la sincronización de sus relojes.

Es muy difícil sincronizar perfectamente
múltiples relojes distribuidos

Muchas Veces

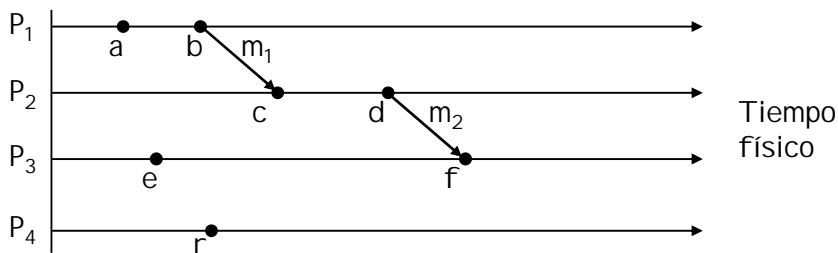
- No se requiere estar sincronizados con una hora UTC
- Dos procesos que no interactúan, no requieren sincronismo

Simplemente se requiere que los procesos
estén de acuerdo en el orden en el que se han
producido los eventos que les relacionan.

En lugar de trabajar
con Relojes Físicos \longrightarrow Se trabaja con
RELOJES LÓGICOS

Relación "Sucedio Antes" $a \rightarrow b$ (a sucedió antes que b)

1. Dos eventos en el mismo proceso
suceden en el orden que indica su reloj común.
2. En procesos comunicados por un mensajes,
el envío es siempre anterior a la recepción.



Hay que buscar la forma
de asignar valores a los
relojes lógicos R , tal que

- Si $a \rightarrow b \Rightarrow R(a) < R(b)$
- Siempre deben ir hacia adelante

Como hemos visto hasta ahora, resulta muy difícil sincronizar múltiples relojes distribuidos para que todos ellos mantengan una única hora estándar con la suficiente precisión que no dé lugar a ambigüedades.

Sin embargo, Lamport señaló que la sincronización de relojes no necesita ser absoluta. Por una parte, si dos procesos no interactúan, no tienen ninguna necesidad de que sus relojes respectivos estén sincronizados, pues la falta de sincronización no será observable y no causará problemas. Por otra parte, lo que normalmente importa no es que todos los procesos estén sincronizados según una misma hora exacta, sino que todos estén de acuerdo sobre el orden en el que se suceden los eventos que se producen. En el ejemplo del comando `make`, si la edición tuvo lugar a las 10:00 (hora local), y la compilación a las 10:06 según el reloj local de la máquina en la que se compiló (aunque realmente tuvo lugar dos minutos después de la edición), este desfase horario no es significativo, siempre que se sepa que la compilación fue posterior a la edición.

Para algunos entornos en los que no se requiere una sincronización exacta con una hora externa de referencia (tiempo UTC), en lugar de tratar con los relojes físicos que hemos visto, se trabaja con **relojes lógicos**, en los que solamente tiene importancia el orden de los eventos, no la medida del momento exacto en el que se producen.

Para sincronizar relojes lógicos, Lamport definió la relación "**sucedio antes**", según la cual, la expresión $a \rightarrow b$ quiere decir " a sucedió antes que b ", y significa que todos los procesos coinciden en que primero sucedió el evento a y posteriormente el b . Esta relación se observa directamente en dos situaciones:

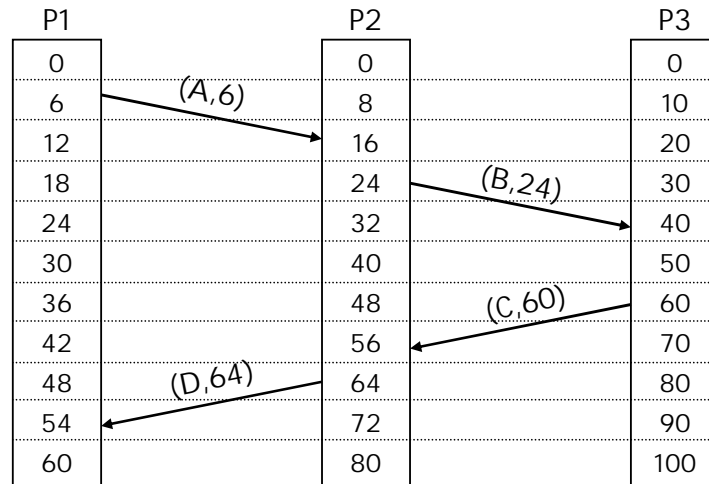
1. Si dos eventos se producen en el mismo proceso, tiene lugar en el orden que indica su reloj común.
2. Cuando dos procesos se comunican mediante un mensaje, el evento de enviarlo se produce siempre antes del evento de recibirlo.

Si dos eventos, x e y , se producen en procesos diferentes que no intercambian mensajes (ni directa ni indirectamente), entonces no es cierto $x \rightarrow y$, ni $y \rightarrow x$. En este caso se dice que tales eventos son concurrentes, lo que significa que sencillamente no se sabe nada (ni se necesita saber) sobre cuál de ellos sucedió primero.

Se debe observar que la relación "sucedio antes" es transitiva, es decir, si $a \rightarrow b$ y $b \rightarrow c$, entonces $a \rightarrow c$.

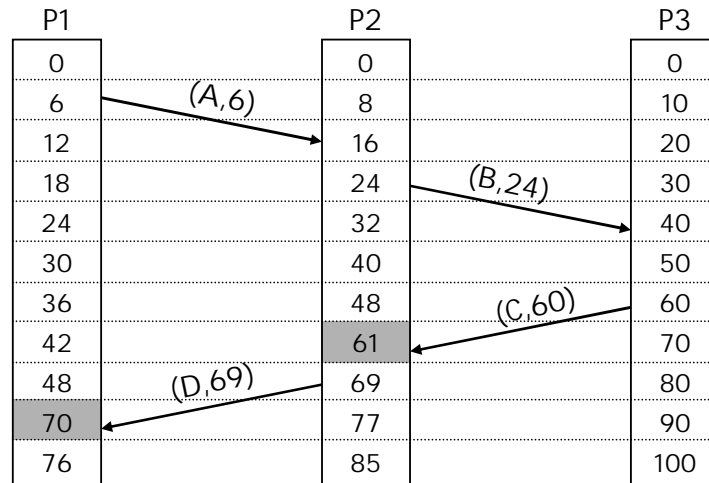
Lo que se necesita es una forma de medir el tiempo tal que para todo evento a , se le pueda asignar un valor de tiempo $R(a)$ en el que todos los procesos estén de acuerdo. Estos valores deben tener la propiedad de que si $a \rightarrow b \Rightarrow R(a) < R(b)$. Además, como ya sabemos, el reloj R siempre debe ir hacia adelante, nunca hacia atrás. Veamos el mecanismo de Lamport para asignar tiempos a los eventos.

Figura A



¡ SORPRENDENTE !

Figura B



¡ RAZONABLE !

Lamport inventó un mecanismo denominado **reloj lógico**, el cual es simplemente un contador software monótono creciente, cuyo valor no necesita tener ninguna relación concreta con ningún reloj físico. En cada ordenador hay un reloj lógico R que se utiliza para poner marcas de tiempo a los eventos que se producen, tal que $R_p(a)$ indica la marca de tiempo del evento a en el proceso p . Así, las dos situaciones en las que se observa la relación “sucedió antes” se tratan de la siguiente manera con los relojes lógicos:

1. R_p se incrementa en una unidad antes de que se produzca cada evento en el proceso, es decir, $R_p = R_p + 1$.
2. a) Cuando un proceso p envía un mensaje m , se le pega al mensaje el valor $t = R_p$.
b) Cuando el proceso q recibe el mensaje (m, t) , calcula $R_q := \max(R_q, t)$, y aplica el paso 1 antes de marcar el evento de recepción del mensaje (m, t) .

Veamos esto con un ejemplo, considerando los procesos de la *Figura A*. Los tres procesos se ejecutan en máquinas distintas, cada una con su propio reloj a su correspondiente velocidad.

Como se puede ver, cuando el reloj ha producido 6 ticks (eventos) en el proceso 1, el reloj del proceso 2 ha producido 8 ticks, y el del proceso 3, 10 ticks. En el momento 6, el proceso 1 envía un mensaje A al proceso 2. El tiempo requerido para la propagación del mensaje depende del reloj de referencia. En cualquier caso, el reloj del proceso 2 marca 16 cuando llega el mensaje. Si el mensaje lleva asociada una marca de tiempo del momento del envío, el proceso 2 deducirá que el tiempo de propagación del mensaje A ha requerido 10 ticks, lo cual es simplemente posible. De igual manera, el mensaje B del proceso 2 al proceso 3 necesita 16 ticks, que también es posible. Fijémonos que aunque desde un punto de vista externo o absoluto, los tiempos de propagación de los mensajes calculados por los procesos son erróneos, la deducción que puedan realizar los procesos sobre el orden en que se han producido los eventos, es correcta.

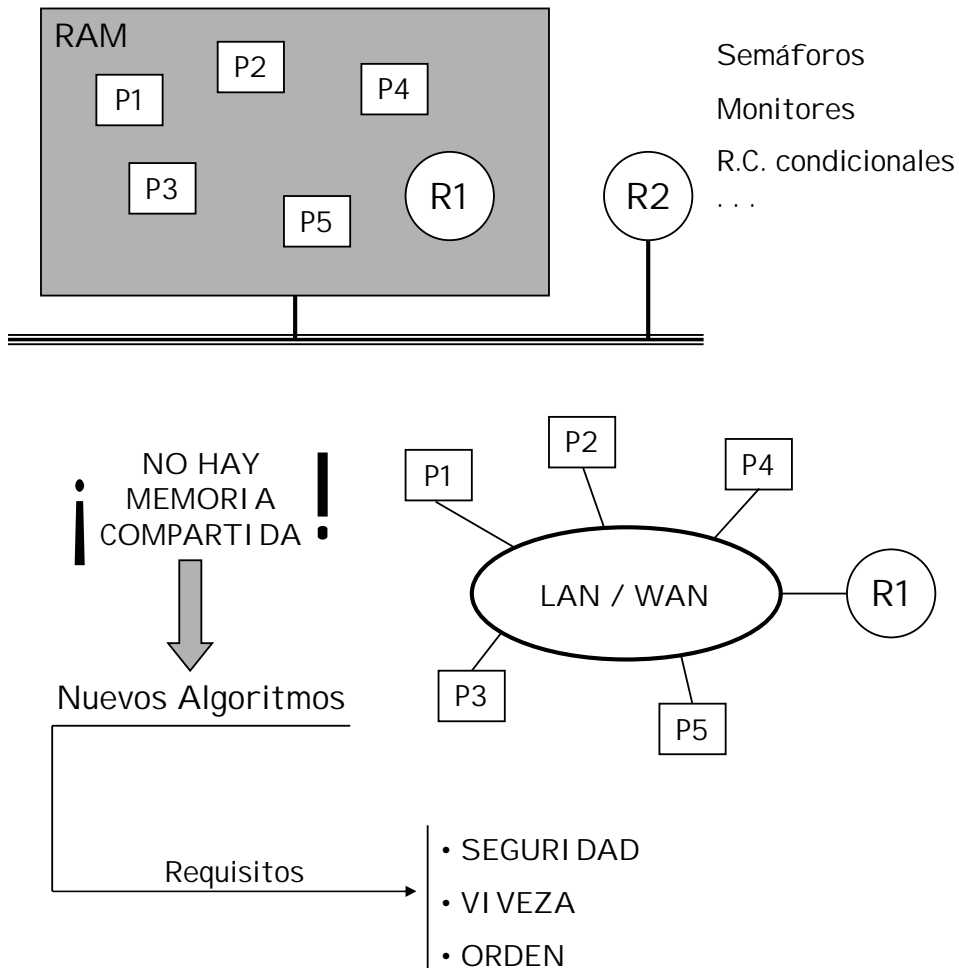
Ahora vayamos con los mensajes de respuesta. El mensaje C sale del proceso 3 en el momento 60, y llega al proceso 2 en el 56 ¿?. Igualmente, el mensaje D sale del proceso 2 en el momento 64 y llega al proceso 1 en el momento 54. Claramente, estos valores son imposibles, y es lo que hay que evitar.

La solución de Lamport establece que si el mensaje C salió en el momento 60, según la relación “sucedió antes”, debe llegar en el momento 61 o después. Por eso, cada mensaje lleva pegado el momento del envío (según el reloj del emisor), y cuando llega al receptor, si el reloj de éste muestra un valor anterior al momento del envío, el receptor actualiza su reloj al del momento del envío más una unidad. Según esto, en la *Figura B*, vemos ahora que el mensaje C llega al proceso 2 en el momento 61, y D llega al proceso 1 en el momento 70.

Con una simple suma, este algoritmo consigue la relación de orden global válido para todos los procesos. La condición es que entre dos eventos el reloj debe haber generado, al menos, un tick, así, si un proceso envía o recibe dos mensajes sucesivos, debe haber avanzado su reloj un tick como mínimo entre ellos.

Sin embargo, según esto, dos eventos, en dos máquinas distintas, sí pueden producirse simultáneamente. Cuando se requiera distinguirlos u ordenarlos (**relación de orden total**), a la marca de tiempo de cada evento se le puede añadir el número de su proceso, como un valor decimal (separado por una coma). Así, si dos eventos se producen en el momento 40 en los procesos 1 y 2, sus marcas de tiempo respectivas serán 40,1 y 40,2.

Exclusión Mutua



Aunque el problema de las regiones críticas se produce primordialmente en programas con variables compartidas, en los sistemas distribuidos también se producen situaciones en las que hay recursos compartidos que no pueden ser utilizados por más de un proceso al mismo tiempo.

En sistemas con uno o más procesadores que comparten memoria, suelen utilizarse mecanismos como semáforos, monitores, regiones críticas condicionales, etc. Sin embargo, en los sistemas distribuidos los procesos ya no comparten la memoria física (suponemos que no se dispone de memoria compartida distribuida), por lo que debemos pensar en otros algoritmos que nos proporcionen la exclusión mutua.

Los **requisitos básicos** que debe cumplir un algoritmo de exclusión mutua son los siguientes:

Seguridad: Como mucho, sólo un proceso puede estar ejecutándose dentro de la región crítica en un momento dado.

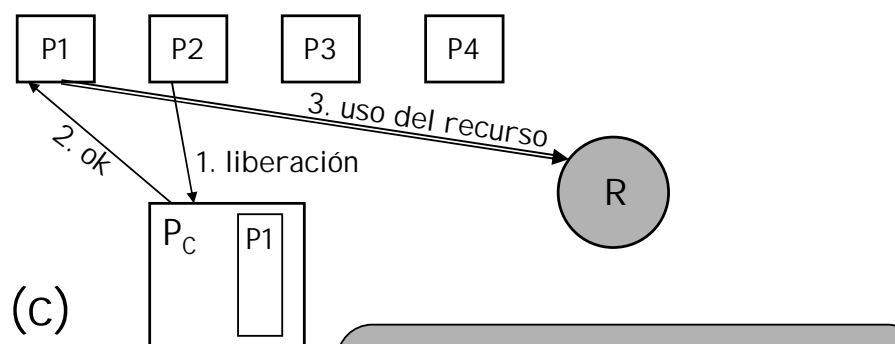
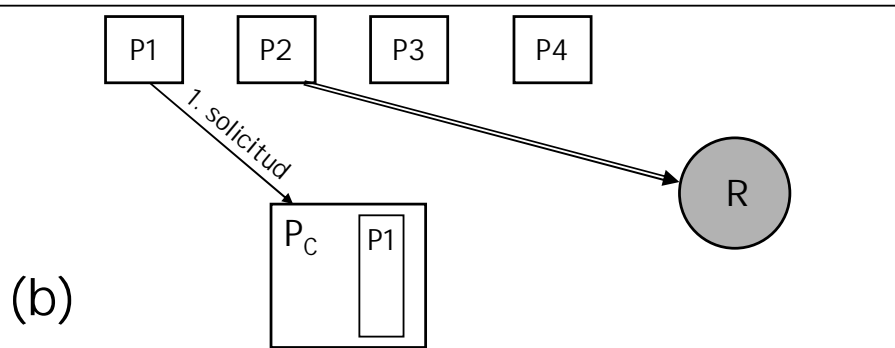
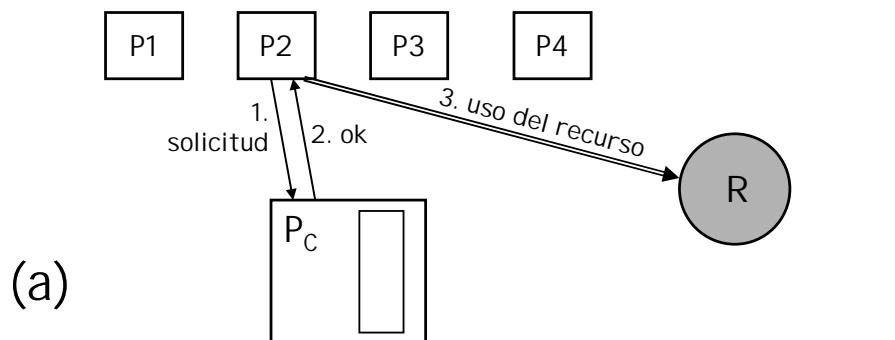
Viveza: Un proceso que desea entrar en una región crítica debe poder entrar en algún tiempo (siempre que cualquier proceso ejecutándose dentro de la región la abandone). Esto quiere decir que no deben producirse interbloqueos ni inanición.

Orden: La entrada a la región crítica debe realizarse en el orden causal “sucedió antes” definido por Lamport. (Un proceso P1 puede continuar su ejecución mientras espera a que se le conceda la entrada en una región crítica y durante este tiempo puede enviar un mensaje a otro proceso P2. Este proceso P2 después de recibir el mensaje de P1 puede intentar entrar en la misma región crítica que P1. Pues bien, este requisito de orden especifica que P1 deberá entrar en la región crítica antes que P2).

Se pueden tomar dos enfoques en el diseño de estos algoritmos para proporcionar la exclusión mutua: **algoritmos centralizados** y **algoritmos distribuidos**.

Comenzaremos por ver el caso del enfoque centralizado.

Algoritmo Centralizado



Requiere 3
Mensajes

PROBLEMAS

- Fallo del Cliente
- Fallo del Coordinador
- Cuello de Botella

La forma más simple de conseguir la exclusión mutua en un sistema distribuido la presentó Lamport en 1978, y se basa en la utilización de un proceso servidor como **coordinador de la región crítica**. (Ya veremos más adelante cómo elegir un coordinador).

Siempre que un proceso cliente P2 quiera entrar en la región crítica, envía un mensaje de solicitud al coordinador, indicándole la región crítica en la que desea entrar. Si ningún otro proceso está en ese momento dentro de la región crítica, el coordinador envía una respuesta otorgando el permiso. Cuando le llega la respuesta al proceso solicitante, entra en la región crítica. Cuando P2 sale de la región crítica, se lo comunica al coordinador para liberar su acceso exclusivo.

Supongamos ahora que mientras P2 está dentro de la región crítica otro proceso P1 pide permiso para entrar en la misma región crítica. El coordinador sabe que la región crítica está ocupada, por lo que no concede el permiso –por ej. no respondiendo a la petición– con lo cual el proceso P1 se queda bloqueado esperando una respuesta. La solicitud de P1 queda encolada en una cola de peticiones en el coordinador.

Cuando P2 sale de la región crítica, se lo hace saber al coordinador, el cual saca de la cola de espera de esa región crítica la primera solicitud (la de P1) y le envía un mensaje otorgándole el permiso para entrar en la región crítica. Al recibir P1 el mensaje de permiso, accede a la región crítica.

Es fácil ver que este algoritmo cumple las tres reglas de exclusión mutua: 1) no hay más de un proceso en un momento dado; 2) no hay inanición, pues cuando un proceso sale entra otro (si está esperando), y 3) la concesión de permisos de acceso se realiza por orden.

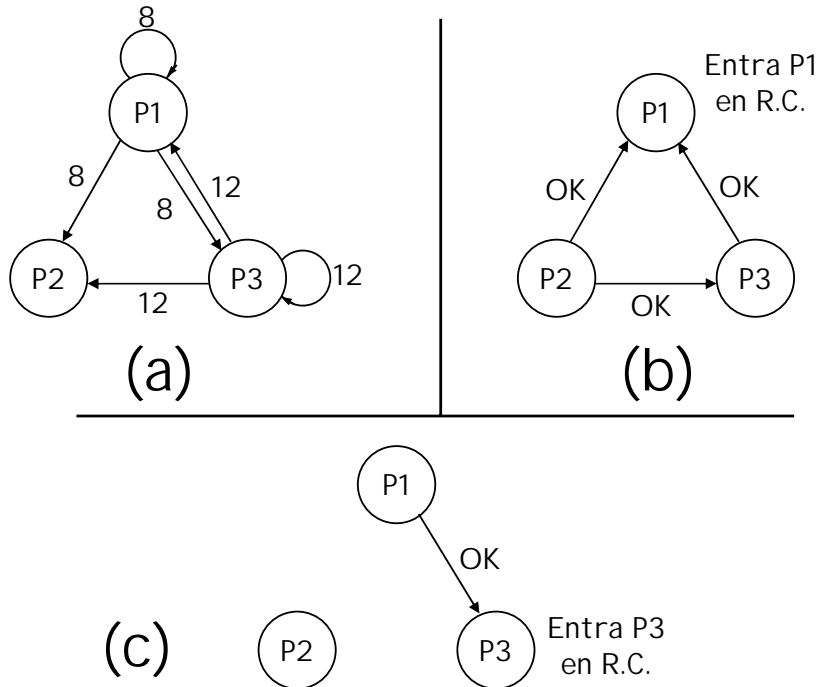
Este algoritmo **consigue su cometido con tres mensajes**: solicitud, concesión y liberación, y puede utilizarse para cualquier política de asignación de recursos.

Puntos de fallo

Pero este algoritmo tiene sus inconvenientes: puede producirse un fallo de los clientes o del coordinador. **Si falla un proceso cliente cuando está dentro de la región crítica** nunca notificará su salida al coordinador, con lo que no entrarán más procesos (se produce inanición).

Si falla el coordinador, un cliente no puede distinguir entre un coordinador muerto o una denegación del permiso. Esto puede detectarlo cualquier proceso cliente al intentar comunicarse con él si se envían confirmaciones a los mensajes. Para recuperarse de esta situación, los procesos clientes deben elegir un nuevo coordinador, que puede ser incluso uno de los clientes. Una vez elegido, y a menos que el estado de peticiones se mantenga replicado, para restaurar el estado original, los clientes que habían solicitado la entrada en la región crítica deben volver a pedirlo.

Otro problema que se puede presentar es que si el coordinador se ocupa de múltiples recursos o si el sistema está formado por un grupo muy numeroso de equipos, el proceso coordinador puede convertirse en un **cuello de botella** del sistema.

Algoritmo por Marcas de Tiempo

Requiere $2(n-1)$ Mensajes para Entrar en la R.C.

PROBLEMAS

- n puntos de fallo (mejora con ACKs)
- Más tráfico en la red
- Se ha replicado la carga del coordinador
- Si hay varias R.C. \Rightarrow Atención constante a las peticiones

Debido a los problemas del algoritmo centralizado, Lamport diseñó un algoritmo distribuido en 1978 que requiere una ordenación total de todos los eventos del sistema (por ej. mediante relojes lógicos). Ya que requería un alto número de mensajes para la entrada en la región crítica, en 1981 Ricart y Agrawala propusieron una alternativa mejorada. Veamos cómo funciona esta última alternativa.

Cuando un proceso quiere entrar en una región crítica, **construye un mensaje de petición** con el nombre de la región, su identificador de proceso y la hora actual. A continuación envía el mensaje a todos los procesos del grupo, incluido él mismo. **Cuando un proceso recibe una petición** de otro proceso, según su estado, toma una de las siguientes acciones:

1. Si el receptor no está en la región crítica y no quiere entrar, envía un mensaje OK al remitente.
2. Si el receptor está dentro de la región crítica, no responde, simplemente encola la petición.
3. Si el receptor quiere entrar en la región crítica pero todavía no lo ha hecho, compara la marca de tiempo del mensaje que le ha llegado con la de su propio mensaje de petición. Si la marca de su solicitud es menor que la del mensaje recibido, lo encola y no devuelve nada. En caso contrario, le envía un mensaje OK y la propia petición.

Después de enviar una solicitud de entrada en una región crítica, el proceso se pone a **esperar hasta que ha recibido el permiso** de TODOS los demás procesos del grupo. Tan pronto como recibe todos los permisos, puede entrar en la región crítica. **Cuando sale de la región, envía mensajes de OK** a todos los mensajes de su cola y por último los borra de dicha cola.

Al igual que en el caso centralizado, se consigue la exclusión mutua sin interbloqueo y sin inanición, **requiriendo por cada entrada en la región crítica $2(n-1)$ mensajes**, siendo n el número de procesos en el sistema.

Puntos de fallo

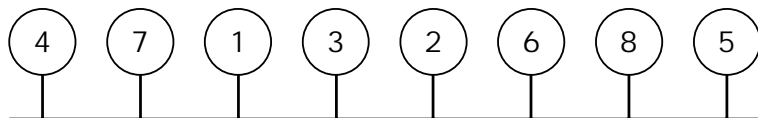
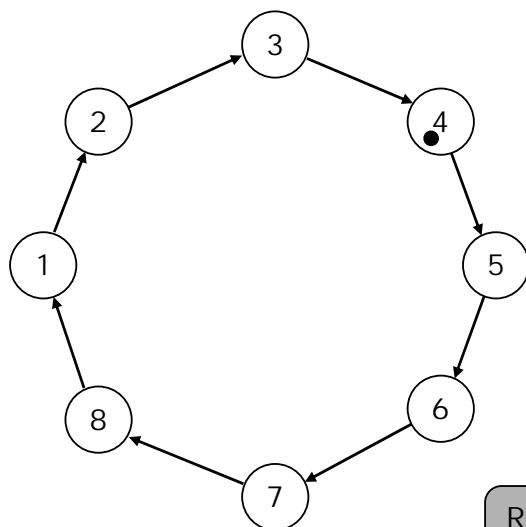
Ahora ya no tenemos el punto de fallo del algoritmo centralizado **¡ahora hay n puntos de fallo!** Si cualquiera de los equipos falla, no responderá a las peticiones, lo cual se interpretará (incorrectamente) como la denegación del permiso, bloqueando así todas las peticiones subsiguientes al fallo. ¡Hemos multiplicado por n las probabilidades de fallo y con mucho **más tráfico de red!**

Esto puede arreglarse si a los mensajes de petición siempre se les responde con un ACK, de tal forma que si un equipo no contesta, se puede suponer que está caído y se le saca del grupo.

Otra posibilidad consiste en entrar en la región crítica cuando se reciba el permiso de una mayoría, no de todos los equipos. En este caso, cuando un proceso concede un permiso de entrada en una región crítica, no puede volver a concederlo hasta que el que está dentro de la región crítica notifique su salida.

Si en el algoritmo centralizado teníamos el problema del cuello de botella en el equipo coordinador, ahora **hemos replicado la carga del coordinador en todos los equipos**, pues todos ellos están involucrados en la decisión sobre la entrada en una región crítica.

Por último, si los procesos comparten varias regiones críticas, todos ellos están obligados a atender continuamente los mensajes de petición, incluso cuando están dentro de una región crítica.

Algoritmo *Token-Ring*Visión física de una redAnillo Lógico
con Testigo

Requiere 1 a $n-1$ Mensajes
para Entrar en la R.C.

PROBLEMAS

- Pérdida del testigo
 - Fallo del cualquier proceso
 - Múltiple regeneración del testigo.
 - Posible ocupación inútil de la red.
- Mejor con ACKs

Aquí tenemos otro algoritmo distribuido para conseguir la exclusión mutua en un sistema distribuido, desarrollado por Le Lann en 1977.

Dada una serie de equipos conectados por una red de comunicaciones (por ej., mediante un bus ethernet), **debe formarse un anillo lógico**, de tal forma que cada equipo sea un nodo del anillo, y donde a cada uno de ellos se le asigna una posición en el anillo (por ej. según su dirección de red). Cada nodo del anillo debe saber cuál es la dirección de su vecino (el siguiente nodo del anillo).

Al arrancar el sistema, al proceso de posición 1 se le da un testigo o ficha (*token*), la cual va a ir circulando por el anillo según vamos a ver. **Cuando el proceso k tenga el testigo, debe transferirlo, mediante un mensaje, al proceso $k+1$** (módulo número de nodos). Así, el testigo va a ir pasando por todos los nodos del anillo.

Cuando un proceso recibe el testigo, **si quiere entrar en la región crítica, retiene el testigo** y entra en la región crítica. Cuando salga de la región crítica le pasará el testigo al siguiente nodo del anillo.

Este algoritmo, que también cumple las reglas de la exclusión mutua (aunque no respeta el orden), puede requerir de 1 a $n-1$ mensajes para conseguir entrar en la región crítica.

Puntos de fallo

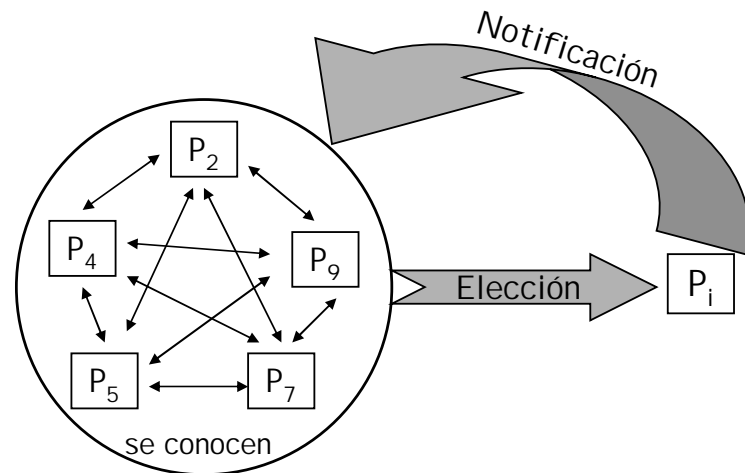
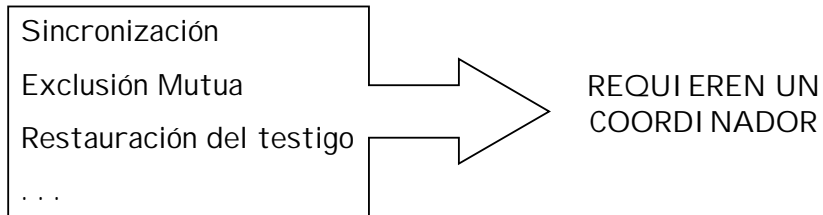
En primer lugar debe partirse de una red sin pérdida de mensajes, pues de **perderse un mensaje con el testigo**, sería difícil detectarlo, pues el tiempo entre apariciones sucesivas del testigo en un nodo en la red no está acotado, por lo que el hecho de que no se consiga el testigo no quiere decir que se haya perdido, quizás algún nodo lo está utilizando. Como es común, también presenta problemas **si falla cualquier proceso** del anillo, sobre todo si esto sucede mientras el proceso está **dentro de la región crítica**, y por lo tanto con el testigo, pues el siguiente vecino no tiene forma de saber si el nodo anterior ha fallado o simplemente está realizando un trabajo largo dentro de la región crítica.

Algunos de estos problemas pueden resolverse si los mensajes de paso del testigo se contestan con otro de reconocimiento. Así, si se pierde un mensaje, se pueden hacer reintentos. Después de n reintentos puede suponerse que el nodo destino está caído, con lo que habrá que eliminarlo del anillo lógico. Por último, simplemente hay que enviar el testigo al nodo que estaba a continuación del nodo en fallo. Para poder hacer esto, se requiere que todos los nodos mantengan la configuración completa del anillo.

No obstante, cuando se utilizan temporizaciones para detectar la pérdida del testigo, puede darse el caso de que varios procesos regeneren el testigo, lo cual, obviamente, no es deseado. Misra ideó un algoritmo en 1983 que detectaba la pérdida del testigo sin necesidad de temporizaciones, utilizando para ello dos testigos. Este algoritmo no vamos a tratarlo aquí, pero lo describe el texto de Raynal en el apartado 2.4.2.

Si cuando un proceso recibe el testigo no desea entrar en la región crítica, simplemente debe pasarlo a su siguiente vecino. Así, si ninguno de los procesos desea entrar en la región crítica, el testigo estará circulando por la red a gran velocidad (ocupación inútil de la red). Chow 10.1.3 describe un algoritmo para una estructura en árbol que evita la transmisión de mensajes cuando ningún procesador desea entrar en la región crítica.

Elección de Coordinador



OBJETIVO: Asegurar que cuando comience una elección, se concluya con que todos los procesos están de acuerdo en cuál es el nuevo coordinador



Como ya hemos visto en apartados anteriores, muchos algoritmos distribuidos utilizan un proceso coordinador que se encarga de algunas funciones necesarias para los demás procesos del sistema (sincronización de relojes, exclusión mutua, restauración del testigo o *token*, etc.). Si este proceso coordinador se cae por cualquier motivo, para que el sistema continúe funcionando se requiere un nuevo coordinador en algún nodo. Los algoritmos que determinan el proceso que va a actuar de coordinador se denominan **algoritmos de elección**.

Para simplificar, en los apartados que siguen, supondremos una correspondencia biunívoca entre procesos y nodos de la red (caso de extrema distribución), y nos referiremos a ellos indistintamente como procesos, nodos, equipos o procesadores.

Los algoritmos de elección suponen que cada proceso activo tiene asociado un identificador de proceso que puede utilizarse como prioridad dentro del sistema distribuido (no prioridad del proceso en su máquina). Así, diremos que la prioridad del proceso P_i es i .

El coordinador siempre va a ser el proceso con mayor prioridad. Por lo tanto, cuando un coordinador cae, el algoritmo de elección debe elegir el proceso activo con mayor prioridad. Una vez elegido el proceso coordinador, el identificador o prioridad de éste debe enviarse a todos los procesos activos del sistema. Además, los algoritmos de elección deben proporcionar un mecanismo para que los procesos rearmados (o recuperados), puedan averiguar cuál es el coordinador actual.

El objetivo del algoritmo de elección es asegurar que cuando se comience una elección, se concluya con que todos los procesos están de acuerdo en cuál es el nuevo coordinador.

En cierto modo, el problema de la elección es similar al de la coordinación para la exclusión mutua, ya que todos los procesos deben estar de acuerdo sobre quién tiene el testigo. Sin embargo, en una elección todos los participantes deben saber quién tiene el testigo (quién es el líder), mientras que en la exclusión mutua los procesos que no tienen el testigo lo único que tienen que saber es simplemente que no lo tienen. También, en la exclusión mutua, los algoritmos suelen estar diseñados para trabajar en ausencia de fallos; mientras que **la elección se suele realizar precisamente cuando se produce el fallo de un coordinador** y debe elegirse otro, es decir, que la gestión de fallos es parte integral del protocolo.

Vamos a ver dos algoritmos en los que se presupone que cada proceso del grupo conoce los identificadores del resto de los procesos. Lo que no saben los procesos del grupo es cuáles están arrancados y cuáles están parados.

Algoritmo *Bully*

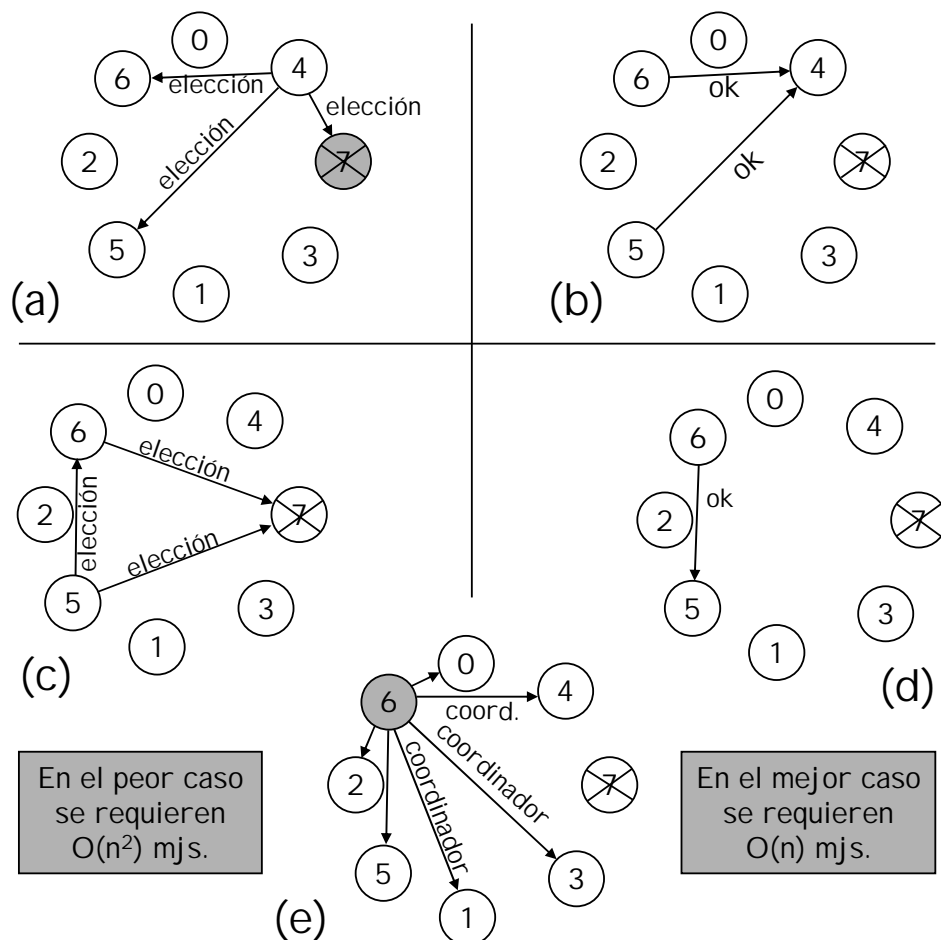
T_p : Tiempo de propagación
 T_t : Tiempo de Tratamiento

Si $T_{respuesta} > 2T_p + T_t$

¡ Nodo Caído !

Si Nodo Caído = Coordinador

ARRANQUE
ELECCIÓN



García-Molina diseñó en 1982 el algoritmo *Bully* (algoritmo del matón o del más gallito), el cual parte de estas suposiciones:

1. Los mensajes se entregan dentro de un periodo máximo de T_p segundos (tiempo de propagación).
2. Un nodo responde a todos los mensajes dentro de T_t segundos desde su recepción (tiempo de tratamiento del mensaje).
3. Los procesos están física o lógicamente ordenados, de tal forma que cada uno de ellos tiene un único identificador y sabe cuántos procesos hay.

Las dos primeras suposiciones implican que **se detecta la caída de un nodo** si no se responde a un mensaje en un tiempo máximo $T = 2T_p + T_t$.

Cuando un proceso P detecte la caída del coordinador, arrancará una elección de coordinador de la siguiente manera:

1. P envía un **mensaje de elección** a todos los procesos con identificadores más altos que el propio y se queda esperando un **mensaje OK**.
2. Si ningún proceso responde, P gana la elección y se convierte en el coordinador.
3. Si cualquiera de los procesos responde, P ya no tiene nada que hacer (no será coordinador), y se queda esperando a un **mensaje de coordinador**.

En cualquier momento un proceso puede recibir un *mensaje de elección* de cualquiera de sus colegas con menor identificador. En tal caso, el receptor devuelve un *mensaje OK* indicando que está vivo y que se hace cargo de la situación. A continuación envía a su vez *mensajes de elección* a los procesos superiores, y así sucesivamente. Cuando para un proceso P_i no existe un proceso superior, o no contesta nadie a los *mensajes de elección*, indica que tal proceso P_i es el proceso vivo con mayor identificador.

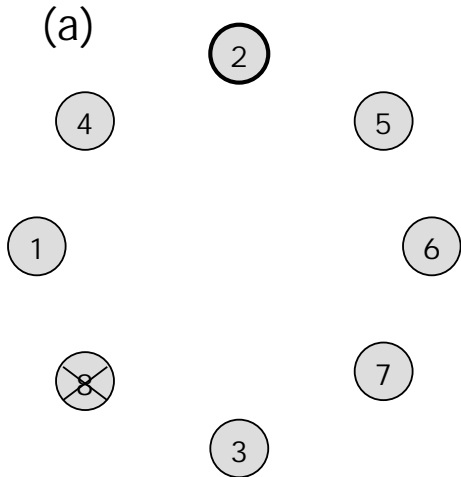
Cuando un proceso se da cuenta de que es el coordinador, debe enviar un *mensaje de coordinador* a todos los demás procesos. Cuando estos reciben tal mensaje, registran al proceso emisor como coordinador y continúan su ejecución.

Si reanuncia un proceso que estaba caído, debe iniciar una elección como la comentada, de tal forma que si él resulta ser ahora el proceso activo de mayor identificador, se convierte en el nuevo coordinador y envía los *mensajes de coordinador* al resto de los procesos. Si no lo es, recibirá un *mensaje de coordinador* indicándole cuál es el coordinador del grupo.

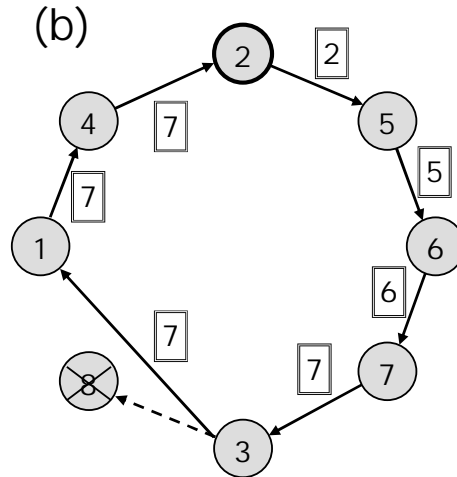
En el peor de los casos, este algoritmo requiere del orden de n^2 mensajes para elegir un coordinador (siendo n el número de procesos). **El mejor de los casos** se produce cuando el fallo del coordinador lo detecta el proceso con el segundo identificador más alto (el siguiente al coordinador caído).

Algoritmo para Anillos

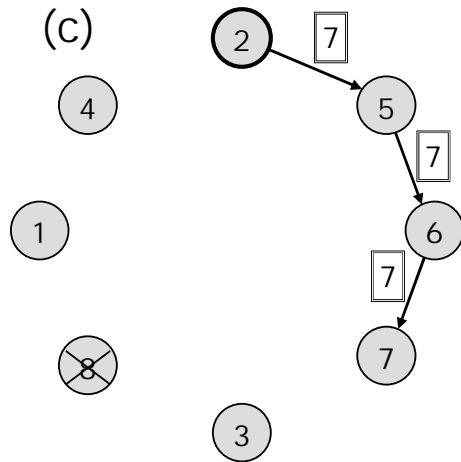
☞ Cada proceso se comunica con un vecino



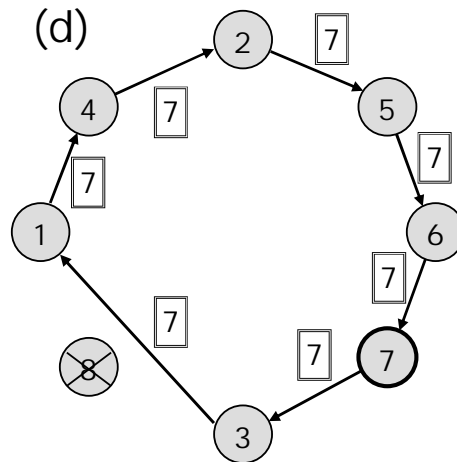
Todos nodos "no participantes"
Nodo 8 falla. Lo detecta el 2



El 2 envía un "mensaje-elección"
"Participantes": 2, 5, 6, 7, 3, 1, 4



Alto al recibir el 7 el mensaje



El 7 difunde Mensaje-coordinador
Todos nodos "no participantes"

Hay varios algoritmos para topologías en anillo (físicas o lógicas). Aquí veremos uno de Chang y Roberts, de 1974, que utiliza el "principio de extinción selectiva".

Este algoritmo se utiliza cuando se tienen las siguientes circunstancias:

- Los procesos están organizados, en cualquier orden, en un anillo.
- No conoce el número total de procesos (n).
- Cada proceso se comunica con su vecino (por ej. el de su izquierda).

El algoritmo es el siguiente: Inicialmente todos los procesos son "**no participantes**". En un momento dado, uno cualquiera de los procesos decide arrancar una elección. Entonces se pone en estado "**participante**" y envía un **mensaje de elección** a su vecino. Este mensaje contiene el identificador del proceso que ha arrancado la elección.

Cuando el vecino recibe el **mensaje de elección**, establece su estado como "participante" y comprueba el identificador del mensaje. Si es mayor que su propio identificador, se le envía directamente a su vecino; si su identificador es mayor que el recibido, lo substituye en el mensaje e, igualmente, se lo envía al vecino.

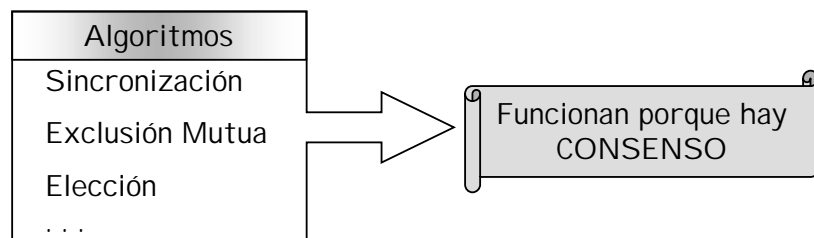
Y así sucesivamente hasta que el **mensaje de elección** llega a un proceso que comprueba que el identificador del mensaje es el propio, lo cual quiere decir que el **mensaje de elección** ha pasado por todos los procesos y solamente ha sobrevivido el mayor identificador, el suyo.

Ahora, este proceso que ha recibido el mensaje con su identificador debe enviar un **mensaje de coordinador** a su vecino indicando el identificador del proceso que va a actuar como coordinador. Cuando un proceso recibe un **mensaje de coordinador** debe poner su estado como "no participante" y reenviárselo a su vecino. Así hasta que el mensaje vuelva al coordinador. En ese momento todos los procesos saben qué proceso es el coordinador y vuelven a quedar como "no participantes".

Puede ocurrir que varios procesos arranquen a la vez una elección y envíen **mensajes de elección**. Pero cuando un proceso "participante" recibe un **mensaje de elección** de otro proceso y el identificador recibido sea menor que el propio, el mensaje se tira. Se tira por que como este proceso ya era "participante", quiere decir que ya había enviado algún mensaje de elección con un identificador mayor que el recién recibido. Así, los todos los mensajes de elección se van extinguiendo excepto uno, el que lleva el identificador más alto.

Si solamente un proceso ha arrancado la elección, **el peor caso** se da cuando el identificador más alto lo tiene su vecino de la derecha, con lo que se necesitarán $n-1$ mensajes para alcanzarle, más otros n mensajes para dar otra vuelta y volver al proceso de mayor identificador. Por último, se requieren otros n mensajes para hacer llegar a todos los procesos el **mensaje de coordinador**, haciendo un total de $3n - 1$ mensajes.

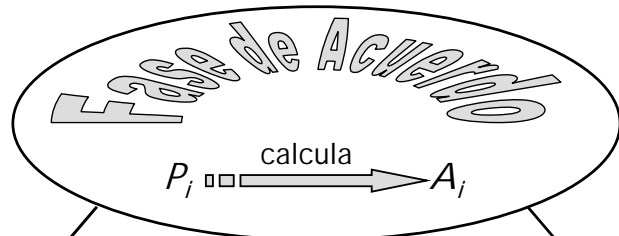
El mejor caso se produce cuando la elección la arranca únicamente el proceso de mayor identificador, requiriendo solamente $2n$ mensajes.



☞ M Procesadores P_1, P_2, \dots, P_m intentan llegar a un acuerdo

☞ Un subconjunto F de procesadores **funcionan mal**

☞ Cada procesador propone un valor V_i



Todos procesadores correctos llegan al mismo valor de acuerdo: A

$$A = F(V_i \text{ correctos})$$

ESCENARIO DE COMUNICACIONES

- Comunicación Síncrona Fiable
- Enlaces Punto a Punto Bidireccionales
- El Mensaje IndicaCuál es el Remitente

Casi todos los investigadores de este campo piensan que el problema más fundamental del cálculo distribuido es el problema del **acuerdo distribuido** o del **consenso distribuido**, es decir, cómo conseguir que un conjunto de procesos (en distintos nodos) se pongan de acuerdo en el valor de un cierto dato. Por ejemplo, en los apartados anteriores hemos visto que los algoritmos para sincronización de relojes, exclusión mutua o elección se basan en que todo el conjunto de los procesadores distribuidos se ponen de acuerdo sobre un cierto valor que se intercambian mediante mensajes.

El **enunciado formal** del protocolo para un acuerdo distribuido es el siguiente:

- Hay M procesadores $P = p_1, p_2, \dots, p_M$ que intentan llegar a un acuerdo.
- Un subconjunto de ellos, F , funcionan mal, y el resto funcionan bien.
- Cada uno de los procesadores propone un valor V_i .
- Mediante el protocolo de acuerdo, cada procesador calcula un valor de acuerdo A_i .
- Cuando la fase de acuerdo termina, se deben cumplir las dos condiciones siguientes:
 - C1: Para todos los procesadores correctos, el valor acordado debe ser el mismo (A).
 - C2: El valor del acuerdo, A , debe ser una función de los valores iniciales $\{V_i\}$ de los procesadores correctos.

Ahora intentaremos definir el problema **de una manera más informal**. Para cualquier protocolo válido de sincronización o coordinación que utilice un conjunto de procesadores, se supone que el protocolo funciona si la información que se intercambia en el protocolo es correcta. Ahora bien, si uno de los procesadores no funciona correctamente puede intercambiar información errónea, lo cual puede conducir a que el protocolo no produzca el efecto deseado (sincronización de relojes, exclusión mutua, elección de un coordinador, etc.).

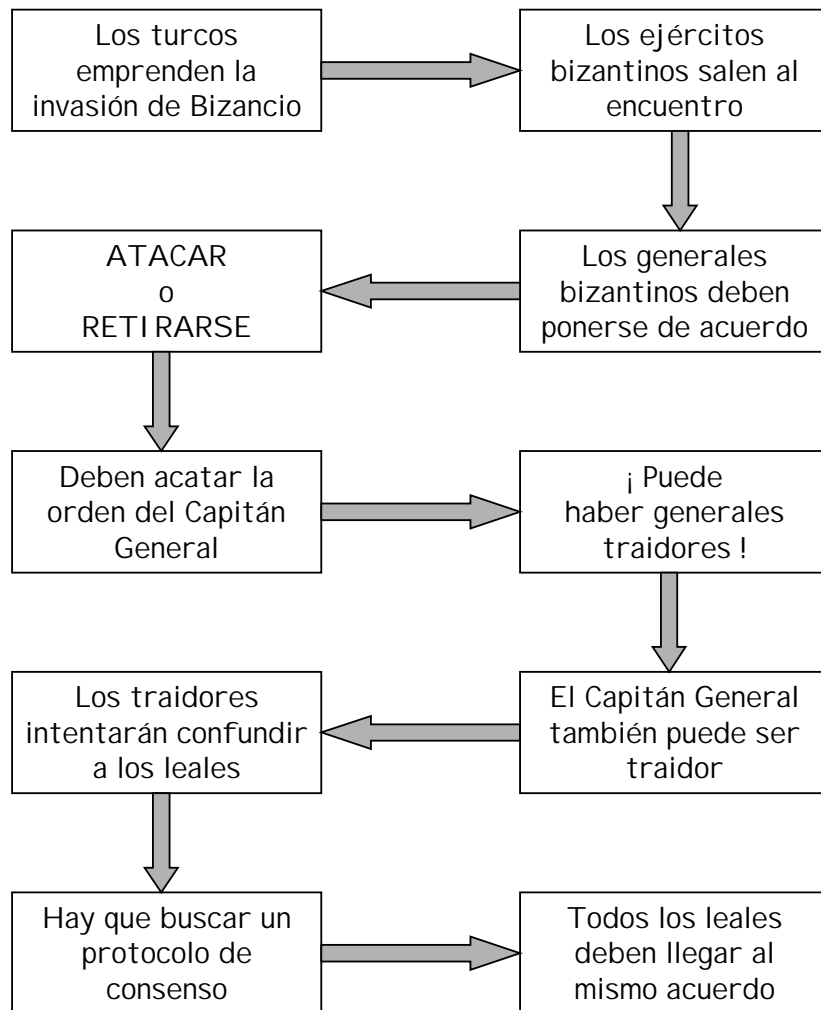
Tenemos entonces, que para que funcione correctamente cualquier protocolo de coordinación o sincronización, se debe partir primero de que los procesadores correctos se pongan de acuerdo en que todos ellos tienen la misma información o el mismo valor del dato que intercambian para ponerse de acuerdo. El problema del consenso consiste en conseguir que aún en presencia de procesadores erróneos (que funcionan mal), los procesadores correctos sean capaces de ponerse de acuerdo sobre un valor, incluso aunque tal valor no sea el óptimo.

Hay situaciones en las que ni siquiera las comunicaciones están exentas de fallos, o en las que los equipos pueden dejar de funcionar (equipo parado o caído), pero para acotar y simplificar un problema suficientemente complicado, aquí supondremos que **las comunicaciones son fiables** y que **los equipos funcionan (bien o mal)**.

Así pues, el escenario de las comunicaciones utilizadas es el siguiente:

- La comunicación es síncrona sobre una red fiable.
- Todos los procesos están unidos mediante enlaces bidireccionales punto a punto comunicando a todos con todos. Esto quiere decir que no se puede hacer *broadcast*.
- Cada mensaje recibido indica fielmente cuál es el proceso emisor.

Los Generales Bizantinos



El Acuerdo Bizantino se refiere a algoritmos de acuerdo distribuido en los que algunos procesadores que no se comportan correctamente (por fallo o mal-intencionadamente) pueden enviar mensajes con información errónea por la red, lo cual puede llegar a evitar que los procesadores correctos alcancen un acuerdo.

El nombre del Acuerdo Bizantino proviene de la historia que se suele utilizar para plantear el problema. Veamos el **Problema de los Generales Bizantinos**, ya que se adapta totalmente a los escenarios descritos en la transparencia anterior.

“Un día, hace mucho tiempo, el sultán turco emprendió la invasión de Bizancio. El emperador de Bizancio, enterado de esto, avisó a sus ejércitos para que, desde puntos distintos, salieran al encuentro de los invasores. Cada ejército bizantino estaba dirigido por un general.

Los ejércitos que salieron de Constantinopla para encontrar a los turcos eran suficientemente poderosos como para resistir la invasión sólo si su acción estaba coordinada (todos atacan o todos se retiran). Después de varios días de marcha, los ejércitos bizantinos acamparon cerca del ejército turco. Esa noche cada general debía pensar en la posibilidad de un ataque al amanecer. Cada uno de los generales bizantinos tenía su propia opinión sobre la fortaleza del ejército turco y, por lo tanto, su propia idea sobre si convenía atacar o retirarse. Ya que el ataque tenía que estar coordinado, todos los generales habían llegado al acuerdo que todos acatarían una decisión de consenso. Así, por la noche, cada general envió mensajeros a los demás generales para adoptar una decisión consensuada.

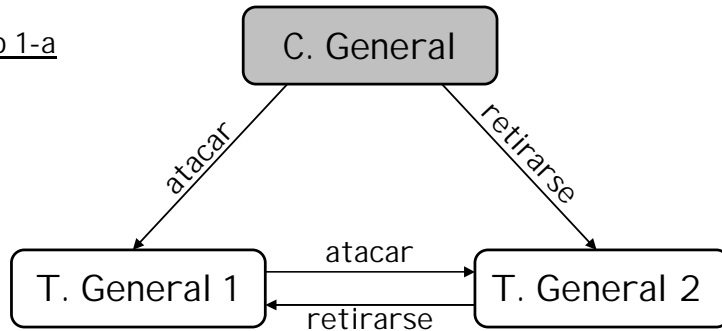
Sólo había un problema con el plan de los generales: los bizantinos tenían fama de traidores, por lo que algunos de ellos podrían haber sido sobornados por el sultán turco. Los generales leales sabían que si su ataque era coordinado, saldrían victoriosos; pero si unos atacaban mientras otros se retiraban, serían vencidos. Los generales traidores intentarían engañar a los leales para evitar el ataque coordinado. Por eso, los generales leales se debían poner de acuerdo en un protocolo para asegurar un acuerdo aún en presencia de algunos traidores.

En el ejército bizantino hay un capitán general y varios tenientes generales, y se supone que se debe acatar la orden del capitán general, pero en caso de que no sea así, todos los generales leales, al menos, sí deben tomar la misma decisión, pues la peor situación se da cuando unos atacan y otros se retiran. No se debe olvidar que el propio capitán general también puede ser traidor y puede intentar confundir a los tenientes generales.”

Veamos a continuación bajo qué circunstancias se puede lograr un acuerdo entre los generales leales y cuándo esto es imposible.

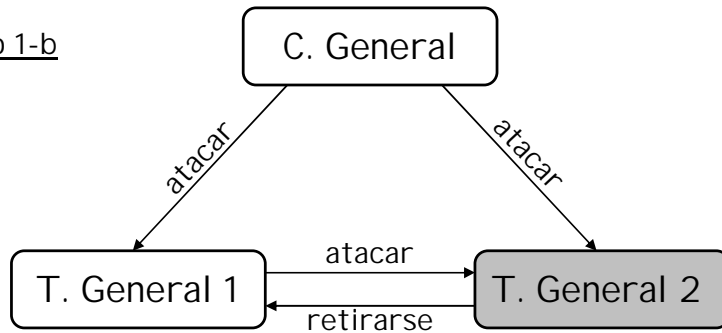
Tres Generales - Un Traidor

Caso 1-a



T. General 1: ¿C. General es Traidor?

Caso 1-b



T. General 1: ¿?

El problema comienza cuando el capitán general envía la orden a los tenientes generales. Ya que el capitán general puede ser traidor (y enviarles órdenes distintas), los tenientes deben intercambiarse mensajes indicando cuál es la orden que les ha llegado del capitán general. Pero claro, si hay algún teniente que es traidor, al resto de los tenientes les enviará órdenes contradictorias para confundirlos. Se pretende que, en cualquier caso, todos los generales leales adopten la misma postura (atacar o retirarse).

Lamport, Shostak y Pease desarrollaron un algoritmo en 1982 que ofrece una solución de consenso bajo ciertas circunstancias. Veamos algunos casos sencillos.

Caso 1: Tres generales. Si hay un capitán general y dos tenientes generales, y uno de ellos es traidor ¿pueden los generales leales llegar a un acuerdo, es decir, adoptar todos la misma postura –atacar o retirarse?

La respuesta es “no”, ya que no hay suficientes generales para formar una opinión consensuada.

Supongamos que el capitán general es el traidor (caso 1-a). Entonces le dirá al teniente general T_1 “atacar”, y al teniente T_2 “retirarse”. Los tenientes intentarán verificar la orden hablando entre ellos. T_1 le dirá a T_2 que su orden es “atacar”, mientras que T_2 le dirá a T_1 que a él le dijeron “retirarse”.

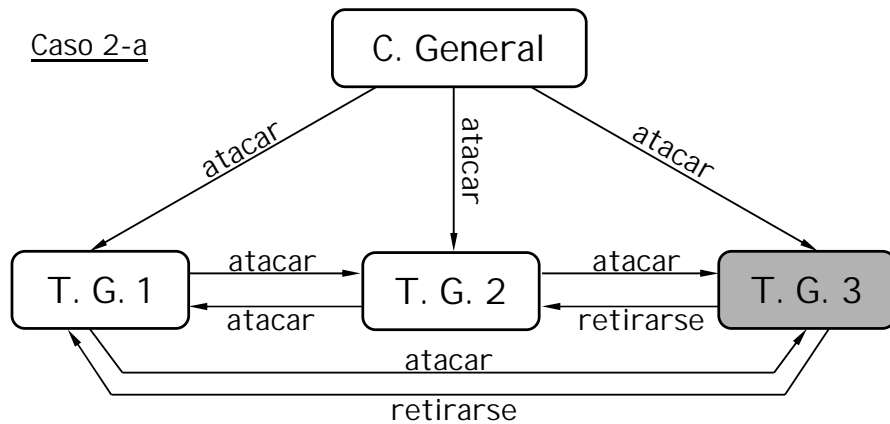
Los tenientes podrían deducir que el capitán es traidor, pero veamos otra situación en la que el capitán general es leal, y uno cualquiera de los tenientes es traidor. Por ejemplo, supongamos que el teniente 2 es traidor (caso 1-b). El capitán da la orden de “atacar” a los dos tenientes. Ahora los tenientes intercambian la orden recibida para asegurarse. Así, que T_1 le dice a T_2 “atacar”, pero como T_2 es traidor, le dice a T_1 que la orden recibida es “retirarse”.

Tenemos entonces que cuando es traidor el capitán general o el teniente 2, el teniente 1 escucha órdenes distintas del capitán y del teniente 2. Entonces ¿qué opción tomar?

Ya hemos visto que con tres generales no se puede llegar a un consenso. Veamos en la siguiente transparencia lo que ocurre con cuatro generales.

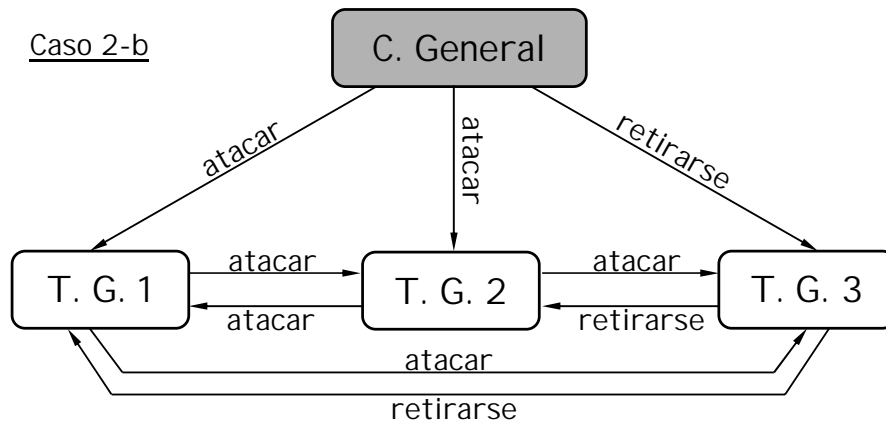
Cuatro Generales - Un Traidor

Caso 2-a



Orden de Consenso: ATACAR

Caso 2-b



Orden de Consenso: ATACAR

CONDICIÓN DE
CONSENSO

GENERALES > 3 · TRAI DORES

Caso 2: Cuatro generales. Ahora hay un capitán general y tres tenientes generales, y uno de ellos es traidor. ¿Se puede ahora llegar a un acuerdo? Veremos que ahora el acuerdo sí es posible. Al recibir la orden directa del capitán general y los mensajes de los otros dos tenientes generales, los tenientes leales deciden la orden de consenso según la siguiente función de mayoría:

Mayoría (v_1, v_2, \dots, v_n): Devolver el valor v que sea mayoría entre v_1, v_2, \dots, v_n

Supongamos que el capitán general es leal y que el teniente 3 es traidor (caso 2-a). El capitán general da la orden de “atacar”. En la figura vemos que un teniente leal recibe la orden de “atacar” del capitán y del otro teniente leal. El teniente traidor puede enviar cualquier orden a los otros dos tenientes, pero en cualquier caso, no afecta a la función de mayoría, por lo que los tenientes leales deciden “atacar”.

Si el capitán general es el traidor, independientemente de las ordenes que envíe a los tenientes, puesto que estos son leales, al intercambiarse las ordenes recibidas del capitán general los tres van a recibir los mismos tres mensajes, luego al aplicar la función de mayoría, los tres tenientes tomarán la misma decisión.

Como hemos visto, en presencia de un traidor se requieren, al menos, cuatro generales, en total, para poder llegar a un acuerdo.

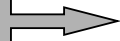
Estos resultados pueden aplicarse a cualquier número G de generales o procesadores entre los que haya t traidores. Lamport, Shostak y Pease demostraron que para poder llegara aun acuerdo, se requiere que:

$$G > 3t$$

Es decir que **el número total de generales (leales y traidores) debe ser, al menos $3t + 1$.**

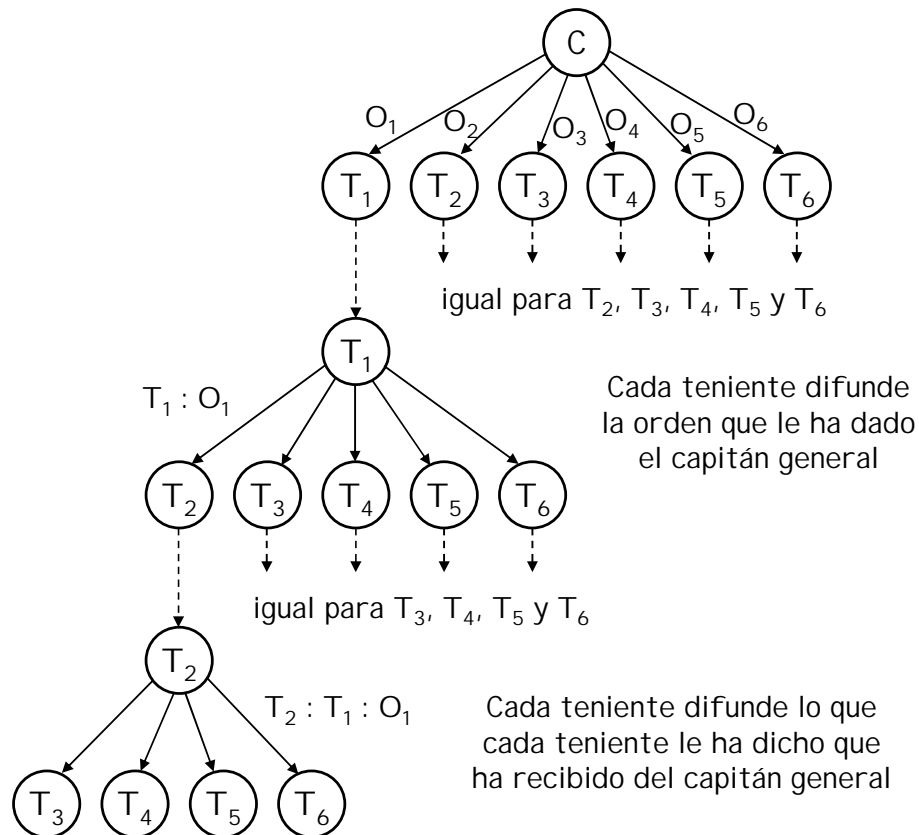
Aunque aquí no vamos a demostrar esto, en Chow 11.2 se trata con bastante profundidad este problema de los generales bizantinos.

t Traidores



$t + 1$ Rondas de Mensajes

Siete Generales - Dos Traidores



Hemos visto que cuando hay un traidor, se requieren dos rondas de mensajes, una del capitán general a los tenientes, y otra para comunicarse los tenientes entre ellos lo que les ha dicho el capitán a cada uno. Cuando hay más de un traidor se complica bastante el algoritmo, pues **para t traidores se requieren $t+1$ rondas de mensajes**.

Por ejemplo, para siete generales y dos traidores, el teniente T_1 no solamente tiene que decir a los demás tenientes lo que le ha ordenado el capitán general, sino también lo que el teniente T_2 le ha dicho a T_1 que le ha ordenado el capitán general, lo que el teniente T_3 le ha dicho a T_1 que le ha ordenado el capitán general, ... Asimismo, T_2 tendrá que decir a los demás tenientes lo que el teniente T_1 le ha dicho a T_2 que le ha ordenado el capitán general, etc., etc.

En el gráfico de la transparencia se ve un esbozo de la solución para siete generales y dos traidores.

Para entender la notación de los mensajes, lea el símbolo ":" como "dice". Por ejemplo, $T_1:O_1$ significa " T_1 dice que ha recibido la orden O_1 "; $T_2:T_1:O_1$ significa " T_2 dice que T_1 dice que ha recibido la orden O_1 ".