Universidade de São Paulo – USP
Instituto de Ciências Matemáticas e de Computação
Sistemas Distribuídos – Prof. Marcos José Santana

Sincronização em Sistemas Distribuídos

Edmilson Marmo Moreira

"Somente o homem que se interessa por tudo é que poderá considerar-se um verdadeiro sucesso."

Henry Ford

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 de um SD não compartilham o mesmo espaço de endereçamento.

Motivação

Controle de Tráfego

Existem 4 componentes:

- 2 semáforos
- 2 centros de controle que se comunicam através de troca de mensagens assíncronas.

As Mensagens podem sofrer atrasos arbitrários.

Cada semáforo pode mudar de **verde** para **vermelho** automaticamente.

Um semáforo somente pode mudar seu estado de **verde** para **vermelho** se ele for autorizado pelo outro semáforo.

A mensagem de autorização é transmitida através de comunicação síncrona.

Exclusão mútua: em qualquer momento, não pode haver mais de um carro no cruzamento.

Cada componente do sistema tem acesso a somente um relógio local.

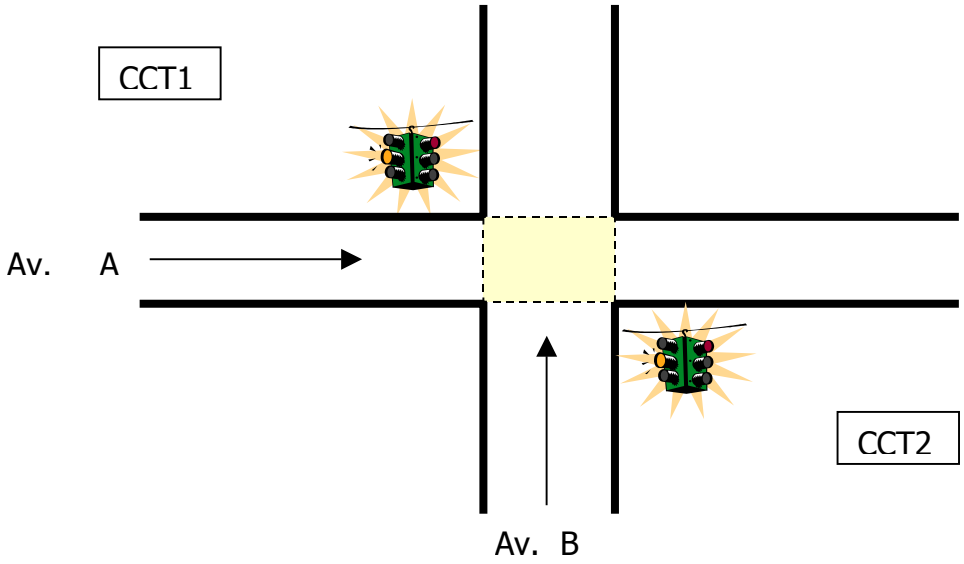
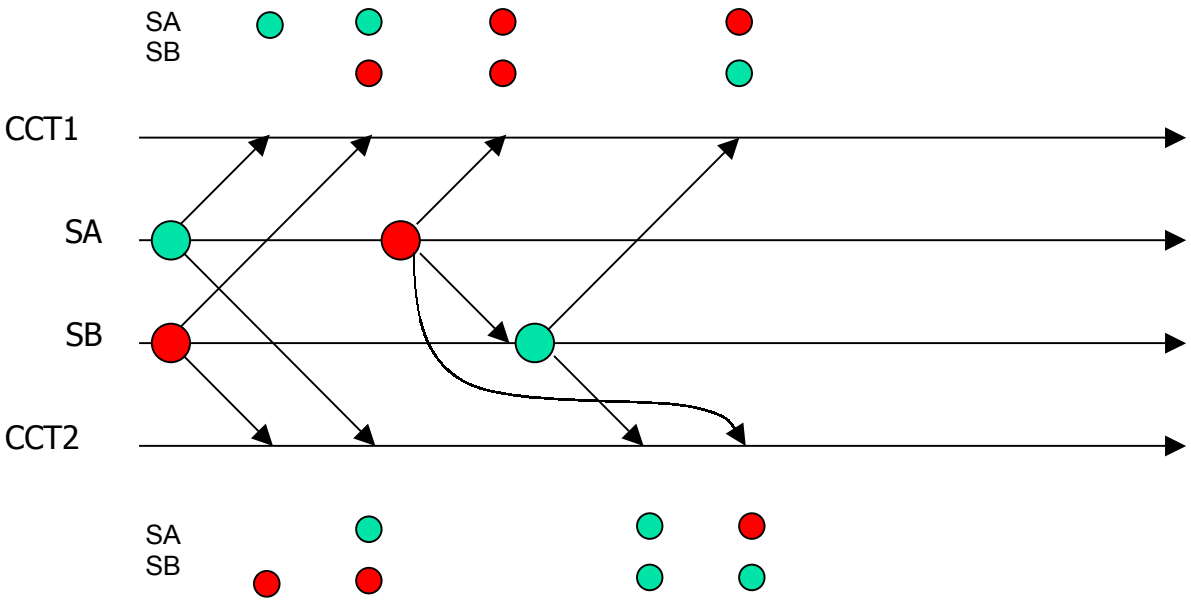


Diagrama de Espaço Tempo



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 ocorrerem.

Os cristais que são utilizados como dispositivos de relógio estão sujeitos a atrasos (***clock drift***).

Esses 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 pela 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.

Curiosidade

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}.

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 (*Coordinated Universal 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.

Para sincronizar computadores com o sinal UTC existem duas regras que devem ser observadas:

1. 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.

2. 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.

3 Relógios Lógicos

Um relógio real registra o tempo real, uma quantidade contínua, ilimitada, homogênea.

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.

Para se obter a sincronização de relógios lógicos, Lamport definiu uma relação de **causa e efeito** (**relação de precedência**) entre os eventos.

Relação de precedência

A expressão $a \rightarrow b$ é lida como "***a acontece antes de 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 pode ser observada em duas situações:

- Se ***a*** e ***b*** são eventos no mesmo processo, e se ***a*** ocorre antes de ***b***, então $a \rightarrow b$ é verdadeira.

- Se ***a*** é o evento de uma mensagem sendo enviada para um processo, e ***b*** é o evento da mesma mensagem sendo recebida por um outro processo, então a relação $a \rightarrow b$ também é verdadeira, pois uma mensagem não pode ser recebida antes de ter sido enviada.

Além disso, a relação de causalidade é uma **relação transitiva**, de forma que se $a \rightarrow b$ e $b \rightarrow c$, então $a \rightarrow c$.

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 $y \rightarrow x$ são verdadeiros.

Estes eventos são considerados **concorrentes** ($x \parallel y$ ou $y \parallel x$), o que simplesmente significa que nada pode ser dito (ou nada precisa ser dito) a respeito de quando tais eventos ocorreram, ou sobre qual deles ocorreu antes e qual ocorreu depois.

A partir do mecanismo descrito por Lamport se torna claro o conceito de relógio lógico:

Relógio Lógico

Relógio lógico *C* é uma função que mapeia um evento *e* em um sistema distribuído para um domínio de tempo *T*, denotado como *C(e)* e chamado de *timestamp* de *e*, sendo definido como segue:

$$C : H \rightarrow T$$

tal que a seguinte propriedade seja satisfeita:

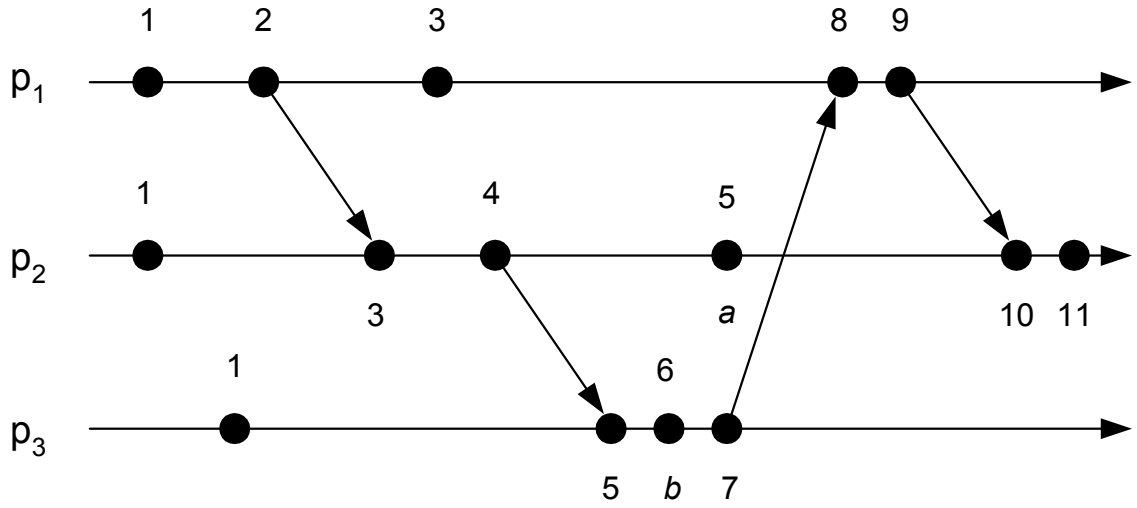
$$e_1 \rightarrow e_2 \Rightarrow C(e_1) < C(e_2).$$

Essa propriedade é chamada de **Condição para Consistência do Relógio**, quando *T* e *C* satisfazem a condição:

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

o sistema é considerado **fortemente consistente**.

Exemplo:



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

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

Por exemplo, na figura acima eventos *a* e *b* não possuem relação de causa e efeito, mas a relação *C(a) < C(b)* é verdadeira.

Relógio Vetorial

Neste tipo de relógio o tempo é representado por um **vetor de inteiros** não-negativos com *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 **0**.

vl_i [j] representa a última informação conhecida pelo processo p_i do tempo local do proceso p_j, ou seja, é o número de eventos de p_j, que precede casualmente o evento e_i de p_i.

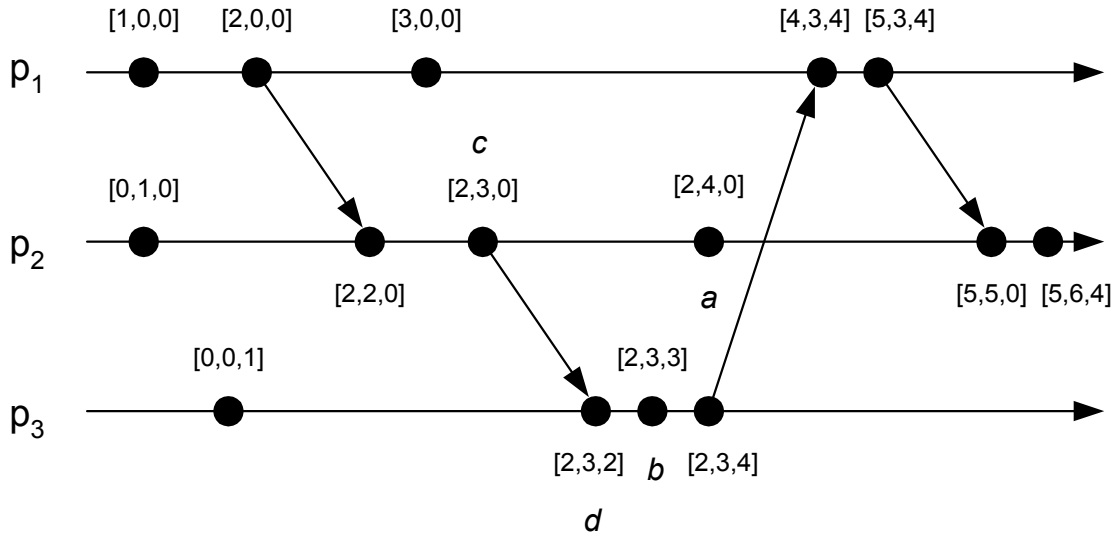
Se vl_i [j] = x, então o processo p_i sabe que o tempo local no processo p_j progrediu até x.

vl_i [i] conta o número de eventos de p_i executou até e_i.

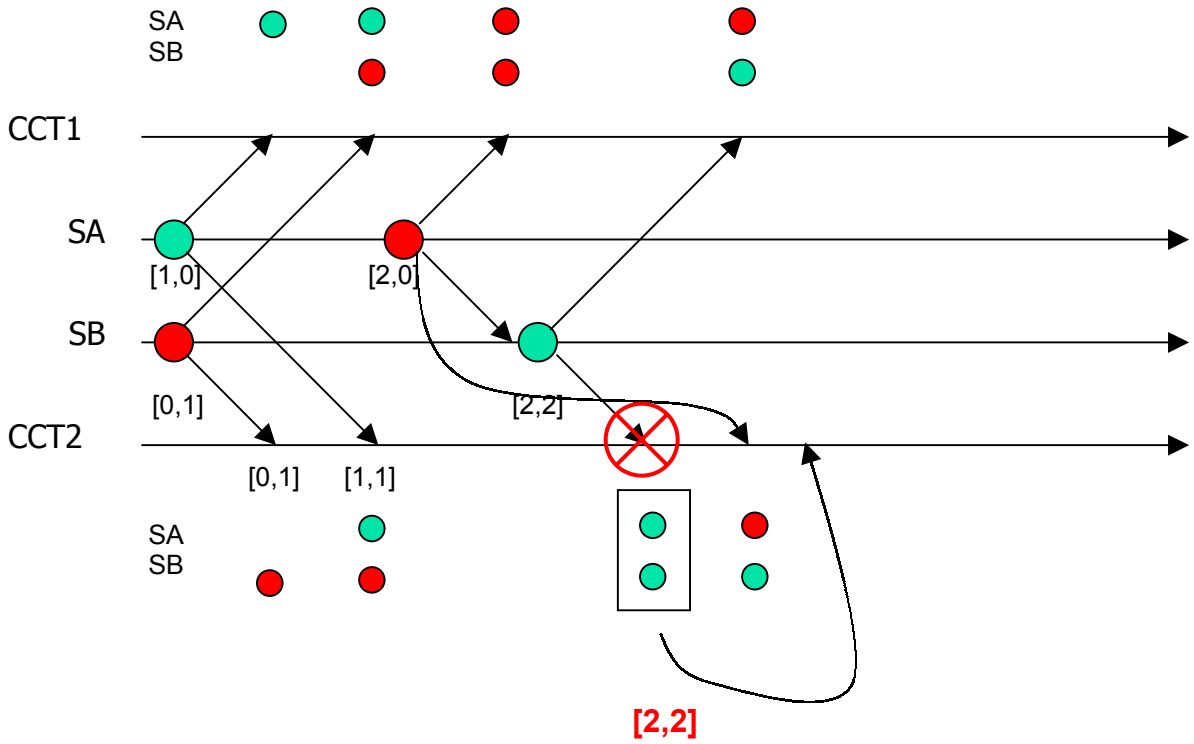
Como a quantidade de informações armazenadas pelo relógio vetorial é maior que o relógio lógico ele é possível detectar uma forte condição de relógio em sua estrutura.

O *timestamp* associado com um evento é o valor do relógio vetorial do seu processo quando o evento é executado.

Exemplo:



Resolvendo o Problema do Controle de Tráfego



4 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 e ou monitores. Em ambientes distribuídos impor a exclusão mútua exige cuidados maiores.

Um Algoritmo Centralizado

A maneira mais direta de implementar exclusão mútua em um SD é simular o comportamento de um sistema com um único processador.

Um processo é eleito coordenador (geralmente executando em uma máquina como o endereço de rede mais alto).

Quando um processo deseja entrar na região crítica, ele envia uma mensagem *request* 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 *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.

Um Algoritmo Distribuído

Um algoritmo distribuído pode ser alcançado com a ordenação total dos eventos de um SD (Lamport, 1978).

A seguir será descrito o algorimto 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 todos 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 *request* de outro processo, a sua ação depende de seu estado com respeito à região crítica descrita na mensagem.

Três casos devem ser distingüidos:

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 um mensagem *OK* para o processo emissor. Se a sua mensagem é a que possui o menor *timestamp,* o receptor coloca na fila a mensagem *request* 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.

Um Algoritmo *Token Ring*

Uma abordagem diferente é utilizar um anel lógico envolvendo os processos de um SD.

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 passe 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.

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.

5 Validação de Predicados Globais

Para solucionar alguns problemas da Computação Distribuída é necessário detectar algumas propriedades globais.

Assim, é importante ter uma visão global e consistente da Computação Distribuída em um determinado instante de tempo.

Um **Estado Local** de um processo é valor dos registradores, variáveis de memória em um determinado instante.

> O estado σ_i^k é o estado local de um processo p_i imediatamente após ter sido executado o evento e_i^k.
>
> O estado σ_i^o o seu estado inicial antes de qualquer evento ser executado.

Um **Estado Global** em um Sistema Distribuído é a união dos estados individuais de cada processo.

> O **Estado Global** é uma *n*-tupla de estados locais $\sum = (\sigma_1, ..., \sigma_n)$ um para cada processo.

A observação remota de um estado pode ser obsoleto, incompleto ou inconsistente.

Estado Global Inconsistente: Se nunca poderia ter sido construído por um observador externo ao sistema.

Exemplos de Problemas para Predicados Globais:

Detecção de *Deadlock*.

Detecção de terminação.

Token perdido,

Depuração e monitoração.

Checkpoints Globais Consistentes

Conceitos Básicos

Histórico Local: de um processo p_i é uma seqüência possível de eventos $h_i=e_i^1 e_i^2 ...$

Histórico Global: é um conjunto $H = h_1 \cup ... \cup h_n$ contendo todos os seus eventos.

> *Em um sistema distribuído assíncrono onde não existe um relógio global, os eventos podem ser ordenados somente com base na noção de "causa e efeito".*

Um **Corte** de uma computação distribuída é um subconjunto C de seu histórico global H e contém um prefixo inicial de cada histórico local.

> Pode-se especificar um corte $C = h_1^{C1} \cup ... \cup h_n^{Cn}$ através de uma tupla natural de números $(c_1, ..., c_2)$ correspondendo à indexação dos últimos eventos de cada processo.

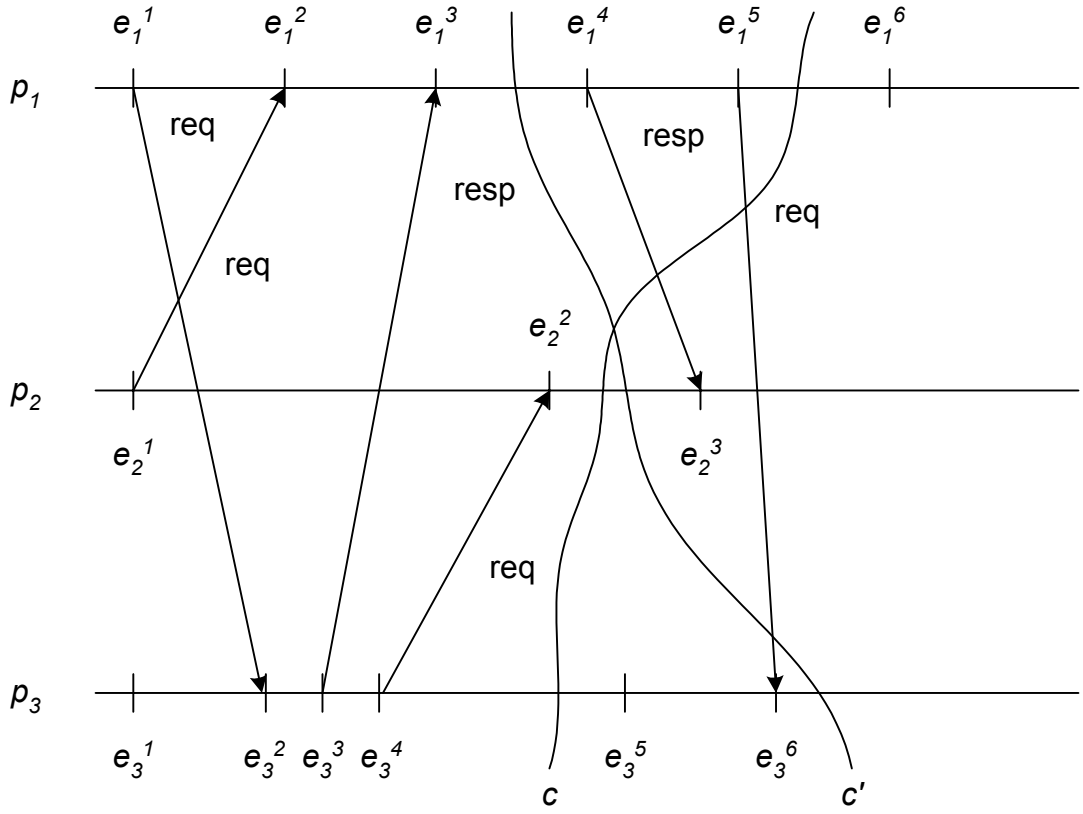
O conjunto dos últimos eventos $(e_1^{C1}, ..., e_n^{Cn})$ incluídos no corte $(c_1, ..., c_n)$ é chamado de **fronteira do corte**.

Cada corte definido pela tupla $(c_1 , ..., c_n)$ tem um estado global correspondente que é $(\sigma_1^{C1}, ..., \sigma_n^{Cn})$.

Um corte C é consistente se para todos os eventos e e e' a expressão abaixo é válida:

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

> *É fácil determinar visualmente se um corte é consistente ou não. Se todas as setas que atravessam o corte têm as suas bases à esquerda do corte e suas pontas à direita do corte então ele é consistente, caso contrário, ele é inconsistente.*



A análise da consistência de cortes é muito importante para a avaliação dos predicados globais, pois

> ***um estado global é consistente se ele estiver associado a um corte global consistente.***

6 Conclusões

O entendimento dos mecanismos de sincronização em SD é fundamental para a implementação de aplicações consistentes e confiáveis.

As soluções baseadas em relógios lógicos são mais genéricas e facilitam a migração de uma aplicação de um SD para outro.

Existem outros mecanismos de relógio que são variações daquelas apresentados nesta nota de aula, como por exemplo, relógios matriciais que são amplamente usados nos sincronismos de banco de dados distribuídos.

Referências

M.Chandy and L.Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. On Computing Systems,* 3(1):63-75, February 1985.

O.Babaoglu and Keith Marzullo. Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms. In S. Mullender, editor, *Distributed Systems,* p. 55-96. Addison-Wesley, 1993.

Leslie Lamport. Time, Clocks and the Ordering of Events in a Distributed Systems, *Communications of the ACM,* 21(7): 558-565, July, 1978.

L. Buzato. Algoritmos Distribuídos e suas Aplicações. *Anais da VI Escola Regional de Informática de São Paulo, SBC*, p. 71-99, Abril, 2001.