Capítulo 6
Sincronização em Sistemas Distribuídos

Edmilson Marmo Moreira

Universidade Federal de Itajubá - UNIFEI

Instituto de Engenharia de Sistemas e Tecnologias da Informação - IESTI

"Um homem que nunca muda de opinião, em vez de demonstrar a qualidade da sua opinião demonstra a pouca qualidade da sua mente."

Marcel Achard

1 Introdução

As técnicas de sincronização são importantes, pois freqüentemente os sistemas distribuídos precisam realizar tarefas que necessitam de um comportamento sincronizado.

Em sistemas com uma única CPU, regiões críticas, exclusões mútuas e outros problemas de sincronização, são geralmente resolvidos usando métodos tais como semáforos e monitores. Estes métodos não são adequados para sistemas distribuídos, uma vez que os processadores não compartilham o mesmo espaço de endereçamento.

A maneira como é tratado o tempo em um sistema distribuído é uma característica importante do projeto. Isto ocorre, porque o tempo é uma quantidade que se deseja medir com certa precisão, como, por exemplo, no registro da hora em que uma transação bancária distribuída foi realizada ou para se definir qual processo terá direito ao acesso a sua região crítica (COULOURIS; DOLLIMORE; KINDBERG, 2001).

Entretanto, a noção de tempo físico é relativamente problemática em sistemas distribuídos. Isto ocorre devido à dificuldade em se manter sincronizados os relógios dos diversos *hosts* do sistema.

Este capítulo apresenta uma discussão sobre relógios físicos e lógicos e como estes conceitos são utilizados para a sincronização de processos em um sistema distribuído.

2 Relógios físicos

Os relógios dos computadores são dispositivos físicos que geram interrupções com freqüência contínua. A saída dessas interrupções pode ser lida por um *software* que traduz o número dessas interrupções para um valor do tempo real. Esse número pode ser

usado para marcar qualquer evento dos processos que estão executando no computador. Essa marca é conhecida como ***timestamp***.

As aplicações que estão interessadas somente na ordem dos eventos e não no tempo real em que eles ocorrem, utilizam apenas o valor do contador. Sucessivos eventos irão corresponder a diferentes *timestamps* somente se a resolução do relógio, ou seja, a freqüência que o dispositivo trabalha for menor que a razão em que os eventos ocorrem.

Os cristais que são utilizados como dispositivos de relógio estão sujeitos a atrasos (*clock drift*). Estes atrasos são provocados por fenômenos naturais, principalmente a variação de temperatura, que altera a agitação dos átomos modificando a freqüência dos dispositivos.

Vários fatores colaboram para dificultar a sincronização dos relógios em um sistema distribuído, como por exemplo:

- Impossibilidade de manter todos os relógios trabalhando na mesma freqüência. Mesmo que os cristais sejam do mesmo material físico, a própria estrutura do computador onde ele se encontra faz com que o seu comportamento possa ser diferente do relógio de outro computador na mesma rede.

- Dificuldade de definir o tempo de propagação de uma mensagem por uma rede. Mesmo que uma mensagem seja enviada por um processo em um computador contendo a hora do relógio local para outro processo em outro computador, o tempo gasto para o recebimento da mensagem poderá apresentar variações.

- Possibilidade de ocorrer falhas nos processadores e/ou no meio de comunicação.

Todos esse fatores possuem mecanismos diferentes de tratamento.

O dispositivo físico que permite medir o tempo com maior precisão atualmente é o relógio atômico com base no Césio 133 (Cs^{133}). A freqüência desse dispositivo possui uma precisão de aproximadamente 10^{-13}. Essa precisão é enorme comparada com os cristais de quartzo, amplamente utilizados nos relógios comuns e que possuem uma precisão de 10^{-6}, dando uma diferença de um segundo a cada 1.000.000 de segundos ou 11,6 dias.

A saída do relógio atômico é utilizada como o padrão para o tempo real, sendo conhecido como *International Atomic Time*. Desde 1967, o padrão para o segundo é 9.192.631.770 oscilações do Césio 133.

Um padrão internacional bastante conhecido é o UTC (*Universal Coordinated Time*) que é baseado no relógio atômico. Diversas estações de rádio espalhadas pelo mundo operando em ondas curtas com o prefixo WWV, enviam regularmente em *broadcast* um pulso no início de cada segundo UTC.

A precisão fornecida pela WWV é de mais ou menos 1 ms, mas devido às condições da atmosfera, que podem alterar o comprimento do sinal, na prática a precisão é em torno de mais ou menos 10 ms.

Vários satélites da Terra também oferecem o serviço UTC. O satélite GEOS (*Geostationary Environment Operational Satellite*) pode informar o tempo UTC com precisão de 0,5 ms, havendo outros satélites que podem fornecer o tempo com uma precisão ainda mais alta.

Para se obter o tempo UTC, seja por meio das transmissões em ondas curtas ou das emissões dos satélites, é preciso definir as posições relativas entre o transmissor e o receptor, de maneira a compensar o retardo de propagação do sinal. Existem dispositivos para a recepção dos sinais enviados por ondas de rádio ou por satélite.

Segundo (TANENBAUM, 1995), para sincronizar computadores com o sinal UTC existem duas regras que devem ser observadas. Se o tempo proporcionado por um serviço de tempo, como o sinal UTC, for maior que o relógio atual de um computador C, então será necessário adiantar o relógio do computador. Quando o computador está adiantado com relação ao serviço de tempo, não será uma boa solução atrasar o relógio do computador C, pois isso poderá confundir algumas aplicações ou processos que estão utilizando o relógio do computador para sincronização. A solução para essa situação é fazer com que o relógio do computador C avance mais lentamente até sincronizar com o servidor do tempo. Isso não pode ser feito fisicamente, mas é possível mudar as rotinas que respondem às interrupções de relógio para que elas incrementem o contador de relógio em freqüências alternadas. Como exemplo, seja S o tempo dado por um relógio local de um computador e seja H o tempo obtido através de um relógio físico externo. Seja d o fator de compensação, tal que

$$S(t) = H(t) + d(t) \tag{1}$$

A forma mais simples para d modificar S continuamente é uma função linear do relógio físico:

$$d(t) = aH(t) + b \tag{2}$$

onde a e b são constantes a serem encontradas. Substituindo d na primeira equação 1, obtém-se:

$$S(t) = H(t) + aH(t) + b = (1 + a)H(t) + b \tag{3}$$

Seja Rl o valor do relógio local quando $H = h$, e seja T o tempo real. Têm-se que $Rl > T$ ou $T > Rl$. Se S precisa alcançar o tempo real, após N oscilações do relógio local, têm-se:

$$Rl = (1 + a)h + b \tag{4}$$

e

$$T = (1 + a)(h + N) + b \tag{5}$$

E resolvendo essa equação obtêm-se:

$$a = (T - Rl)/N \tag{6}$$

e

$$b = Rl - (1 + a)h. \tag{7}$$

Se em um Sistema Distribuído houver uma máquina com receptor WWV, o problema da sincronização dos relógios resume-se em manter todas as demais máquinas do sistema em sincronismo com ela.

2.1 Algoritmo de Cristian

O algoritmo de Cristian trabalha utilizando sistemas que possuem uma máquina com receptor WWV. A máquina com o receptor WWV será denominada de servidor de relógio *time server*.

Periodicamente, cada máquina envia uma mensagem para o servidor de relógio pedindo pelo valor da hora corrente. O servidor responde, o mais rápido que puder, com uma mensagem contendo o seu tempo corrente, C_{UTC} (Figura 1).

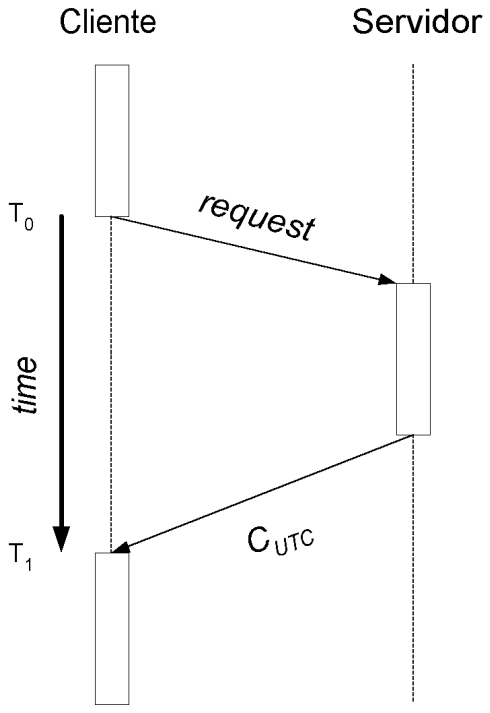

Figura 1: Algoritmo de Cristian

Como uma primeira aproximação, quando o emissor recebe a mensagem de resposta (*reply*), ele pode atualizar o seu clock para C_{UTC}.

Este algoritmo possui dois problemas. O primeiro, e mais complexo, é que o tempo UTC nunca deve estar menor do que a hora local da máquina emissora. Isso pode ocorrer se o relógio do emissor for mais rápido. O problema menor diz respeito ao tempo que a mensagem de resposta leva para chegar ao processo . Uma forma de resolver esse problema é calculando a média entre T_0 e T_1.

3 Estados Locais e Estados Globais

Para se ter uma fotografia completa de uma aplicação distribuída é preciso obter o estado de cada processo que a compõe. Os processos de uma aplicação distribuída têm a sua execução modelada como uma seqüência de eventos, sendo o k-ésimo evento executado pelo processo p_i representado por e_i^k.

Os eventos são classificados como eventos internos e eventos externos ou de comunicação. Os eventos externos representam o envio ou a recepção de uma mensagem, sendo todos os outros eventos considerados internos.

Definição 3.1 (Estado Local) *O estado local σ_i^k de um processo p_i é definido como sendo os valores das variáveis do processo p_i após a execução do seu evento e_i^k.*

A partir da definição de estado local é possível estender esta definição para estado global.

Definição 3.2 (Estado Global) *Estado global é uma n-tupla de estados locais $\Sigma = (\sigma_1, \sigma_2, \ldots, \sigma_n)$, sendo um estado para cada processo.*

Em 1978, Lamport (1978) apresentou um modelo com base na relação de causalidade entre as tarefas de um sistema distribuído e simplificou o problema da falta de um relógio global. O autor observou que em um sistema centralizado o conhecimento do tempo real, na maioria das vezes, é irrelevante para a ordenação dos eventos de uma aplicação, pois esses eventos já estão implicitamente ordenados, visto que um processador realiza uma instrução de cada vez. Assim, Lamport concluiu que também em um sistema distribuído o que importa, na maioria das aplicações, é saber qual a ordem de execução dos eventos e não o tempo real de cada um deles. O que Lamport propôs então, foi a discretização do tempo, ou seja, um sistema distribuído deve ser visto como uma sucessão de eventos discretos, em seqüência, tornando possível a construção de uma relação que captura causalidade entre os eventos. A esta relação Lamport denominou como "Precedência Causal".

Definição 3.3 (Precedência Causal) *A expressão $a \rightarrow b$ é lida como "**a** precede **b**" e significa que todos os processos concordam com o fato de primeiro acontecer o evento **a** e depois ocorrer o evento **b**. A relação de causa e efeito ocorre se:*

1. ***a*** *e* ***b*** *são eventos no mesmo processo e se* ***a*** *ocorre antes de* ***b****;*

2. ***a*** *é o evento do envio de uma mensagem por um processo e* ***b*** *é o evento da mesma mensagem sendo recebida pelo processo de destino.*

A relação de precedência causal é uma relação transitiva, isto é, se $a \rightarrow b$ e $b \rightarrow c$, então $a \rightarrow c$. Além disso, se dois eventos x e y acontecerem em processos diferentes e não trocarem mensagens entre si, nem mesmo indiretamente, através de um terceiro processo, então, nem $x \rightarrow y$, nem $y \rightarrow x$. Tais eventos são, desta forma, considerados concorrentes ($x \| y$ ou $y \| x$).

Pode-se verificar no diagrama da figura 2 que o evento e_2^1 antecede o evento e_3^6, uma vez que existe o caminho composto pelos eventos e_2^1, e_1^2, e_1^3, e_1^4, e_1^5, e_3^6, criando uma relação de dependência entre eles. Em adição, os conceitos de estado local e estado global também podem ser observados. O estado do sistema após a execução dos eventos representados pela tupla $\{e_1^2, e_2^1, e_3^2\}$ é um dos estados globais da computação que o diagrama representa.

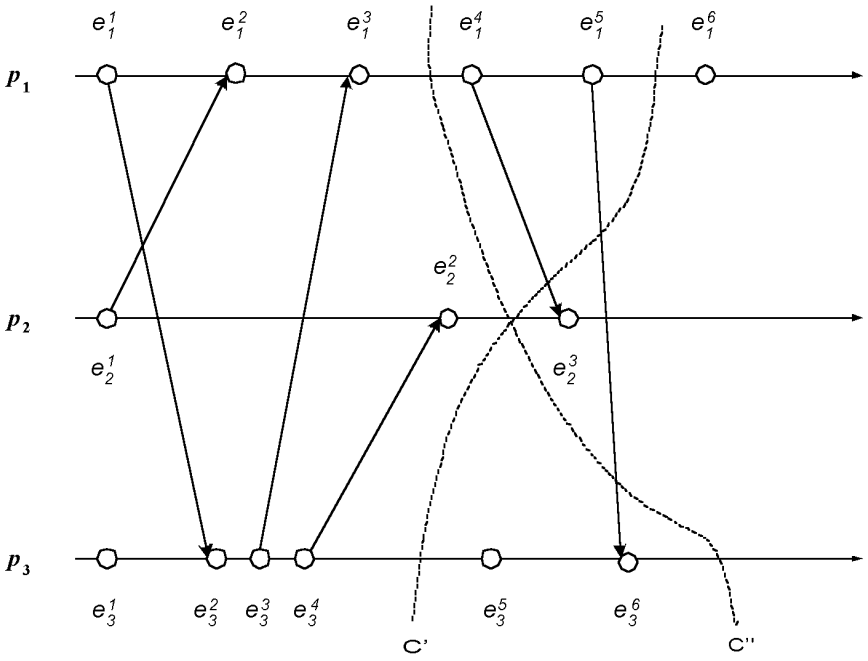

Figura 2: Diagrama de espaço×tempo com dois cortes em uma computação distribuída (BABAOGLU; MARZULLO, 1993)

4 Cortes Globais Consistentes

A definição de corte global consistente é importante para se identificar quando um estado global obtido do sistema é também consistente, uma vez que a observação desse sistema pode ser obsoleta ou mesmo inválida para um observador externo.

Definição 4.1 (Corte) *Um corte é um subconjunto C dos eventos executados pela aplicação distribuída.*

O conjunto contendo o último evento de cada processo contido no corte é denominado "fronteira do corte". A figura 2 contém dois cortes C' e C'' correspondendo as tuplas $\{e_1^5, e_2^2, e_3^4\}$ e $\{e_1^3, e_2^2, e_3^6\}$, respectivamente.

Definição 4.2 (Corte Consistente) *Um corte* $\boldsymbol{C}$ *é consistente se para todo evento* $\boldsymbol{e}$ *e* $\boldsymbol{e'}$*:*

$$(e \in C) \wedge (e' \rightarrow e) \Rightarrow e' \in C$$

Um corte consistente é fechado à esquerda sobre a relação de precedência causal. Toda a seta que intercepta o corte tem sua origem à esquerda. Dessa forma, na figura 2, o corte C' é consistente e o corte C'' não é consistente, uma vez que o evento e_3^6 corresponde ao recebimento de uma mensagem cujo respectivo comando de envio (evento e_1^5) não faz parte do corte.

A partir da definição de corte consistente, obtém-se o conceito de "Estado Global Consistente", que corresponde a um corte global consistente. Na figura 2, o corte C' é um corte global consistente e, por conseqüência, o estado global da aplicação é consistente para um observador externo.

5 Relógios Lógicos

Os relógios dos computadores são dispositivos físicos que geram interrupções com freqüência contínua. A saída dessas interrupções pode ser lida por um *software* que traduz o número de interrupções para um valor do tempo real. Este número pode ser usado para marcar qualquer evento dos processos que estão executando no computador. As aplicações que estão interessadas somente na ordem dos eventos e não no tempo real em que eles ocorrem, utilizam apenas o valor do contador, como é o caso da simulação distribuída.

Um relógio lógico mede o tempo discreto, ou seja, um contador acumula o número de eventos ocorridos entre um evento de referência e um outro evento e a sincronização dos relógios lógicos ocorre seguindo a relação de precedência causal entre os eventos (definição 3.3) .

Definição 5.1 (Relógio Lógico) *É uma função* $\boldsymbol{C}$ *que mapeia um evento* $\boldsymbol{e}$ *em um sistema distribuído para um domínio de tempo* $\boldsymbol{T}$*, denotado como* $\boldsymbol{C(e)}$ *e chamado de timestamp de* $\boldsymbol{e}$*, sendo definida como*

$$C : H \rightarrow T$$

tal que a seguinte propriedade seja satisfeita:

$$e_1 \rightarrow e_2 \Rightarrow C(e_1) < C(e_2)$$

Essa propriedade é chamada de condição para consistência do relógio. O sistema é considerado fortemente consistente quando T e C satisfazem a condição:

$$e_1 \rightarrow e_2 \Leftrightarrow C(e_1) < C(e_2)$$

Para implementar um sistema utilizando relógios lógicos é necessário manter uma estrutura de dados local para todos os processos. Esta estrutura tem o objetivo de representar o tempo lógico do processo. Além disso, é necessário um protocolo, ou seja, um conjunto de regras para atualizar a estrutura de dados sem permitir que o relógio lógico perca a sua condição de consistência.

Existem, basicamente, duas regras que o protocolo deve implementar:

R1 Define como o relógio lógico local deve ser atualizado por um processo quando ele executa um evento (envio ou recebimento de uma mensagem e evento interno).

R2 Define como um processo atualiza o seu relógio global para manter sua visão de tempo global e progresso global, especificando qual informação sobre o tempo lógico deve ser anexada em uma mensagem e como esta informação é usada pelo processo que recebe a mensagem para atualizar sua visão do tempo global.

Sistemas que utilizam relógios lógicos podem diferir na forma de representar o tempo lógico e também no protocolo para atualizar os seus relógios. Entretanto, todos os sistemas de relógios lógicos implementam as duas regras e, conseqüentemente, garantem a propriedade fundamental de consistência de relógio associada à causalidade (RAYNAL; SINGHAL, 1995).

O tempo discreto proposto por Lamport (1978) apresenta um algoritmo simples, com base nas duas regras citadas, para permitir que processadores mantenham um relógio lógico. Cada processo p_i mantém uma variável inteira C_i para armazenar o seu tempo lógico local. As regras para atualizar os relógios são:

R1 Antes de tratar um evento, o processo p_i executa a instrução a seguir, onde d é a evolução escalar do tempo:

$$C_i = C_i + d \qquad (d > 0)$$

R2 Cada mensagem enviada pelo processo p_i para o processo p_j é rotulada com o valor do relógio local do processo emissor. Quando p_j recebe a mensagem com *timestamp* C_{msg}, ele executa as seguintes ações:

1. $C_j = max(C_j, C_{msg})$.
2. Aplicar regra 1.
3. Liberar a mensagem.

O diagrama de espaço×tempo da figura 3 apresenta um exemplo da execução de três processos ordenados pelas das regras de um relógio lógico com evolução escalar igual a 1 ($d = 1$). Se o valor de d é sempre igual a 1, o relógio lógico apresenta uma interessante propriedade: se um evento e possui um *timestamp* h, então $h - 1$ representa a duração lógica mínima, contada em unidades de eventos, necessária para que o evento e ocorra, ou seja, pelo menos $h - 1$ eventos foram produzidos, seqüencialmente, antes do evento e, independente dos processos que os produziram.

Sistemas que utilizam relógios lógicos através de variáveis inteiras não são fortemente consistentes, isto é, para quaisquer dois eventos e_1 e e_2,

$$C(e_1) < C(e_2) \not\Rightarrow e_1 \rightarrow e_2.$$

Como exemplo, pode-se observar na figura 3 que os eventos a e b não possuem relação de dependência entre si, mas a relação $C(a) < C(b)$ é verdadeira.

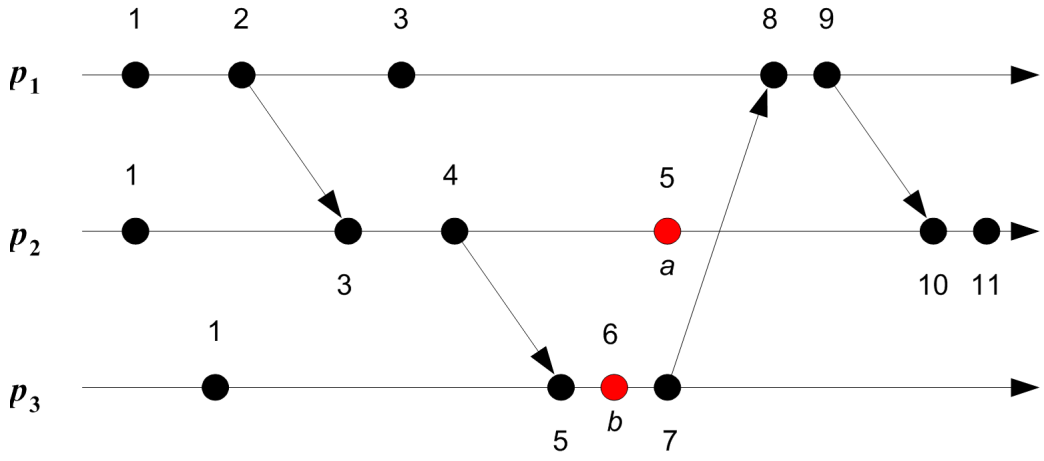

Figura 3: Evolução do tempo escalar

6 Relógios Vetoriais

Os relógios vetoriais foram desenvolvidos, independentemente, por Fidge (1991), Mattern (1989) e Schmuck (1988). Neste tipo de relógio, o tempo é representado por um vetor de inteiros não-negativos de n-dimensões. Cada processo p_i mantém um vetor $vl_i[1..n]$, onde $vl_i[i]$ é o relógio lógico local de p_i e descreve a evolução do tempo lógico no processo p_i. No estado inicial de cada processo, os respectivos vetores possuem todas as posições valendo zero. $vl_i[j]$ representa a última informação conhecida pelo processo p_i do tempo

local do processo p_j, ou seja, é o número de eventos de p_j que precede causalmente os eventos de p_i. Se $vl_i[j] = x$, então o processo p_i sabe que o tempo local no processo p_j progrediu até x. $vl_i[i]$ conta o número de eventos que p_i executou até e_i. Como a quantidade de informações armazenadas pelo relógio vetorial é maior do que aquelas armazenadas pelo relógio lógico é possível detectar uma forte condição de relógio em sua estrutura.

Há, igualmente, duas regras de atualização para os processos que utilizam relógios vetoriais:

R1 Antes de executar um evento, o processo p_i atualiza o seu relógio local através da instrução:

$$vl_i[i] = vl_i[i] + d \qquad (d > 0)$$

R2 Em cada mensagem m enviada pelo processo p_i para o processo p_j, o relógio vetorial é anexado. Quando p_j recebe a mensagem, as seguintes ações são empreendidas:

1. Atualizar o relógio local com a seguinte instrução:

$$1 \leq k \leq n : vl_j[k] = max(vl_i[k], vl_j[k])$$

2. Aplicar a regra 1.
3. Liberar a mensagem.

Em sistemas que utilizam relógios vetoriais, o *timestamp* associado a um evento é o valor do relógio vetorial do seu processo quando o evento é executado.

Segundo Babaoglu e Marzullo (1993), os relógios vetoriais apresentam propriedades importantes para a ordenação de eventos em uma aplicação distribuída. A primeira delas é a forte condição de relógio. Dado dois vetores de números naturais n-dimensionais V e V', define-se a relação "menor que" ($<$) entre eles da seguinte forma:

$$V < V' \Leftrightarrow (V \neq V') \wedge (\forall k : 1 \leq k \leq n : V[k] \leq V'[k]).$$

Na figura 3 foi observado, através dos eventos a e b, que não era possível identificar uma forte condição de relógio. No entanto, se a mesma aplicação fosse implementada utilizando relógios vetoriais, os valores correspondentes aos eventos a e b seriam, respectivamente, $[2, 4, 0]$ e $[2, 3, 3]$ (figura 4). Como $a \not\rightarrow b$, então, $vl(a)$ não pode ser menor que $vl(b)$ e vice-versa.

Por construção, se um evento e_i do processo p_i antecede o evento e_j do processo p_j, sendo $i \neq j$, então $vl_i[i] \leq vl_j[i]$. A condição $vl_i[i] = vl_j[i]$ é possível e representa a situação onde e_i é o último evento de p_i, que precede causalmente e_j de p_j (e_i deve ser um evento de envio de mensagem). É possível identificar esta situação na figura 4, através

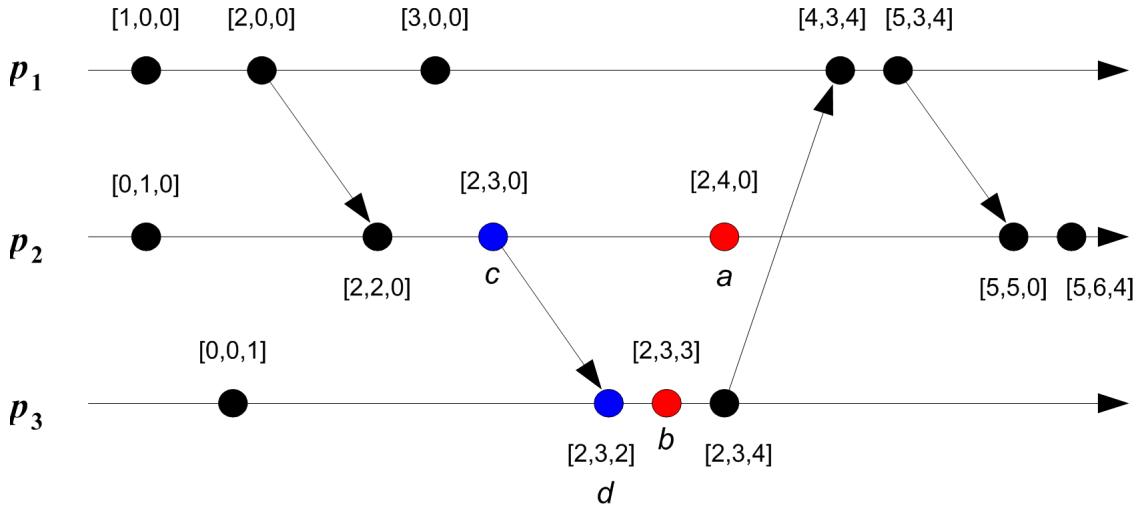

Figura 4: Evolução dos mesmos processos da figura 3 utilizando um relógio vetorial

dos eventos c e d, pois $vl_c[2] = vl_d[2] = 3$. Neste exemplo, destaca-se o teste na posição dois do vetor, correspondendo ao índice do processo que enviou a mensagem.

Uma vez que os vetores proporcionam um mecanismo com forte condição de relógio, de maneira direta, obtém-se um meio de verificar se dois eventos são concorrentes. Dado o evento e_i do processo p_i e o evento e_j do processo p_j, então,

$$e_i \| e_j \Leftrightarrow (vl_i[i] > vl_j[i]) \wedge (vl_j[j] > vl_i[j]).$$

Outra característica interessante dos relógios vetoriais é a possibilidade de verificar se dois eventos podem pertencer a mesma fronteira de um corte consistente. Dois eventos, e_i do processo p_i e e_j do processo p_j com $i \neq j$, não poderão fazer parte do mesmo corte consistente, se e somente se,

$$(vl_i[i] < vl_j[i]) \vee (vl_j[j] < vl_i[j]).$$

Os termos da disjunção caracterizam as duas possibilidades para o corte incluir um evento de recebimento de mensagem (*receive*) sem o respectivo evento de envio (*send*). Dessa forma, um corte é consistente se, em sua fronteira, a condição acima for falsa.

Finalmente, é possível ainda identificar o número de eventos que antecede causalmente um determinado evento. Dados o evento e_i do processo p_i e seu relógio vetorial $vl_i(e_i)$, o número de eventos e, tal que $e \rightarrow e_i$, é dado por:

$$\sharp(e_i) = (\sum_{j=1}^{n} vl_i(e_i)[j]) - 1.$$

Na figura 4, pode-se verificar que o número de eventos que antecede o evento b é 7, pois:

$$\sharp(b) = (\sum_{j=1}^{n} vl_b[j]) - 1 = (vl_b[1] + vl_b[2] + vl_b[3]) - 1 = (2 + 3 + 3) - 1 = 7.$$

Usualmente, dependendo da aplicação distribuída, somente um subconjunto de eventos produzidos é necessário para a análise de suas dependências. Assim, não é preciso detectar a relação causal entre todos os eventos, mas apenas em um subconjunto (ANCEAUME; HéLARY; RAYNAL, 2002). Um exemplo é a localização das dependências entre *checkpoints* com objetivo de identificar se o conjunto é consistente. Esta análise será realizada nas próximas seções.

7 Exclusão mútua

Sistemas envolvendo múltiplos processos são frequentemente mais facilmente programados utilizando regiões críticas. Quando um processo deve ler ou atualizar estruturas de dados compartilhadas, ele primeiro tenta garantir exclusão mútua para o acesso a região crítica.

Em um ambiente centralizado a tarefa de impor a exclusão mútua é facilitada pelo uso de semáforos ou monitores. Em ambientes distribuídos impor a exclusão mútua exige cuidado maior.

7.1 Um algoritmo centralizado

A maneira mais direta de implementar exclusão mútua em um sistema distribuído é simular o comportamento de um sistema com um único processador. Um processo é eleito coordenador, geralmente executando em uma máquina com o endereço de rede mais alto. Quando um processo deseja entrar na região crítica, ele envia uma mensagem de solicitação para o coordenador informando qual a região crítica que ele deseja entrar e pedindo permissão para fazê-lo. Se nenhum outro processo estiver na região crítica, o processo coordenador envia uma mensagem de resposta (*reply* garantindo permissão ao processo. Quando a mensagem chega o processo entra na região crítica.

Se existir um outro processo na região crítica, o processo coordenador pode tomar duas atitudes (dependendo da implementação do sistema):

1. Enviar uma mensagem *reply* com o conteúdo *permission denied*.
2. Não enviar resposta enquanto a região crítica não estiver disponível.

Quando um processo sai da região crítica ele envia uma mensagem avisando o processo coordenador. Se houver processo aguardando, o coordenador retira o primeiro da fila e envia uma mensagem liberando o processo.

É fácil verificar que esse mecanismo impõe a exclusão mútua, entretanto algoritmos centralizados podem se tornar pontos de falha e de gargalo da rede.

7.2 Um algoritmo distribuído

Um algoritmo distribuído pode ser alcançado com a ordenação total dos eventos proposta por Lamport (1978).

A seguir será descrito o algoritmo apresentado por Ricart e Agrawala (1981).

Quando um processo quer entrar na região crítica, ele constrói uma mensagem contendo o nome da região crítica que ele deseja utilizar, o número de identificação do processo, e o tempo corrente. Essa mensagem é enviada para os outros processos, inclusive ele próprio. O envio da mensagem é assumido como confiável, ou seja, toda mensagem terá confirmação.

Quando um processo recebe uma mensagem de requisição *request* de outro processo, a sua ação depende de seu estado com respeito à região crítica descrita na mensagem.

Três casos devem ser distinguidos:

1. Se o receptor não está na região crítica e não deseja entrar nela, ele envia de volta uma mensagem de OK para o emissor.

2. Se o receptor está na região crítica, ele não responde a mensagem, ao contrário, ele coloca a requisição em um fila de espera.

3. Se o receptor quer entrar na região crítica, mas ainda não conseguiu, ele compara o *timestamp* da mensagem que chegou com o *timestamp* da mensagem que ele enviou. A menor vence. Se a mensagem que chegou é menor, o receptor envia uma mensagem OK para o processo emissor. Se a sua mensagem é a que possui o menor *timestamp*, o receptor coloca na fila a mensagem de requisição e não responde ao emissor.

Após enviar todas as mensagens de requisição da região crítica, o processo fica bloqueado aguardando a resposta de todos os outros processos. Assim que todas as mensagens de permissão chegarem, ele pode entrar na região crítica.

Quando o processo sai da região crítica ele envia uma mensagem OK para todos os processos em sua fila de espera e retira-os da fila.

7.3 Um algoritmo *token ring*

Uma abordagem diferente é utilizar um anel lógico envolvendo os processos de um sistema distribuído (TANENBAUM, 1995). Cada processo está associado a uma posição do anel. As posições do anel podem ser alocadas na ordem numérica dos endereços de rede.

Quando o anel é inicializado, o processo 0 está de posse do *token*. O *token* circula pelo anel. Ele passa do processo k para o processo $k + 1$, em comunicações ponto-a-ponto.

Quando um processo adquire o *token* de seu vizinho, se ele estiver tentando entrar na região crítica, ele entra na região crítica, realiza o seu trabalho e sai da região crítica. Após ter saído da região crítica, ele passa o *token* para o próximo processo do anel. Não é permitido entrar na região crítica sem a posse do *token*. Se um processo recebe o *token* e não está interessado em entrar na região crítica, ele imediatamente passa o *token* adiante.

A exatidão desse algoritmo é evidente. Somente um processo pode possuir o *token* em um determinado instante, conseqüentemente, somente um processo pode estar na região crítica. Assim, o *token* circula entre os processo em uma ordem bem-definida, e *starvations* nunca acontecem, pois, uma vez que um processo decide entrar na região crítica, no pior caso ele deverá aguardar pela utilização da região crítica por todos os outros processos.

8 Algoritmos para eleição

Muitos algoritmos distribuídos requerem um processo para agir como coordenador, inicializador ou alguma outra atividade em especial. Algumas situações que exigem um coordenador já foram vistas como, por exemplo, o algoritmo centralizado de exclusão mútua.

Nestas situações é necessário um algoritmo para escolher qual processo deve assumir a função de coordenador. Se todos os processos são exatamente do mesmo tipo, sem nenhuma característica que os distingam dos demais, não existe uma forma de selecionar um deles em especial. Conseqüentemente, assume-se que cada processo possui um número único, por exemplo seu endereço de rede.

Em geral, algoritmos de eleição tentam localizar o processo com o maior número e designá-lo como coordenador. Em adição, assume-se que todos os processos conhecem os números dos outros processos.

O objetivo de um algoritmo de eleição é garantir que quando uma eleição termina todos os processos concordam com a decisão de quem é o novo coordenador.

8.1 O algoritmo *Bully*

No algoritmo *bully* (TANENBAUM, 1995), a eleição inicia quando um processo P nota que o coordenador não responde a uma mensagem de requisição. Este processo realiza a eleição da seguinte forma:

1. P envia uma mensagem ELECTION para todos os processos com números maiores.
2. Se nenhum processo responder, P vence a eleição e torna-se o coordenador.
3. Se algum processo responde a mensagem, a disputa termina, ou seja, a tarefa de P termina.

A qualquer momento um processo pode obter uma mensagem de eleição (`ELECTION`) de um de seus colegas com números menores. Quando essa mensagem chega, o receptor envia uma mensagem `OK` de volta para o emissor para indicar que ele está vivo. O processo receptor inicia uma nova eleição, a menos que ele já esteja realizando essa tarefa. Eventualmente, esses passos irão ocorrer com todos os processos.

Esse procedimento termina com um processo vencedor e avisando a todos os outros que ele é o novo coordenador. Se um processo que era coordenador e havia falhado retornar, ele inicia uma nova eleição. Se acontecer de seu número ser o maior, ele irá vencer a eleição e retornará a ser o coordenador.

Assim o garoto mais forte na cidade sempre vence. Vem desta imagem o nome do algoritmo *"bully algorithm"*.

Para ilustrar o funcionamento deste algoritmo, a figura 5 apresenta um grupo de oito processos, numerados de 0 até 7. Inicialmente, o processo 7 era o coordenador, mas ele falhou. O processo 4 foi o primeiro a notar essa situação. Assim, ele enviou uma mensagem de eleição (`ELECTION`) para todos os processos maiores do que ele (5, 6 e 7). Os processos 5 e 6 respondem com a mensagem de `OK`.

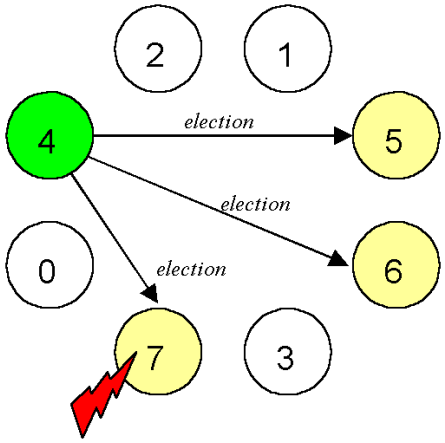

Figura 5: Funcionamento do algoritmo de eleição

Ao receber a primeira das respostas, o processo 4 sabe que a sua tarefa terminou, o que significa que um dos processos que responderam irá ganhar e tornar-se o coordenador. Desta forma o processo 5 e o processo 6 iniciam uma eleição. Cada um enviando mensagens para os processos de números maiores (Fig. 6).

O processo 6 avisa o processo 5 que ele está ativo. Neste ponto o processo 6 sabe que o processo 7 está morto, portanto ele é o vencedor da eleição. Assim, o processo 6 avisa, através do envio de uma mensagem específica (`COORDINATOR`) que ele é novo coordenador. Quando o processo 4 recebe esta mensagem, ele pode continuar a operação que ele estava tentando realizar quando descobriu que o processo 7 estava morto, mas, a partir deste instante, usando o processo 6 como processo coordenador.

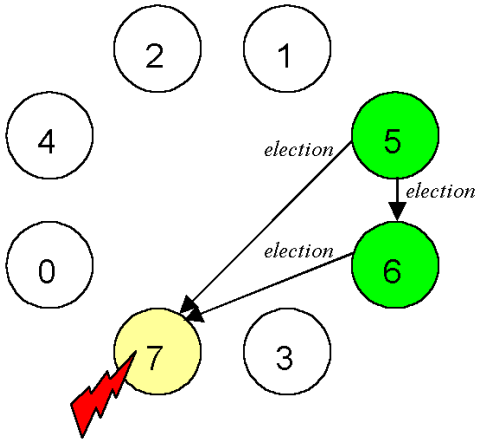

Figura 6: Comportamento dos processos mais fortes

8.2 Um algoritmo de anel

Outro algoritmo de eleição é baseado no uso de um anel, mas sem um *token*. É assumido que os processos estão fisicamente ou logicamente ordenados, isto é, cada processo sabe quem é o seu sucessor. Quando algum processo nota que o coordenador não está funcionando, ele constrói uma mensagem de eleição (`ELECTION`) contendo seu número e envia a mensagem para o seu sucessor. Se o sucessor estiver desativado, o emissor salta para o próximo sucessor, ou seja, o próximo no anel até que um processo em execução seja localizado.

A cada passo o receptor adiciona o número de seu processo na lista da mensagem. Eventualmente, a mensagem retorna ao processo que começou todo o procedimento. Tal processo identifica este evento quando ele recebe uma mensagem contendo seu número.

Neste ponto, o tipo de mensagem é alterado para `COORDINATOR` e circula mais uma vez para informar quem é o novo coordenador (o membro da lista com o maior número) e quem são os membros do novo anel.

A figura 7 apresenta uma ilustração do comportamento deste algoritmo. Neste exemplo, os processos 2 e 5 não conseguem se comunicar com o processo coordenador (processo 7) e iniciam, simultaneamente, o procedimento de eleição. Entretanto, ambos irão chegar ao mesmo resultado, ou seja, o processo 6 deve ser o novo coordenador.

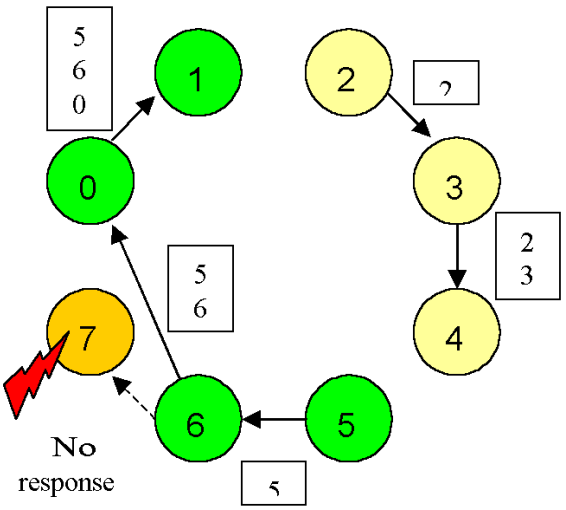

Figura 7: Funcionamento do algoritmo do anel

Referências

ANCEAUME, E.; HéLARY, J. M.; RAYNAL, M. Tracking immediate predecessors in distributed computations. *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, p. 210–219, 2002.

BABAOGLU, O.; MARZULLO, K. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In: MULLENDER, S. (Ed.). *Distributed Systems*. University of Bologna: Addison-Wesley, 1993. p. 55–96.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. *Distributed Systems - Concepts and Design*. 3. ed. Addison-Wesley Publishing Company Inc., 2001. 772 p.

FIDGE, C. Logical time in distributed computing systems. *IEEE Computer*, v. 24, n. 8, p. 28–33, August 1991.

LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, v. 21, n. 7, p. 558–565, July 1978.

MATTERN, F. Virtual time and global states of distributed systems. *Proceedings of the Parallel and Distributed Algorithms*, p. 215–226, 1989.

RAYNAL, M.; SINGHAL, M. Logical time: A way to capture causality in distributed systems. *IEEE Computer*, p. 49–56, 1995. Publication Interne.

RICART, G.; AGRAWALA, A. K. An optimal algorithm for mutual exclusion in computer networks. *Communication of the ACM*, v. 24, n. 1, p. 9–17, January 1981.

SCHMUCK, F. *The Use of Efficient Broadcast in Asynchronous Distributed Systems*. Tese (Doutorado) — Cornel University, 1988. TR88-928.

TANENBAUM, A. S. *Distributed Operating Systems*. Upper Saddle River: Prentice Hall, 1995. 648 p.