

Algoritmos Distribuídos

Introdução^{*}

Antonio Alfredo Ferreira Loureiro

`loureiro@dcc.ufmg.br`

`http://www.dcc.ufmg.br/~loureiro`

^{*}Este material está baseado no capítulo 3 do livro “Distributed Systems”, second edition, Sape Mullender, editor, ACM Press, 1993, que também foi publicado como “Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms”, Özalp Babaoglu e Keith Marzullo, Technical Report UBLCS-93-1, January 1993, Laboratory for Computer Science, University of Bologna, Italy.

Sumário

- Visão geral
- Sistemas de memória distribuída
- Processadores de comunicação
- Sistema distribuído assíncrono
- Computação distribuída

Visão geral

- O que é (informal)
 - Algoritmo + Distribuído (processamento e comunicação)
- Tipos de processamento e comunicação variam
- Características:
 - Cada *processo* tipicamente executa o mesmo algoritmo
 - Os modelos de comunicação variam bastante mas estaremos interessados principalmente em processos que comunicam entre si apenas usando mensagens

Sistemas de memória distribuída

- Passagem de mensagem e sistemas de memória distribuída
 - Conceitos relacionados
- Paradigma “passagem de mensagem” está relacionado com comunicação em diferentes níveis
 - Tipicamente processador e aplicação

Sistemas de memória distribuída

- Sistemas de memória distribuída:
 - Conjunto de processadores interconectados (de acordo com uma topologia) por um conjunto de enlaces de comunicação
 - Não existe compartilhamento de memória
 - Troca de informação é feita através de mensagens
- Topologias típicas das redes de interconexão
 - Ponto-a-ponto: irregular, anel
 - Difusão: barramento
- Programas que executam nesse sistema:
 - Um ou mais programas sequenciais em cada processador

Sistemas de memória distribuída

- No nível de aplicação, tarefas em processadores distintos podem se comunicar também usando compartilhamento de memória
 - Emulado através de passagem de mensagem
- Existem várias outras abordagens propostas na literatura, nem todas práticas
 - Sugestão: façam um levantamento de tais mecanismos e tentem pensar como isso poderia ser feito através de passagem de mensagem

Exemplos de sistemas de memória distribuída

- Redes de computadores de longa distância
 - Arpanet (1969) que deu origem a Internet
- Arquitetura de redes baseada em camadas
- Ciclo importante:

Características do ambiente
afetam

Projeto desses sistemas
que levam ao desenvolvimento de
Novas tecnologias
que afetam

Características do ambiente

Exemplos de sistemas de memória distribuída

- Processamento paralelo:
 - Sistema formado por processadores interconectados (de acordo com uma topologia) e onde existe compartilhamento de memória entre os processadores
- Características:
 - Empregados para resolver o mesmo problema
 - Não escalam bem
- Sistema típico que tem predominado:
 - Sistemas de memória distribuída onde a comunicação entre os processadores é feita ponto-a-ponto
 - Sugestão: Vejam a grande quantidade de arquiteturas e linguagens de programação para esses sistemas

Exemplos de sistemas de memória distribuída

- Redes de estações de trabalho
 - Redes de computadores onde a comunicação é tipicamente difusão
- Suportam outros tipos de serviço devido à proximidade das estações:
 - Compartilhamento de sistemas de arquivos

Processadores de comunicação

- Cenário típico:
 - Duas tarefas, cada uma executando em um processador distinto A e B, comunicam entre si
 - Não existe um enlace de comunicação que ligue diretamente A e B
- Solução:
 - Usar processadores intermediários entre A e B para levar mensagens entre as duas tarefas
- Nesse cenário, processadores executam
 - As próprias tarefas
 - Gerenciam tráfego de mensagens entre processadores
- Situação atual:
 - Processador (hospedeiro): executa tarefas
 - Processador de comunicação: trata mensagens
- Problema importante decorrente:
 - Roteamento e controle de fluxo

Roteamento e controle de fluxo

- Sistema de memória distribuída pode ser modelado por um grafo não dirigido $G_P = (N_P, E_P)$
 - N_P : conjunto de processadores
 - E_P : conjunto de enlaces de comunicação full-duplex
- O roteamento da mensagem (q, Msg) é feito por um processador r usando a função $next_r(q)$
- $R(p, q) \subseteq E_P$: conjunto de enlaces por onde passa uma mensagem de p para q
 - Tipicamente $R(p, q) \neq R(q, p)$

Roteamento e controle de fluxo

- Roteamento pode ser:
 - Fixo ou dinâmico
 - Determinístico ou não
- Determina o projeto da função *next*
- Questão:
 - Que problemas podem surgir para a aplicação a combinação destes esquemas?
- Sugestão:
 - Procure pesquisar/estudar propostas e suas limitações para resolver esses problemas

Sistema distribuído assíncrono

- SD assíncrono formado por:
 - Coleção de processos seqüenciais p_1, p_2, p_n
 - Rede de computadores com canais unidirecionais entre pares de processos para troca de mensagens
 - Canais podem ser confiáveis ou não
 - Mensagens podem ser entregues fora de ordem
- Questões decorrentes:
 - Isto significa que não existem canais bidirecionais?
 - Qual é o modelo matemático que pode representar esse sistema?
 - Como é feita a comunicação entre um par arbitrário de processos?

Sistema distribuído assíncrono

- Objetivo deste modelo:
 - Definir um conjunto “fraco” de suposições
 - Estabelece um custo superior na resolução de problemas. Por quê?
- Modelo mais fraco para um SD é um sistema assíncrono:
 - Processos podem ter velocidades diferentes mas limitadas
 - Mensagens podem ter atrasos diferentes mas limitados
 - Comunicação é o único mecanismo para sincronização de processos no sistema

Computação distribuída

- Descreve a execução de um programa distribuído por uma coleção de processos
- A atividade de cada processo seqüencial é modelada como uma seqüência de eventos
- Evento pode ser:
 - Interno: causa apenas uma mudança do estado local
 - Externo: causa comunicação com outro processo

Sistema distribuído assíncrono

- Comunicação é feita através dos eventos $send(m)$ e $receive(m)$:
 - São eventos correspondentes
 - Esse “casamento” de eventos é feito através da mensagem m
- Questões decorrentes:
 - O que significa eventos correspondentes?
 - O que significa se diferentes processos enviam o mesmo dado para o mesmo processo?

Sistema distribuído assíncrono

- Significado de $send(m)$:
 - Mensagem m é enfileirada num canal de saída para ser transmitida para o processo de destino
- Significado de $receive(m)$:
 - Remove mensagem m de uma fila de entrada no processo destino
 - Processo p deve ter declarado seu desejo em receber uma mensagem e a mensagem deve ter sido recebida
 - O evento só ocorre se a mensagem já chegou
- Atrasos referentes a $receive(m)$:
 - Mensagem é “atrasada” porque o processo não está pronto
 - Processo é “atrasado” porque a mensagem não chegou
- Questões decorrentes:
 - Que problemas podem ocorrer para o processo que recebe mensagens?
 - O que acontece em cada um dos dois tipos de atrasos acima?

Sistema distribuído assíncrono

- Formas de obter passagem de mensagem em uma linguagem de programação:
 - *Remote Procedure Call*
 - Broadcast
 - Transações distribuídas
 - Objetos distribuídos
 - *Distributed shared memory*

Sistema distribuído assíncrono

- História local (*local history*) de um processo p_i durante uma computação é, possivelmente, uma seqüência infinita de eventos

$$h_i = e_i^1 e_i^2 \dots$$

- Observações sobre h_i :
 - Enumeração canônica
 - Representa uma ordem total imposta pela execução seqüencial do processo p_i nos eventos locais

- Seja

$$h_i^k = e_i^1 e_i^2 \dots e_i^k$$

um prefixo de uma história local h_i contendo os primeiros k eventos:

- h_i^0 é a seqüência vazia

Sistema distribuído assíncrono

- História global de uma computação é o conjunto

$$H = h_1 \cup h_2 \cup \dots \cup h_n$$

- H contém todos os eventos que ocorreram no SD
- Uma história local pode ser representada como uma seqüência de eventos ou um conjunto
 - Todos os eventos de uma computação têm um identificador único na enumeração canônica (seqüência)
 - h_i como um conjunto contém exatamente os mesmos eventos de h_i como uma seqüência
- Questões decorrentes:
 - Por que h_i é, possivelmente, uma seqüência infinita?
 - É possível ter uma referência de tempo associada aos eventos de h_i ?
 - E em relação a H (história global)?

Sistema distribuído assíncrono

- Num sistema distribuído assíncrono, eventos de uma computação só podem ser ordenados usando a noção de “causa-e-efeito” (*cause-and-effect*):
 - Dois eventos ocorrem numa certa ordem somente se a ocorrência do primeiro afeta o segundo
 - Isto significa que o fluxo de execução vai do primeiro evento para o segundo
- Cenários para esse fluxo de execução:
 - Os dois eventos estão no mesmo processo
 - Os dois eventos estão em processos diferentes p_i e p_j e estão relacionados por uma troca de mensagem entre p_i e p_j

Sistema distribuído assíncrono

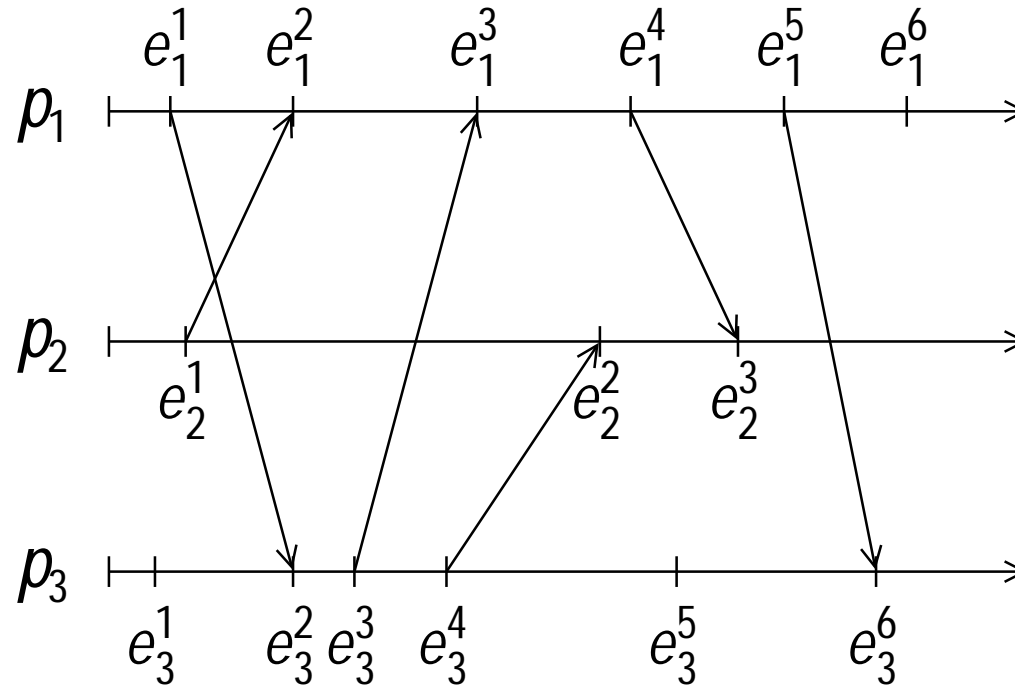
- Conceitos apresentados por Lamport usando a relação binária \rightarrow (*happens-before*)
 1. Se $e_i^k \in h_i$ e $k < l$, então $e_i^k \rightarrow e_i^l$
 2. Se $e_i = \text{send}(m)$ e $e_j = \text{receive}(m)$, então $e_i \rightarrow e_j$
 3. Se $e \rightarrow e'$ e $e' \rightarrow e''$, então $e \rightarrow e''$

Sistema distribuído assíncrono

- Comentários sobre $e \rightarrow e'$:
 - Somente no caso dos eventos $send(m)$ e $receive(m)$ é que a relação “causa-e-efeito” existe claramente
 - Em geral, pode-se dizer que a ocorrência de e' pode ter sido influenciada pelo evento e
- Computação distribuída:
 - POSET (*Partially Ordered Set*) definido pelo par (H, \rightarrow)
 - Todos eventos são rotulados com a identificação canônica
 - Eventos de comunicação também possuem o identificador da mensagem
- Conseqüências da definição de computação distribuída:
 - Ordem total dos eventos em cada processo
 - Correspondência entre eventos $send(m)$ e $receive(m)$

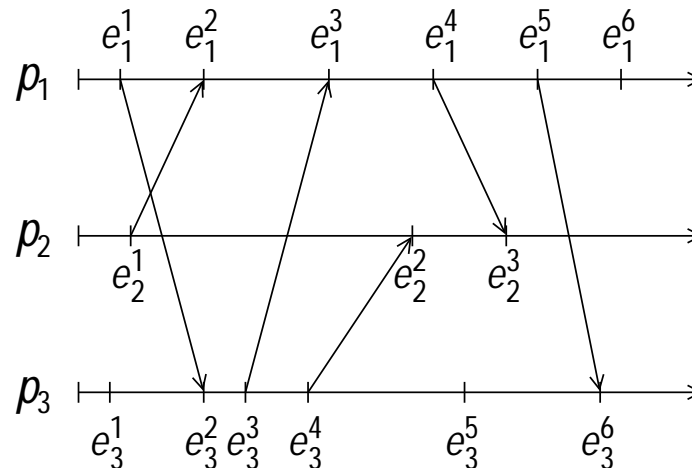
Sistema distribuído assíncrono

- Normalmente, uma computação distribuída é representada graficamente através de um diagrama tempo-espço



Sistema distribuído assíncrono

- Certos eventos de uma história global podem não ser relacionados causalmente
 - É possível que nem $e \rightarrow e'$ nem $e' \rightarrow e$ seja verdade
 - São eventos concorrentes e representados por $e||e'$
- Por exemplo, $e_2^1 \rightarrow e_3^6$, mas $e_2^2||e_3^6$



Estado local de um processo

- Inclui informação “relevante” para o processo como valores de variáveis locais, filas dos canais de entrada e saída, e mensagens já enviadas e recebidas
 - O estado local de um processo poderia incluir o estado do ambiente computacional onde o processo é executado
 - Do ponto de vista prático, está restrito ao espaço de endereçamento do processo
 - O estado do canal pode ser codificado como parte do estado local do processo
- Seja s_i^k o estado local do processo p_i imediatamente após ter executado evento e_i^k
- Seja s_i^0 o estado inicial do processo p_i antes de qualquer evento ter sido executado

Estado global de uma computação distribuída

- Definida como uma n -tupla de estados locais, um para cada processo

$$S = (s_1, s_2, \dots, s_n)$$

- Qualquer propriedade da computação deve ser feita em relação ao estado global
 - Por que isso?

Corte de um computação distribuída

- É um subconjunto C da história global H e contém um prefixo inicial de cada história local
- Um corte pode ser representado por

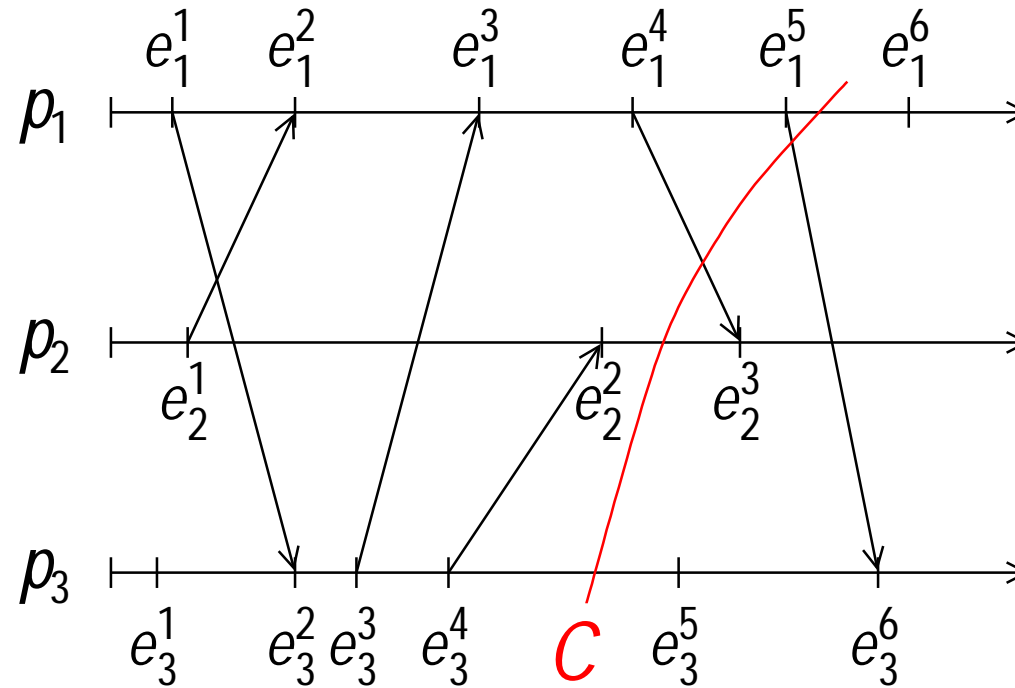
$$C = h_1^{c_1} \cup \dots \cup h_n^{c_n}$$

ou ainda pela tupla de números naturais (c_1, \dots, c_n) , que corresponde ao índice do último evento incluído no corte para cada processo

- O conjunto de últimos eventos $(e_1^{c_1}, \dots, e_n^{c_n})$ incluído no corte (c_1, \dots, c_n) é chamado de fronteira do corte
- Cada corte válido definido por (c_1, \dots, c_n) define um estado global correspondente identificado por $(s_1^{c_1}, \dots, s_n^{c_n})$

Corte de um computação distribuída

- Um corte tem uma representação gráfica como uma partição do diagrama tempo-espaco ao longo do eixo do tempo

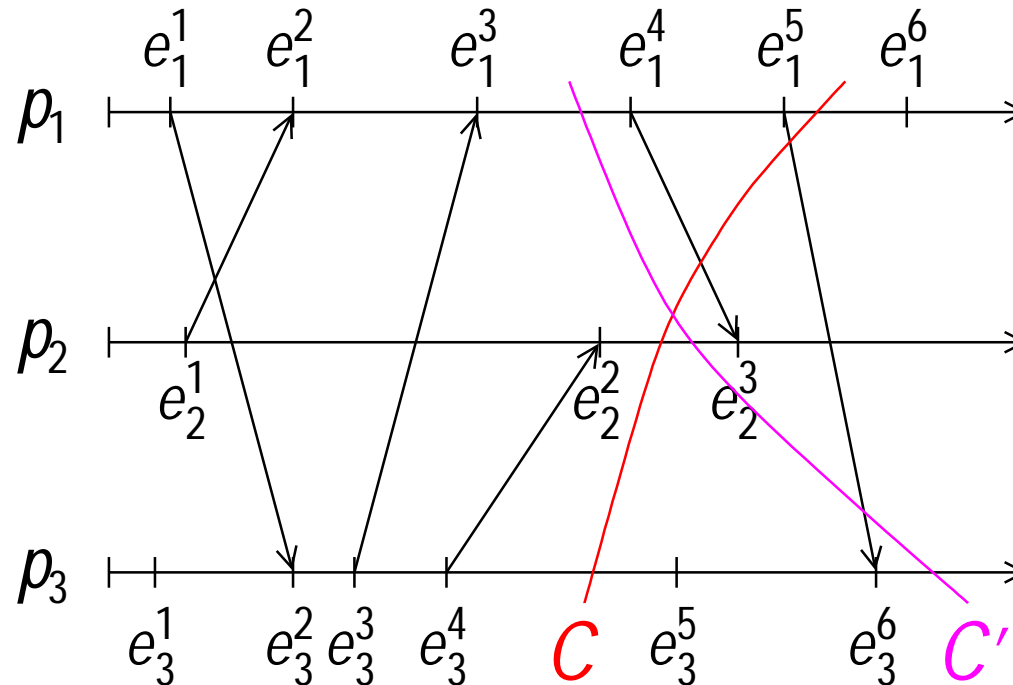


Computação distribuída e ordem total

- Uma computação distribuída é um POSET de eventos
- No entanto, todos os eventos de uma computação real ocorrem numa ordem total
 - Apesar de não existir observador global em sistemas distribuídos
 - No caso de eventos ocorrerem exatamente no mesmo instante do tempo, pode-se assumir que o evento que ocorreu no processo de menor índice ocorreu antes do(s) outro(s) evento(s)
- Uma execução (*run*) de uma computação distribuída é uma ordem total R que inclui todos os eventos da história global e é consistente com cada história local
 - Os eventos de p_i aparecem em R na mesma ordem relativa que aparecem em h_i
- Uma computação distribuída pode ser representada por diferentes execuções

Consistência

- Precedência causal é um mecanismo que permite distinguir um corte consistente C de um corte inconsistente C'
 - Por quê?



Consistência

- Um corte é consistente se para todos os eventos e e e'

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

- Corte consistente é fechado à esquerda para a relação de precedência causal
 - Graficamente, todo o início de uma seta deve estar à esquerda do corte
- Conseqüências:
 - Um estado global consistente corresponde a um corte consistente
 - São estados que podem ter ocorrido durante uma execução e construídos por um observador externo (global)

Consistência

- Corte consistente (e dualmente um estado global consistente) é um conceito fundamental no entendimento de computação distribuída assíncrona
- Similaridades entre tempo escalar e corte consistente:
 - Tempo escalar: indica um instante particular durante uma computação seqüencial
 - Corte consistente: indica um instante durante uma computação distribuída
- Similaridades entre “antes” e “depois” em tempo escalar e corte consistente:
 - Tempo escalar: indica instantes antes e depois de um evento e
 - Corte consistente: um evento e é antes (depois) de um corte C se e está à esquerda (direita) da fronteira de C
- Um predicado só pode ser avaliado em um estado global consistente
 - Por quê?

Consistência

- Uma execução R é consistente se para todos os eventos, $e \rightarrow e'$ implica que e aparece antes de e' em R
 - Ordem total existente em R é uma extensão da ordem parcial definida pela precedência causal
- Uma execução $R = e^1 e^2 \dots$ resulta em uma seqüência de estados globais $S^0 S^1 S^2 \dots$, onde S^0 é o estado global inicial (s_1^0, \dots, s_n^0)

Consistência

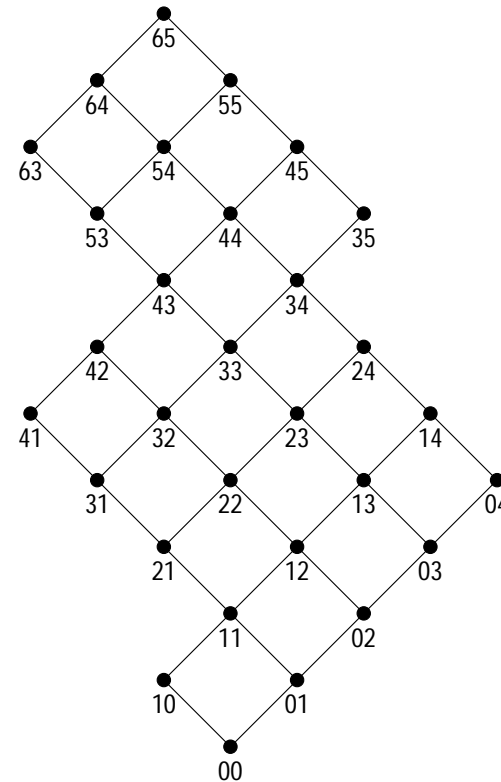
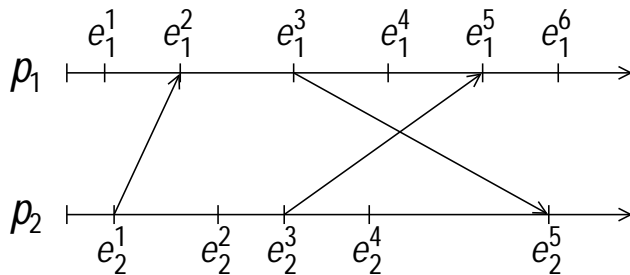
- Se a execução é consistente então os estados globais na seqüência serão todos consistentes
- Cada estado global consistente S^i é obtido do estado anterior S^{i-1} quando algum processo executa o evento e^i
- Para dois desses estados de R , diz-se que S^{i-1} leva a (*leads to*) S^i em R
- \rightsquigarrow_R representa o fecho transitivo da relação \rightsquigarrow numa dada execução R
- O estado global S' é alcançável a partir de S na execução R sse $S \rightsquigarrow_R S'$
- O subscrito R é removido se existe alguma execução tal que S' é alcançável a partir de S , ou seja, $S \rightsquigarrow S'$

Consistência

- O conjunto de todos os estados globais consistentes de uma computação, junto com a relação \rightsquigarrow , define um reticulado (*lattice*), também chamado de Diagrama de Hasse
- Um reticulado consiste de n eixos ortogonais, um eixo para cada processo p_1, \dots, p_n
- A notação SP_{p_1, \dots, p_n} é uma abreviação para o estado global (s^1, \dots, s^n) e $p_1 + \dots + p_n$ é o nível desse estado no reticulado (grafo de estados globais)

Consistência

- Computação distribuída de dois processos e o reticulado correspondente de estados globais



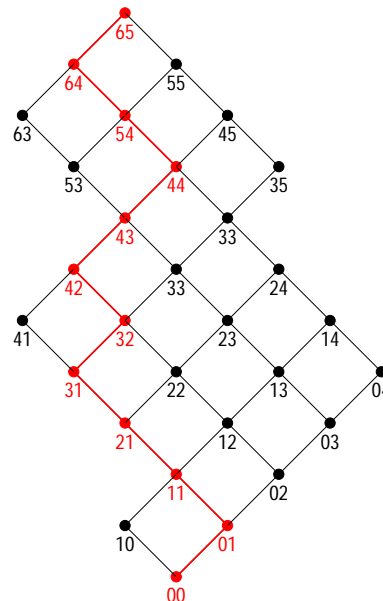
Consistência

- Cada estado global é alcançável a partir do estado global S^{00}
- Um caminho no reticulado é uma seqüência de estados globais, onde cada um difere de um nível
- Cada um desses caminhos corresponde a uma execução consistente da computação distribuída
- Diz-se que a execução passa através dos estados globais incluídos no caminho

Consistência

- Por exemplo, uma possível execução pode passar através da seqüência de estados globais

$S^{00} S^{01} S^{11} S^{21} S^{31} S^{32} S^{42} S^{43} S^{44} S^{54} S^{64} S^{65}$



→ Note que não é possível identificar qual é a execução real da computação

Monitoramento computações distribuídas

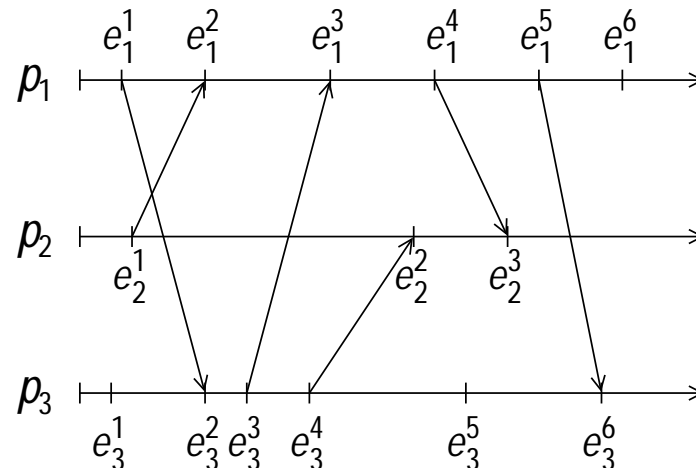
- GPE (*Global Predicate Evaluation*) é o problema de avaliar um predicado Φ em um estado global S de uma computação distribuída
- Um processo, chamado monitor, é responsável por avaliar Φ
- Seja p_M um dos processos p_1, \dots, p_n responsável por avaliar Φ
- Resolver este problema implica em p_M construir um estado global S da computação tal que o predicado Φ seja aplicado
- Observação importante:
 - Eventos executados por causa de p_M não interferem na enumeração canônica dos eventos, ou seja, a computação distribuída de interesse não tem nenhum evento causado pela ação de p_M

Monitoramento computações distribuídas

- Estratégia do processo p_M :
 - Envia para cada processo uma mensagem de $\langle \text{STATE-ENQUIRY} \rangle$
 - p_M tem um papel ativo
- Ao receber essa mensagem, p_i responde com o seu estado local s_i atual
- Ao receber as respostas dos n processos, p_M pode construir o estado global correspondente (s_1, \dots, s_n)
- A posição na história local de cada processo define um corte
 - Esse corte é garantidamente consistente?

Monitoramento computações distribuídas

- Observação importante:
 - Processo monitor é parte do sistema distribuído e sofre as mesmas incertezas dos outros processos de um sistema distribuído assíncrono
- Logo, a solução proposta pode levar a valores de predicados incorretos
- Exemplo, de acordo com o algoritmo proposto, uma possível computação num SD cliente-servidor está mostrada abaixo:



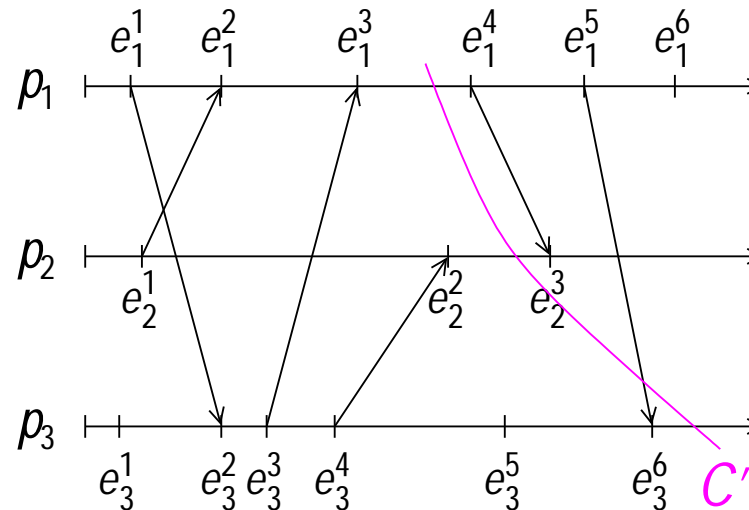
- Processo p_M pode detectar *deadlock*?

Monitoramento computações distribuídas

- Como é feita a detecção de *deadlock* num sistema como esse?
 - Cada processo servidor mantém o estado local contendo os nomes de clientes de quem recebeu requisições e ainda não enviou respostas
 - Processo p_M constrói o *waits-for*⁺ *graph* (WFG⁺) onde os nós correspondem a processos e as arestas ao modelo de bloqueio
 - Nesse grafo, uma aresta é desenhada do nó (processo) i para o nó j se p_j recebeu uma requisição de p_i e ainda não respondeu
 - Observe que WFG⁺ pode ser construído apenas usando os estados locais dos processos
 - Um ciclo no grafo WFG⁺ indica um *deadlock* no sistema, sendo que os nós do ciclo são exatamente aqueles processos envolvidos no *deadlock*
 - Predicado $\Phi =$ “WFG⁺ contém um ciclo” é uma possibilidade para detectar *deadlock*

Monitoramento computações distribuídas

- Suponha que o processo p_M monitore uma computação como descrito
- Suponha que cada processo recebe a mensagem $\langle \text{STATE-ENQUIRY} \rangle$ nos pontos indicados por C'

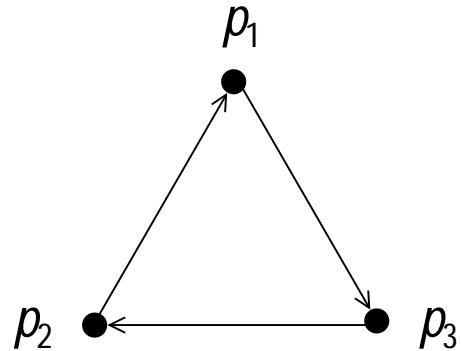


- Nesse caso, os estados reportados por cada processo são:

Processo	Estado
p_1	s_1^3
p_2	s_2^2
p_3	s_3^6

Monitoramento computações distribuídas

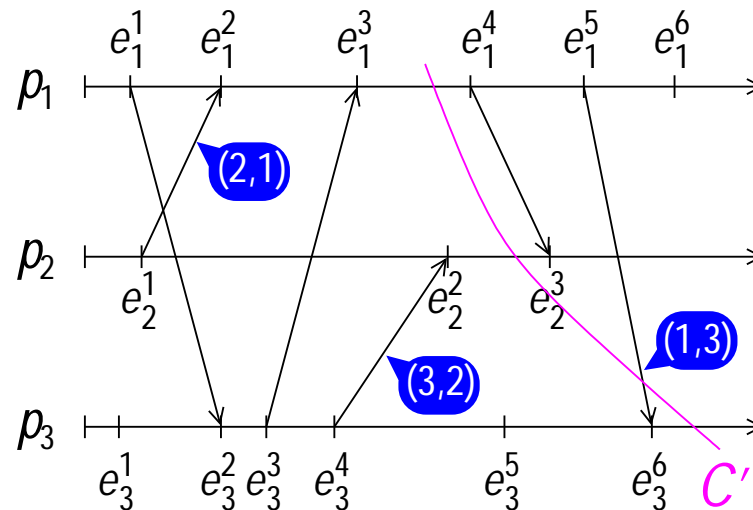
- O WFG⁺ construído por p_M para este estado global terá as arestas (1, 3), (2, 1) e (3, 2)



- Ciclo presente, o que implica que p_M reporta um *deadlock* envolvendo os três processos
- Ocorreu efetivamente um *deadlock*?

Monitoramento computações distribuídas

- Observador global não detecta um *deadlock*
- Condição detectada por p_M é chamada de *ghost deadlock*?
 - Corte C' corresponde a um corte inconsistente
 - Predicado aplicado a um corte inconsistente pode levar a conclusões incorretas

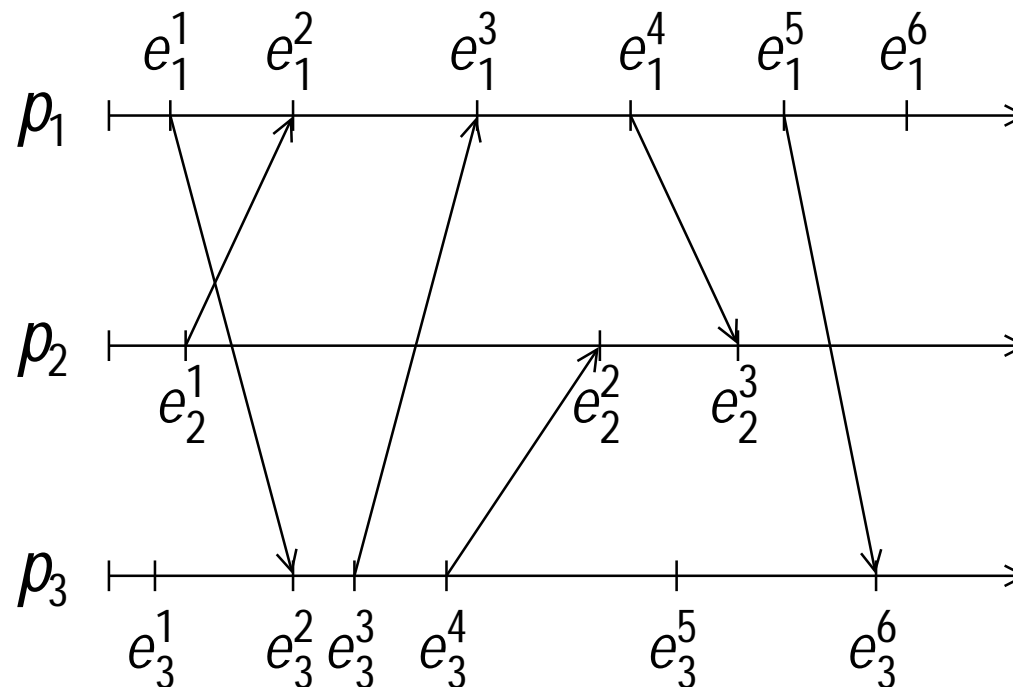


Observando computações distribuídas

- Estratégia do processo p_M :
 - **Não** envia mensagens de $\langle \text{STATE-ENQUIRY} \rangle$
 - p_M tem um papel passivo
- Cada processo é modificado para enviar uma mensagem para p_M quando um evento e ocorre
 - Continua válida a observação anterior que eventos executados por causa de p_M não interferem na enumeração canônica dos eventos
- Questão:
 - É necessário que todo evento seja reportado a p_M ? (lembre-se que p_M está avaliando o predicado Φ)

Observando computações distribuídas

- Um evento e_i^k é relevante para o predicado Φ se o valor de Φ avaliado no estado global (\dots, s_i^k, \dots) é diferente daquele avaliado no estado global $(\dots, s_i^{k-1}, \dots)$
 - Na computação abaixo, os únicos eventos relevantes para a detecção de *deadlock* são aqueles relacionados com o envio e recepção de mensagens



Observando computações distribuídas

- Diferentes mensagens de notificação podem sofrer diferentes atrasos até p_M
 - Efeito relativista da computação distribuída
- Questão:
 - O estado global construído por p_M é consistente?

Observando computações distribuídas

- Uma observação pode corresponder a:
 - Execução consistente
 - Execução inconsistente
 - Nenhuma execução já que eventos de p_i podem ser observados por p_M numa ordem diferente da história local de p_i

- Uma observação consistente corresponde a uma execução consistente
 - Exemplo de uma execução consistente:

$$R = e_3^1 e_1^1 e_3^2 e_2^1 e_3^3 e_3^4 e_2^2 e_1^2 e_3^5 e_1^3 e_1^4 e_1^5 e_3^6 e_2^3 e_1^6$$

- Exemplos de possíveis observações de R :

$$O_1 = e_2^1 e_1^1 e_3^1 e_3^2 e_3^4 e_1^2 e_2^2 e_3^3 e_1^3 e_1^4 e_3^5 \dots$$

$$O_2 = e_1^1 e_3^1 e_2^1 e_3^2 e_1^2 e_3^3 e_3^4 e_1^3 e_2^2 e_3^5 e_3^6 \dots$$

$$O_3 = e_3^1 e_2^1 e_1^1 e_1^2 e_3^2 e_3^3 e_1^3 e_3^4 e_1^4 e_2^2 e_1^5 \dots$$

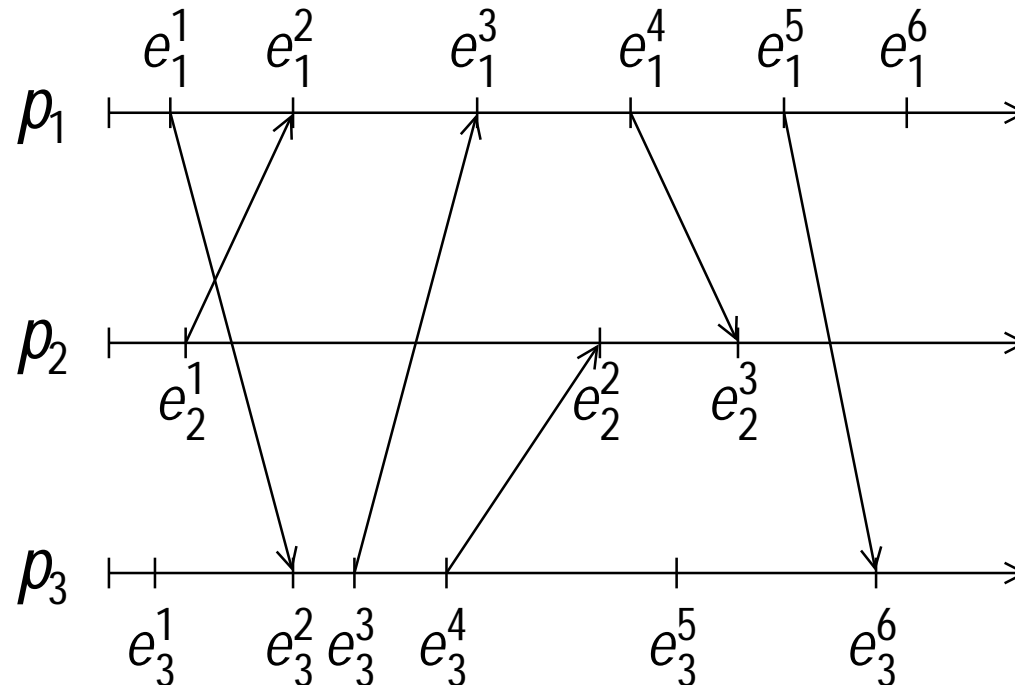
→ Qualquer permutação de uma execução R é uma possível observação

Observando computações distribuídas

- O_1 não corresponde a uma execução já que eventos de p_3 não representam um prefixo inicial de sua história local ($e_3^4 \not\rightarrow e_3^3$ em h_3)

$$R = e_3^1 e_1^1 e_3^2 e_2^1 e_3^3 e_3^4 e_2^2 e_1^2 e_3^5 e_1^3 e_1^4 e_1^5 e_3^6 e_2^3 e_1^6$$

$$O_1 = e_2^1 e_1^1 e_3^1 e_3^2 e_3^4 e_1^2 e_2^2 e_3^3 e_1^3 e_1^4 e_3^5 \dots$$

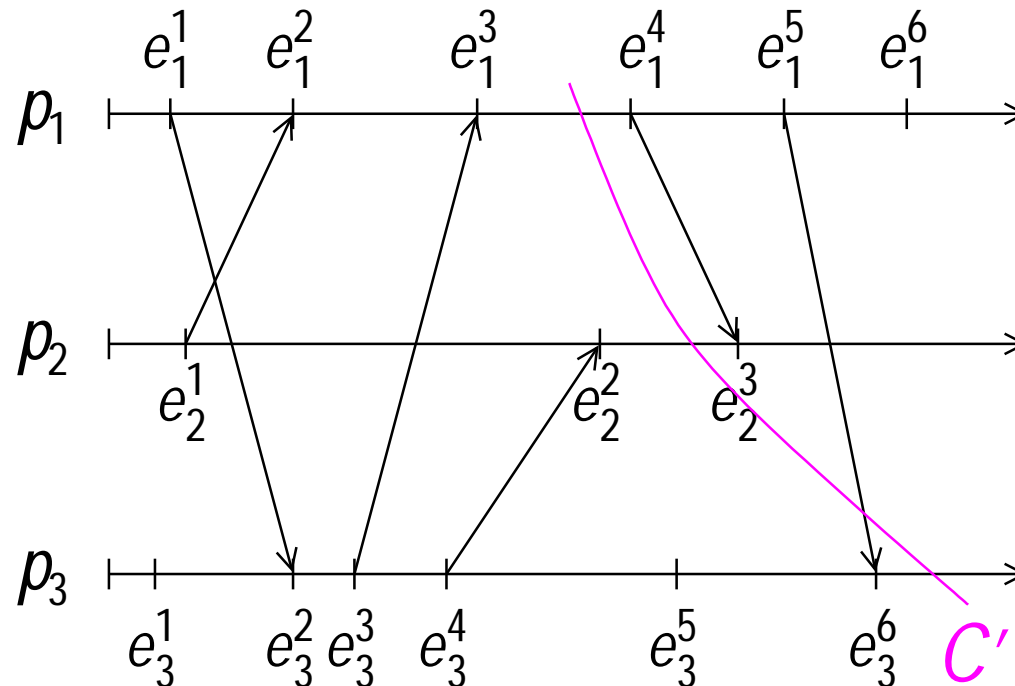


Observando computações distribuídas

- O_2 corresponde a uma execução inconsistente já que o estado global do último evento de cada processo (s_1^3, s_2^2, s_3^6) corresponde ao corte de C'

$$R = e_3^1 e_1^1 e_3^2 e_2^1 e_3^3 e_3^4 e_2^2 e_1^2 e_3^5 e_1^3 e_1^4 e_1^5 e_3^6 e_2^3 e_1^6$$

$$O_2 = e_1^1 e_3^1 e_2^1 e_3^2 e_1^2 e_3^3 e_3^4 e_1^3 e_2^2 e_3^5 e_3^6 \dots$$

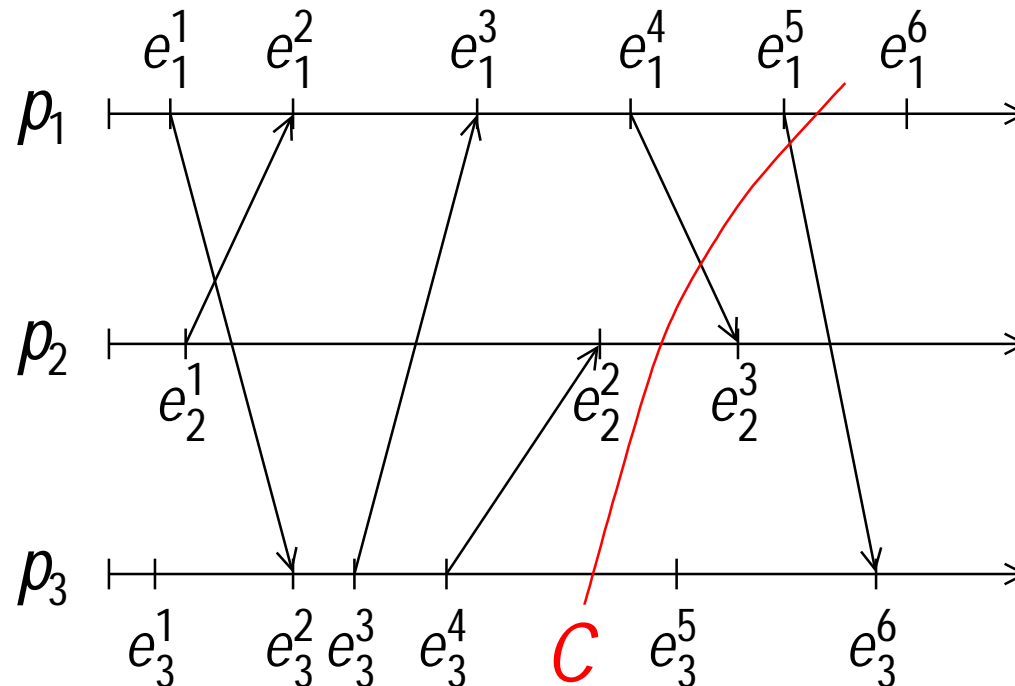


Observando computações distribuídas

- O_3 é uma observação consistente e leva ao mesmo estado global do corte C

$$R = e_3^1 e_1^1 e_3^2 e_2^1 e_3^3 e_3^4 e_2^2 e_1^2 e_3^5 e_1^3 e_1^4 e_1^5 e_3^6 e_2^3 e_1^6$$

$$O_3 = e_3^1 e_2^1 e_1^1 e_1^2 e_3^2 e_3^3 e_1^3 e_3^4 e_1^4 e_2^2 e_1^5 \dots$$



Observando computações distribuídas

- O_1 ocorreu por uma reordenação de mensagens do canal, que pode ser garantida pela propriedade *First-In-First-Out (FIFO) delivery*
- Para todas as mensagens m e m'
FIFO Delivery: $send_i(m) \rightarrow send_i(m') \Rightarrow deliver_j(m) \rightarrow deliver_j(m')$
- Para cada par origem–destino, a entrega FIFO pode ser garantida sobre canais não-FIFO
 - Basta incluir um número de seqüência a cada mensagem na origem e entregar mensagens no destino de acordo com esse número
- Canais FIFO garantem que observações correspondem a execuções
 - As observações são agora consistentes?
Não já que situações como O_2 podem continuar ocorrendo

Observando computações distribuídas

- Mecanismo para garantir consistência:
 - Suponha que todos os processos tenham acesso a um relógio de tempo real RC e que o atraso máximo de mensagens é δ
 - Não é um sistema assíncrono
 - Seja $RC(e)$ o valor do relógio global quando o evento e ocorre
 - Todo o processo inclui $RC(e)$ quando envia uma mensagem de notificação para p_M
 - Funciona como *timestamp*
- Qual deve ser a regra de processamento de p_M para processar as mensagens recebidas?
 - *DR1*: No instante t , processe todas as mensagens com *timestamp* até $t - \delta$
- Conseqüentemente, se $e \rightarrow e' \Rightarrow RC(e) < RC(e')$

Relógio lógico

- Mecanismo usado para identificar o “tempo” (instante) de ocorrência dos eventos
- Ordem dos eventos baseada neste mecanismo é consistente com a precedência causal
 - A condição de relógio pode ser satisfeita num sistema assíncrono
 - Solução adequada para muitas aplicações

Relógio lógico: Funcionamento

- Processo p_i mantém uma variável local LC_i (relógio lógico ou *logical clock*)
- LC_i é usada para associar eventos a números naturais positivos
- O valor do relógio lógico quando o evento e_i^k ocorre em p_i é denominado $LC_i(e_i^k)$
- Cada mensagem m enviada por p_i contém o *timestamp* $TS(m)$
 - $TS(m) = \text{valor de } LC_i(m)$
- $\forall p_i, LC_i \leftarrow 0$, quando os processos são inicializados

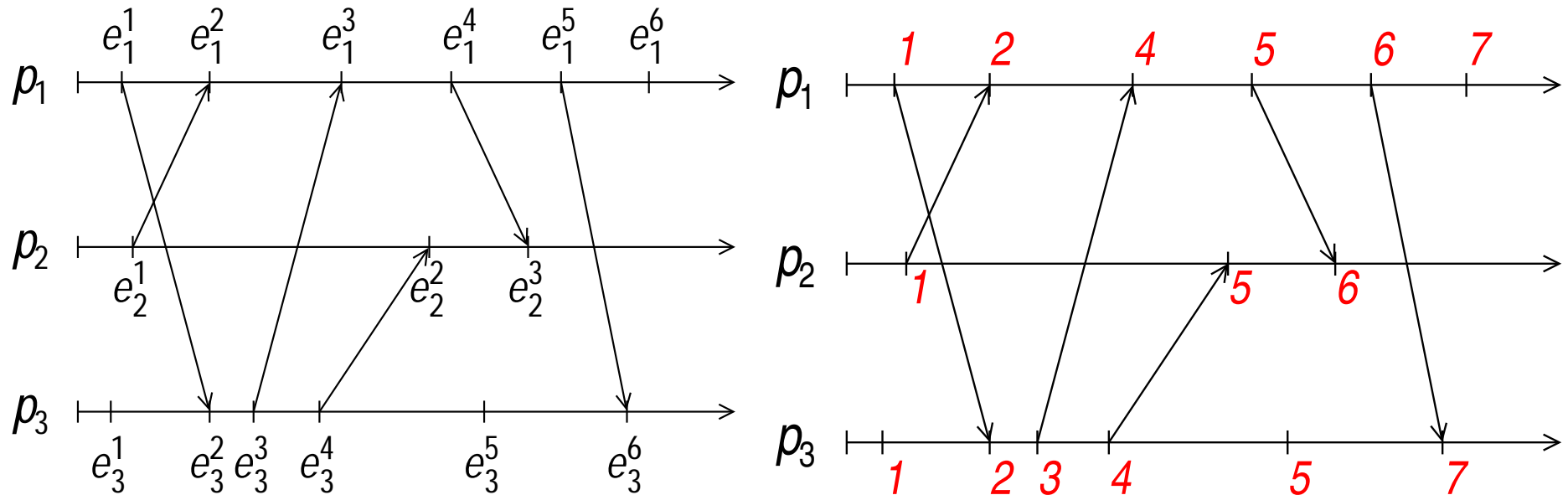
Relógio lógico: Regras de atualização

- Quando um evento e_i^k ocorre, a variável de relógio lógico é atualizado da seguinte forma:

$$LC_i(e_i^k) \leftarrow \begin{cases} LC_i \leftarrow LC_i + 1 & \text{se } e_i^k \text{ é um evento interno ou} \\ & \text{envio de uma mensagem} \\ \max(LC_i, TS(m)) + 1 & \text{se } e_i^k \text{ corresponde à} \\ & \text{recepção de uma mensagem} \end{cases}$$

- Quando uma mensagem é recebida, o seu *timestamp* é maior que o *TS* local e o que está na mensagem

Relógio lógico: Exemplo



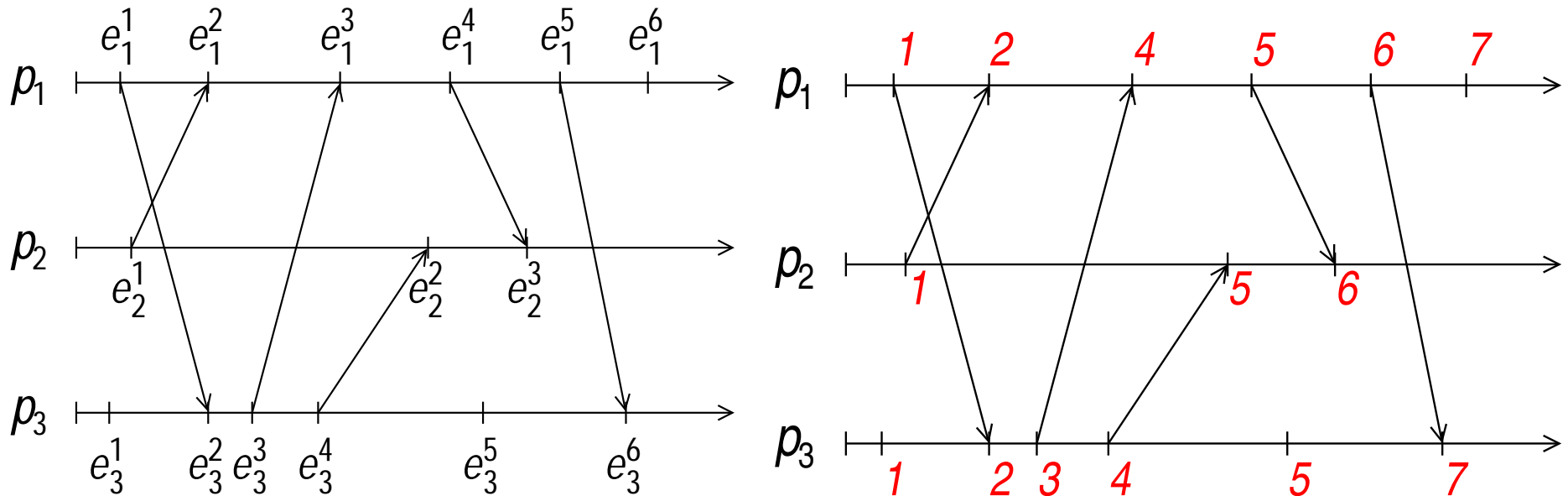
Comentários:

- Os valores do relógio lógico são consistentes com a precedência causal
- \forall eventos e e e' , se $e \rightarrow e'$ então $LC(e) < LC(e')$

→ Relógio lógico satisfaz a condição de relógio

Observação usando relógio lógico

- Qual deve ser a regra para p_M processar as mensagens recebidas?
 - Entregue para p_M mensagens em ordem crescente de relógio lógico



- Observação de p_M :

$e_1^1 e_2^1 e_3^1 e_1^2 e_3^2 e_3^3 e_1^3 e_3^4 e_1^4 e_2^5 e_3^5 e_1^6 e_2^6 e_3^6$

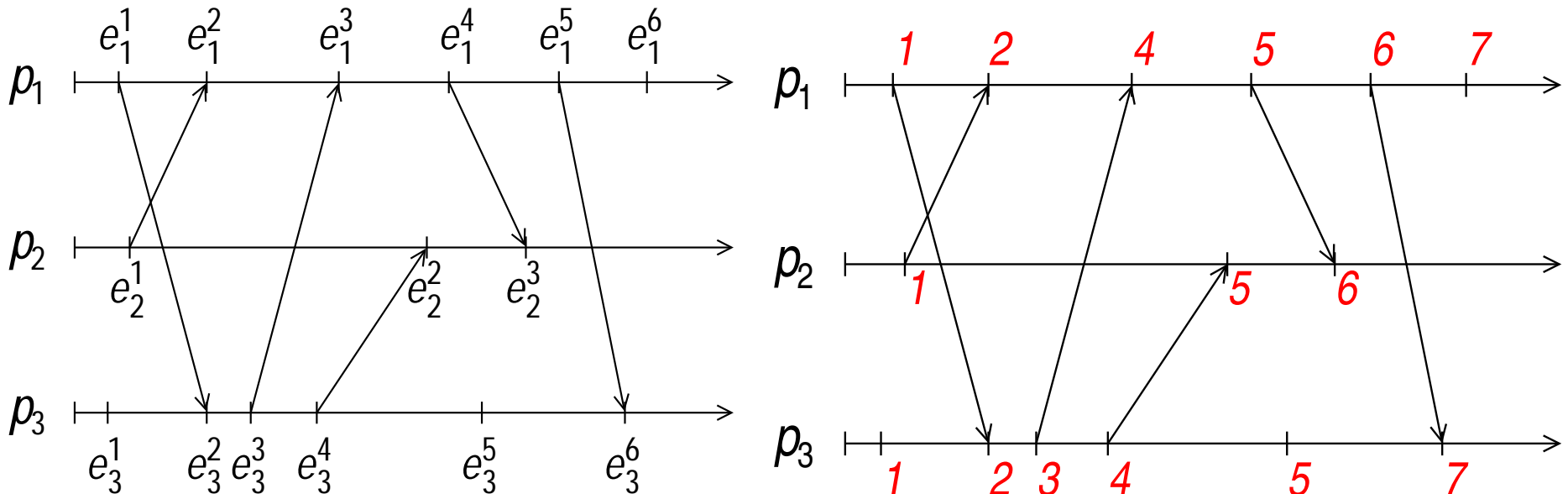
- A observação é consistente?

Observação usando relógio lógico

- Qual é o problema com a regra

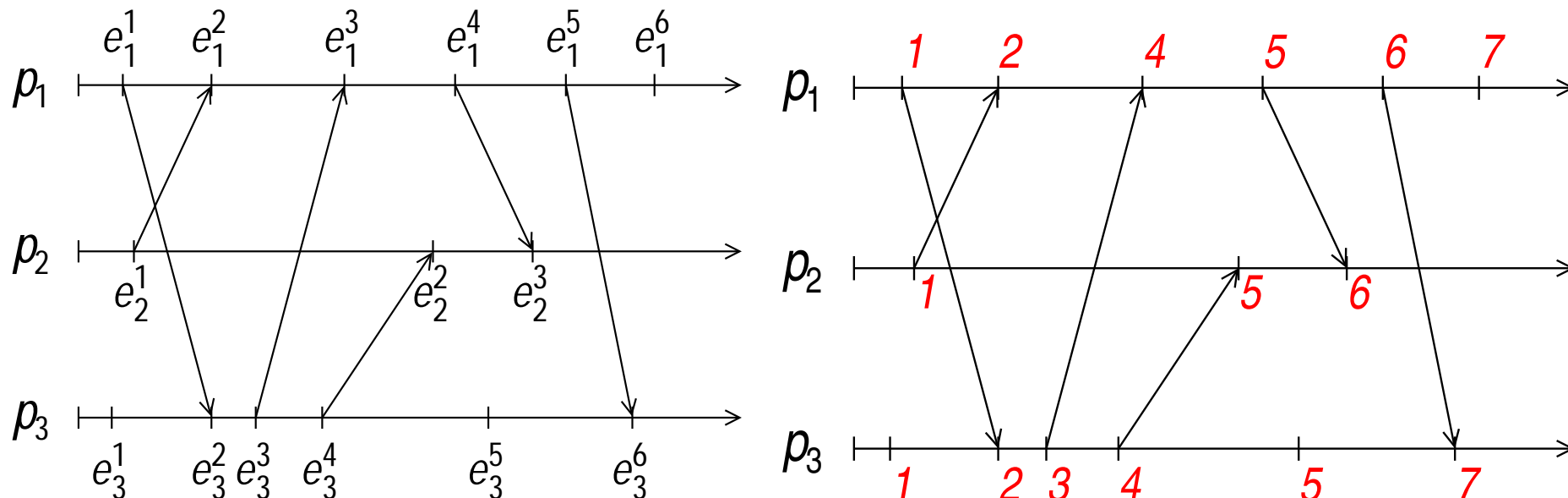
Entregue para p_M mensagens em ordem crescente de relógio lógico?

- Observe a entrega de mensagens para p_M de acordo com os valores de relógio lógico:



Evento	e_1^1	e_2^1	e_3^1	e_1^2	e_3^2	e_3^3	e_1^3	e_3^4	e_1^4	e_2^2	e_3^5	e_1^5	e_2^3	e_1^6	e_3^6
LC	1	1	1	2	2	3	4	4	5	5	5	6	6	7	7

Observação usando relógio lógico



Evento	$e_1^2 \rightarrow e_3^3$	$e_2^1 \rightarrow e_2^2$	$e_3^5 \rightarrow e_3^6$
LC	2 ... 4	1 ... 5	5 ... 7

- A regra de entrega não possui *liveness*
 - Qual é o limite do atraso para entrega de mensagens?
 - Existe um relógio de tempo real para medir esse atraso?
- *LC*, como mecanismo de “marcar” o tempo, não é capaz de fazer *gap-detection*

Propriedades em sistemas distribuídos

- Em geral, as propriedades são divididas em duas grandes classes:
 - *Safety*
 - *Liveness*
- *Safety*
 - Propriedades não desejáveis eventualmente não acontecem
 - Exemplo: *deadlock*
- *Liveness*
 - Propriedades desejáveis eventualmente acontecem
 - Exemplo: terminação, progresso da computação

Gap-detection

- Dados dois eventos

$$e \quad e \quad e'$$

e os seus respectivos valores de relógio lógico

$$LC(e) \text{ e } LC(e'),$$

sendo que

$$LC(e) < LC(e'),$$

determine se existe um outro evento e'' tal que

$$LC(e) < LC(e'') < LC(e').$$

- Esta propriedade é necessária para garantir *liveness* da regra de entrega
 - Pode ser obtida com LC num SA mas deve-se ter mais alguma condição além dos valores do relógio
- Possibilidade:
 - Ter comunicação FIFO entre todos os processos e p_M

Gap-detection

- Invariante:
 - Se p_M recebe uma mensagem m do processo p_i com *timestamp* $TS(m)$, então sabe-se que nenhuma outra mensagem m' pode vir de p_i tal que $TS(m') < TS(m)$
 - Mensagem m é chamada de estável
- Quando ocorre estabilidade de uma mensagem m vinda de p_i em p_M considerando todos os processos?
 - Quando p_M recebe uma mensagem de todos os outros processos com valores de *timestamp* maiores que $TS(m)$
- Qual deve ser a regra para p_M processar as mensagens recebidas?
 - Entregue para p_M mensagens que são estáveis em p_M em ordem crescente de relógio lógico

Gap-detection

- A propriedade de *liveness* passa a ser garantida?
 - Não.
- Possível cenário: O que acontece se um processo não envia uma mensagem para p_M a partir de um certo ponto?
 - Mensagens não podem ser mais entregues para p_M por causa da falta da estabilidade
- Como resolver este problema?
 - Processo monitor p_M envia uma mensagem vazia para todos os processos que deve ser confirmada
 - Serve para “forçar” o envio de mensagens que possam estar nos canais

Gap-detection

- Relógios de tempo real possuem a propriedade de *gap-detection*?
 - Não.
- Como a estabilidade de mensagens foi garantida quando foi suposta a existência de um relógio de tempo real?
 - As mensagens sofriam um atraso máximo de δ

Entrega causal

- Entrega FIFO garante a preservação da ordem de todas as mensagens enviadas por um mesmo processo para p_M
- Idéia que pode ser estendida para todas as mensagens relacionadas causalmente mesmo que tenham sido enviadas por diferentes processos
 - Propriedade chamada de “Entrega Causal” (*Causal Delivery*)

- Propriedade de Entrega Causal (EC):

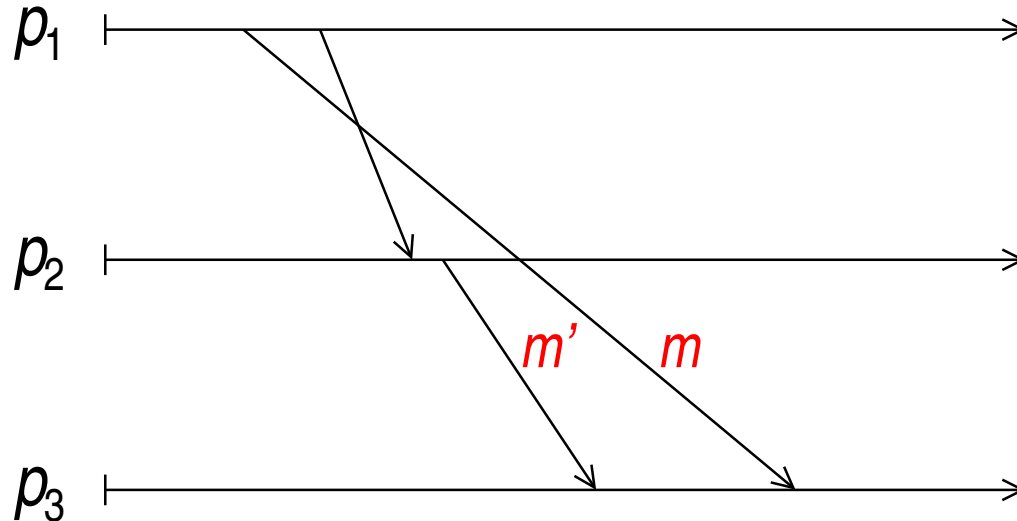
$$send_i(m) \rightarrow send_j(m') \Rightarrow deliver_k(m) \rightarrow deliver_k(m')$$

\forall mensagens m e m' , processos origem p_i e p_j e processo destino p_k

- Conseqüência:
 - Num SA com EC, um processo não pode saber sobre a existência de uma mensagem m através de mensagens intermediárias e mais cedo que o evento correspondente a entrega de m

Entrega causal

- A entrega FIFO de mensagens entre pares de processos é suficiente para garantir EC?
 - Não.
- Exemplo de entrega FIFO mas que não é causal:



Entrega causal

- Importância de EC para a construção de observações consistentes:
 - Se p_M usa a regra de entrega que satisfaz EC, então todas as suas observações serão consistentes
 - A validade desta afirmação vem da própria definição de EC
- Exemplo de entrega de mensagens que não segue EC:

Seja um grupo de pessoas $P = \{p_1, p_2, \dots, p_n\}$ e o seguinte cenário, tipicamente encontrado na Internet:

 - p_1 envia pergunta \mathcal{P} para o grupo P ;
 - p_2 envia resposta \mathcal{R} para o grupo;

Neste caso, uma outra pessoa pode receber \mathcal{R} antes de \mathcal{P} e não irá entender a mensagem.

 - Situação que ocorre com servidores de correio eletrônico e *newsgroups* que não garantem entrega causal.
 - Solução: incluir a pergunta na resposta.

Construção da relação de precedência causal

- Problema a ser resolvido: Dados dois eventos

$$e \quad e \quad e'$$

que são causalmente relacionados e seus LC s, existe um outro evento e'' tal que

$$e \rightarrow e'' \rightarrow e'?$$

- Ao entregar mensagens em ordem crescente de *timestamp* temos que:

$$RC(e) < RC(e') \text{ ou } LC(e) < LC(e'),$$

$$\Rightarrow e \rightarrow e'$$

- Esta conclusão é conservadora já que o *timestamp* usando RC (relógio de tempo real) ou LC (relógio lógico) garante a condição de relógio

- Dado

$$RC(e) < RC(e') \text{ ou } LC(e) < LC(e'),$$

temos que:

(a) e precede causalmente e' , i.e., $e \rightarrow e'$, ou

(b) e é concorrente com e' , i.e., $e || e'$

- Com certeza tem-se que $e' \not\rightarrow e$

Construção da relação de precedência causal

- Seja o seguinte cenário:
 - $e || e'$, e
 - p_M recebe a notificação do evento e'
 - As regras de entrega usando *RC* ou *LC* podem atrasar a entrega de e' , mesmo que seja possível prever os *timesteps* de notificações não recebidas ainda
- Idéia:
 - Definir um mecanismo de tempo *TC* (*Timing Mechanism*) tal que relações de precedência causal entre eventos podem ser deduzidas de seus *timesteps*

Construção da relação de precedência causal

- Condição forte de relógio (*Strong Clock Condition – SCC*):

$$SCC: e \rightarrow e' \equiv TC(e) < TC(e')$$

- Observações:
 - *RC* e *LC* são consistentes com precedência causal
 - *TC* é dito “caracterizar” a precedência causal já que é possível reconstruir toda a computação de uma observação contendo *TC* como *timestamp*
 - É importante na implementação de *CD* e de outras aplicações em sistemas distribuídos

História causal

- Como obter a *SCC*?
 - *TC* que produz o conjunto de todos os eventos que causalmente precedem um evento
- Definição de história causal e um evento e' numa computação distribuída (H, \rightarrow) :

$$\theta(e') = \{e \in H \mid e \rightarrow e'\} \cup \{e'\}$$

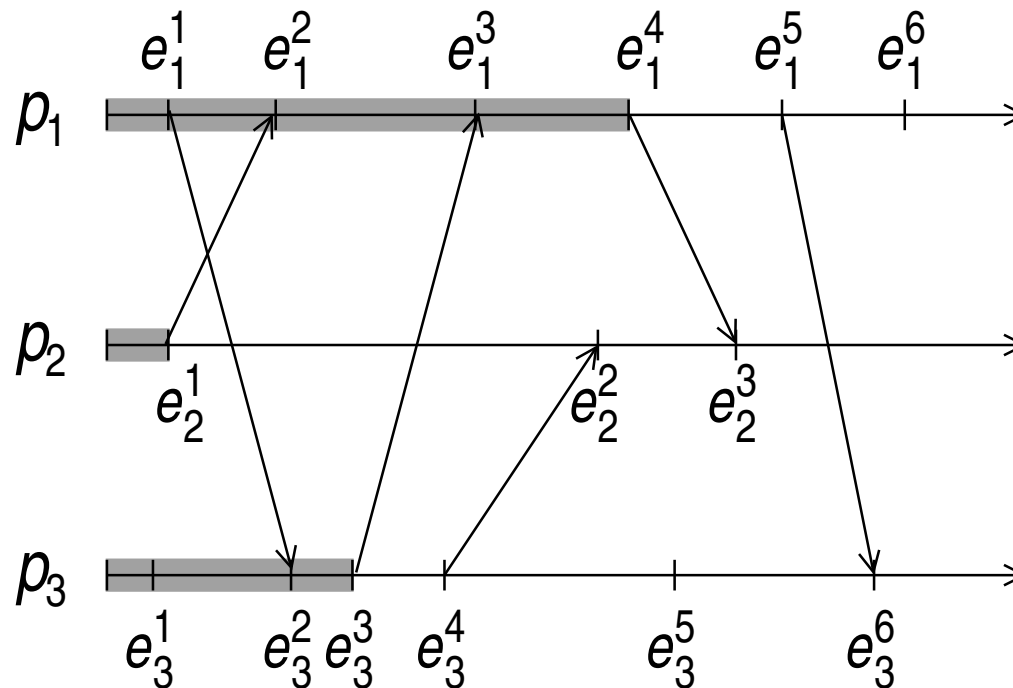
- O que significa o conjunto $\theta(e')$?
 - É o “menor” corte consistente que inclui e' , ou seja, o corte consistente mais próximo de e'
- A projeção de $\theta(e')$ no processo p_i é o conjunto

$$\theta_i(e') = \theta(e') \cap h_i$$

História causal

- História causal do evento e_1^4 :

$$\theta(e_1^4) = \{e_1^1, e_1^2, e_1^3, e_1^4, e_2^1, e_3^1, e_3^2, e_3^3\}$$



História causal

- Como construir uma história causal?

Algoritmo para construção da história causal:

1. $\forall p_i, \theta_i \leftarrow \emptyset;$

2. **se** evento e_i^x corresponde ao recebimento da mensagem m
enviada de p_j para p_i **então**

$\theta_i(e_i^x) \leftarrow \{e_i^x\} \cup$ {O próprio evento e_i^x
 $\theta_i(e_i^{x-1}) \cup$ {História causal do evento anterior em p_i
 $\theta_j(\text{send}_j(m));$ {História causal do evento de envio de m em p_j

senão

$\{e_i^x$ é um evento interno ou um evento de envio de m

$\theta_i(e_i^x) \leftarrow \{e_i^x\} \cup$ {O próprio evento e_i^x
 $\theta_i(e_i^{x-1});$ {História causal do evento anterior em p_i

fimse;

História causal

- Quando a história causal é usada como um valor de relógio, a *SCC* pode ser satisfeita se a comparação de relógios for interpretada como inclusão de conjuntos

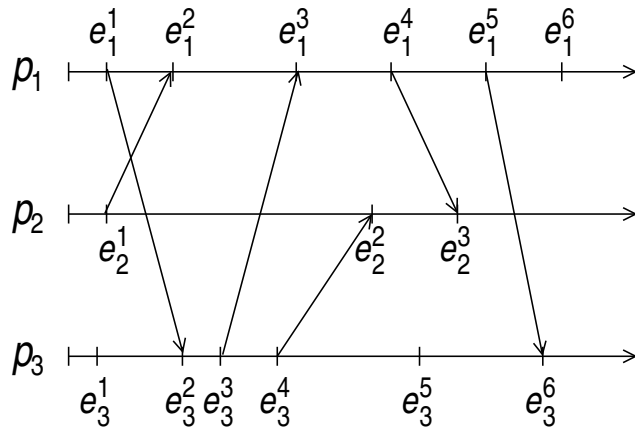
$$e \rightarrow e' \equiv \theta(e) \subset \theta(e')$$

- No entanto, supondo $e \neq e'$, o teste de inclusão de conjunto pode ser substituído por um simples teste de inclusão de um membro em um conjunto:

$$e \in \theta(e')$$

- Problema de história causal:
 - Não é uma solução prática já que cresce bastante

História causal



$$\theta(e_1^1) = \{e_1^1\}$$

$$\theta(e_1^2) = \{e_1^1, e_1^2, e_2^1\}$$

$$\theta(e_1^3) = \{e_1^1, e_1^2, e_1^3, e_2^1, e_2^2, e_3^1, e_3^2, e_3^3\}$$

$$\theta(e_1^4) = \{e_1^1, e_1^2, e_1^3, e_1^4, e_2^1, e_2^2, e_3^1, e_3^2, e_3^3\}$$

$$\theta(e_1^5) = \{e_1^1, e_1^2, e_1^3, e_1^4, e_1^5, e_2^1, e_2^2, e_3^1, e_3^2, e_3^3\}$$

$$\theta(e_1^6) = \{e_1^1, e_1^2, e_1^3, e_1^4, e_1^5, e_1^6, e_2^1, e_2^2, e_3^1, e_3^2, e_3^3\}$$

$$\theta(e_2^1) = \{e_1^1\}$$

$$\theta(e_2^2) = \{e_1^1, e_1^2, e_2^1, e_2^2, e_3^1, e_3^2, e_3^3, e_4^1\}$$

$$\theta(e_2^3) = \{e_1^1, e_1^2, e_1^3, e_1^4, e_2^1, e_2^2, e_3^1, e_3^2, e_3^3, e_4^1\}$$

$$\theta(e_3^1) = \{e_1^1\}$$

$$\theta(e_3^2) = \{e_1^1, e_1^2, e_3^1\}$$

$$\theta(e_3^3) = \{e_1^1, e_1^2, e_2^1, e_3^1, e_3^2, e_3^3\}$$

$$\theta(e_3^4) = \{e_1^1, e_1^2, e_2^1, e_2^2, e_3^1, e_3^2, e_3^3, e_4^1\}$$

$$\theta(e_3^5) = \{e_1^1, e_1^2, e_2^1, e_2^2, e_3^1, e_3^2, e_3^3, e_4^1, e_5^1\}$$

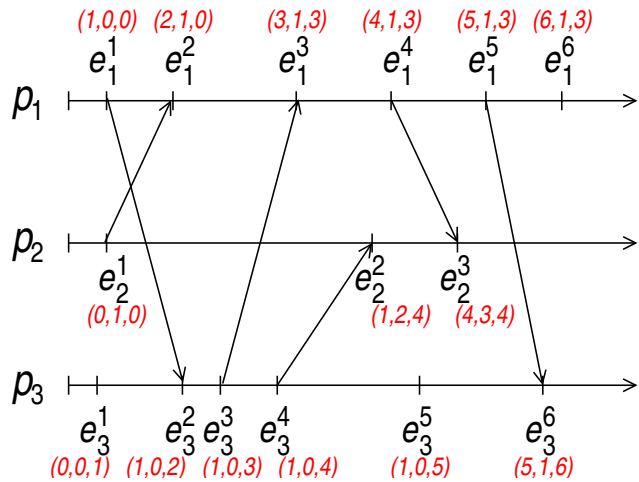
$$\theta(e_3^6) = \{e_1^1, e_1^2, e_1^3, e_1^4, e_1^5, e_1^6, e_2^1, e_2^2, e_3^1, e_3^2, e_3^3, e_3^4, e_3^5, e_3^6\}$$

Vector Clock

- Questão:
 - Como tornar prático o mecanismo de história causal?
- Observe o algoritmo para construção da história causal que:
 - (a) Dado um evento e_i^x , inclui sempre todos os eventos anteriores a esse evento e o próprio evento
 - (b) Se o evento corresponde ao recebimento de uma mensagem m , inclui a história causal do evento de m no processo j
- No caso de (b), deve-se aplicar o mesmo princípio de (a)
 - Conseqüentemente, nos dois casos, a história causal pode ser representada por processo
 - Para cada processo, basta representar o último evento e que ocorreu, que corresponde a um prefixo da história local de p_i
 - A partir desse número k no processo p_i é possível reconstruir $\theta_i(e)$

Vector Clock

História Causal



$$\theta(e_1^1) = \{e_1^1\}$$

$$\theta(e_1^2) = \{e_1^1, e_1^2, e_1^1\}$$

$$\theta(e_1^3) = \{e_1^1, e_1^2, e_1^3, e_1^1, e_1^2, e_1^3, e_1^3\}$$

$$\theta(e_1^4) = \{e_1^1, e_1^2, e_1^3, e_1^4, e_1^1, e_1^2, e_1^3, e_1^3\}$$

$$\theta(e_1^5) = \{e_1^1, e_1^2, e_1^3, e_1^4, e_1^5, e_1^1, e_1^2, e_1^3, e_1^3\}$$

$$\theta(e_1^6) = \{e_1^1, e_1^2, e_1^3, e_1^4, e_1^5, e_1^6, e_1^1, e_1^2, e_1^3, e_1^3\}$$

$$\theta(e_2^1) = \{e_2^1\}$$

$$\theta(e_2^2) = \{e_1^1, e_2^1, e_2^2, e_1^3, e_2^3, e_3^4\}$$

$$\theta(e_2^3) = \{e_1^1, e_2^1, e_3^1, e_1^4, e_2^1, e_2^2, e_2^3, e_1^3, e_2^3, e_3^4\}$$

$$\theta(e_3^1) = \{e_3^1\}$$

$$\theta(e_3^2) = \{e_1^1, e_3^1, e_3^2\}$$

$$\theta(e_3^3) = \{e_1^1, e_3^1, e_2^2, e_3^3\}$$

$$\theta(e_3^4) = \{e_1^1, e_3^1, e_2^2, e_3^3, e_3^4\}$$

$$\theta(e_3^5) = \{e_1^1, e_3^1, e_2^2, e_3^3, e_3^4, e_3^5\}$$

$$\theta(e_3^6) = \{e_1^1, e_2^1, e_3^1, e_1^4, e_1^5, e_1^2, e_1^3, e_2^2, e_3^3, e_3^4, e_3^5, e_3^6\}$$

VC

(p_1, p_2, p_3)

(1, 0, 0)

(2, 1, 0)

(3, 1, 3)

(4, 1, 3)

(5, 1, 3)

(6, 1, 3)

(0, 1, 0)

(1, 2, 4)

(4, 3, 4)

(0, 0, 1)

(1, 0, 2)

(1, 0, 3)

(1, 0, 4)

(1, 0, 5)

(5, 1, 6)

Vector Clock

- História causal pode ser representada por um vetor $VC[1 \dots n]$ (correspondente aos n processos) ao invés de um conjunto
- Para cada evento e
 - $VC(e)[i] = k$
- Cada processo p_i mantém uma cópia do vetor VC onde $VC(e_i^x)$ representa o valor do *vector clock* de p_i quando o evento e_i^x ocorre
- \forall processos $p_i, 1 \leq i \leq n$, inicialmente $VC_i[1 \dots n] = 0$
- Cada mensagem m contém um $TS(m)$ que é o VC correspondente ao evento de *send*

Vector Clock

- Regra de atualização de VC quando um evento e_i^x ocorre:

$$VC_i(e_i^x)[i] \leftarrow VC_i[i] + 1 \quad \forall \text{ evento } e_i^x \text{ em } p_i$$

$$VC_i(e_i^x)[j] \leftarrow \max\{VC_i[j], TS(m)\} \quad \text{se } e_i^x = \text{receive}(m), \\ \forall j, j = 1 \dots n \wedge j \neq i$$

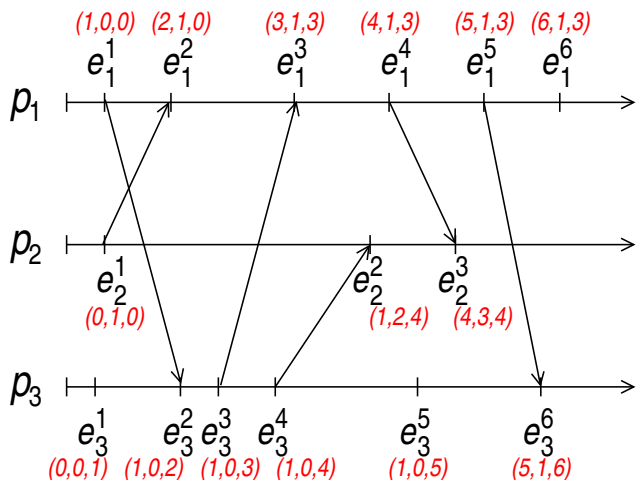
Em outras palavras, quando um evento ocorre:

- O *vector clock* referente ao processo p_i é incrementado
 - $VC_i(e_i^x)[i] \equiv$ o número de eventos em p_i que ocorreram até aquele instante incluindo o próprio evento e_i^x
- O *vector clock* referente aos outros processos $p_j, j = 1 \dots n \wedge j \neq i$ é o máximo entre $VC_i[j]$ e o valor do VC na mensagem m recebida
 - $VC_i(e_i^x)[j], j = 1 \dots n \wedge j \neq i \equiv$ o número de eventos em p_j que causalmente precedem evento e_i^x em p_i

Vector Clock

- É possível definir a relação “menor que” ($<$) entre dois vetores V e V' com n “dimensões” (processos):

$$V < V' \equiv (V \neq V') \wedge (\forall k, 1 \leq k \leq n | V[k] \leq V'[k])$$



VC	Relação $<$
$e_1^1 = (1, 0, 0)$	$\{e_1^1, e_2^1, e_3^1, e_4^1, e_5^1, e_6^1, e_2^2, e_3^2, e_2^3, e_3^3, e_3^3, e_4^3, e_5^3, e_6^3\}$
$e_1^2 = (2, 1, 0)$	$\{e_1^1, e_1^1, e_1^1, e_1^1, e_1^1, e_2^2, e_3^2\}$
$e_1^3 = (3, 1, 3)$	$\{e_1^1, e_1^1, e_1^1, e_1^1, e_1^1, e_2^2, e_3^2\}$
$e_1^4 = (4, 1, 3)$	$\{e_1^1, e_1^1, e_1^1, e_1^1, e_2^2, e_3^2\}$
$e_1^5 = (5, 1, 3)$	$\{e_1^1, e_1^1, e_3^2\}$
$e_1^6 = (6, 1, 3)$	$\{e_1^1\}$
$e_2^1 = (0, 1, 0)$	$\{e_1^1, e_1^1, e_1^1, e_1^1, e_1^1, e_2^2, e_2^2, e_3^2, e_6^2\}$
$e_2^2 = (1, 2, 4)$	$\{e_2^2, e_3^2, e_6^2\}$
$e_2^3 = (4, 3, 4)$	$\{e_3^2\}$
$e_3^1 = (0, 0, 1)$	$\{e_1^1, e_1^1, e_1^1, e_1^1, e_2^2, e_2^2, e_3^2, e_3^2, e_3^2, e_4^3, e_5^3, e_6^3\}$
$e_3^2 = (1, 0, 2)$	$\{e_1^1, e_1^1, e_1^1, e_1^1, e_2^2, e_2^2, e_3^2, e_3^2, e_3^2, e_4^3, e_5^3, e_6^3\}$
$e_3^3 = (1, 0, 3)$	$\{e_1^1, e_1^1, e_1^1, e_1^1, e_2^2, e_2^2, e_3^2, e_3^2, e_3^2, e_4^3, e_5^3, e_6^3\}$
$e_3^4 = (1, 0, 4)$	$\{e_2^2, e_2^2, e_3^2, e_3^2, e_6^2\}$
$e_3^5 = (1, 0, 5)$	$\{e_3^2, e_6^2\}$
$e_3^6 = (5, 1, 6)$	$\{e_3^2\}$

Vector Clock

- É possível expressar a condição forte de relógio (*SCC*) em termos de *vector clocks*

Propriedade 1 **Strong Clock Condition:**

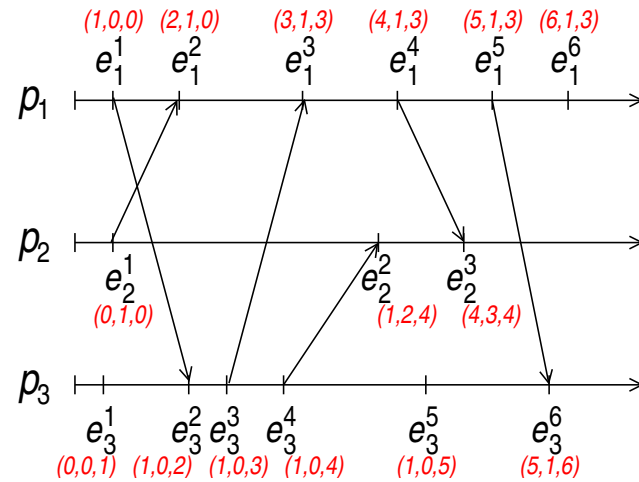
$$e \rightarrow e' \equiv VC(e) < VC(e')$$

- No caso de se conhecer os processos onde e e e' ocorrem, a precedência causal pode ser verificada por uma simples comparação dos respectivos valores de VC

Vector clock

Exemplo da propriedade 1
SCC, ou seja, $e \rightarrow e' \equiv$

$$VC(e) < VC(e')$$



	e_1^1	e_1^2	e_1^3	e_1^4	e_1^5	e_1^6	e_2^1	e_2^2	e_2^3	e_3^1	e_3^2	e_3^3	e_3^4	e_3^5	e_3^6
	100	210	313	413	513	613	010	124	434	001	102	103	104	105	516
e_1^1	<	<	<	<	<	<		<	<		<	<	<	<	<
e_1^2		<	<	<	<	<			<						<
e_1^3			<	<	<	<			<						<
e_1^4				<	<	<			<						<
e_1^5					<	<									<
e_1^6						<									
e_2^1		<	<	<	<	<	<	<	<						<
e_2^2							<	<							<
e_2^3								<							
e_3^1			<	<	<	<		<	<	<	<	<	<	<	<
e_3^2			<	<	<	<		<	<		<	<	<	<	<
e_3^3			<	<	<	<		<	<		<	<	<	<	<
e_3^4								<	<				<	<	<
e_3^5														<	<
e_3^6															<

Vector Clock

Propriedade 2 **Simple Strong Clock Condition** (SSCC): Sejam os eventos e_i no processo p_i e e_j no processo p_j , para $i \neq j$:

$$e_i \rightarrow e_j \equiv VC(e_i)[i] \leq VC(e_j)[i]$$

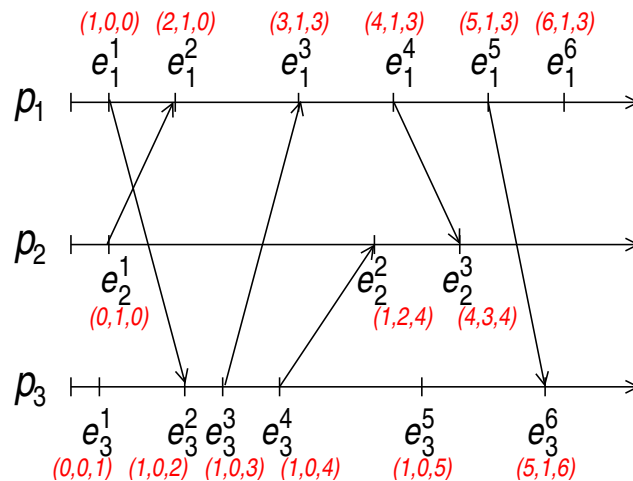
- Observe que o índice i ($[i]$) diz respeito ao processo p_i
- Se existe uma precedência causal entre os eventos e_i e e_j , o valor de $VC_i[i]$ quando e_i ocorre é menor ou igual a $VC_j[i]$ quando e_j ocorre

Vector clock

Questão sobre a propriedade 2 SSCC: A condição

$$VC(e_i)[i] = VC(e_j)[i]$$

pode ocorrer e representa a situação que e_i é o último evento de p_i ($send(m)$) que precede causalmente e_j em p_j



	e_1^1	e_1^2	e_1^3	e_1^4	e_1^5	e_1^6	e_2^1	e_2^2	e_2^3	e_3^1	e_3^2	e_3^3	e_3^4	e_3^5	e_3^6
	100	210	313	413	513	613	010	124	434	001	102	103	104	105	516
e_1^1								=			=	=	=	=	
e_1^2															
e_1^3									=						
e_1^4															
e_1^5															=
e_1^6															
e_2^1		=	=	=	=	=									=
e_2^2															
e_2^3															
e_3^1															
e_3^2															
e_3^3			=	=	=	=									
e_3^4								=	=						
e_3^5															
e_3^6															

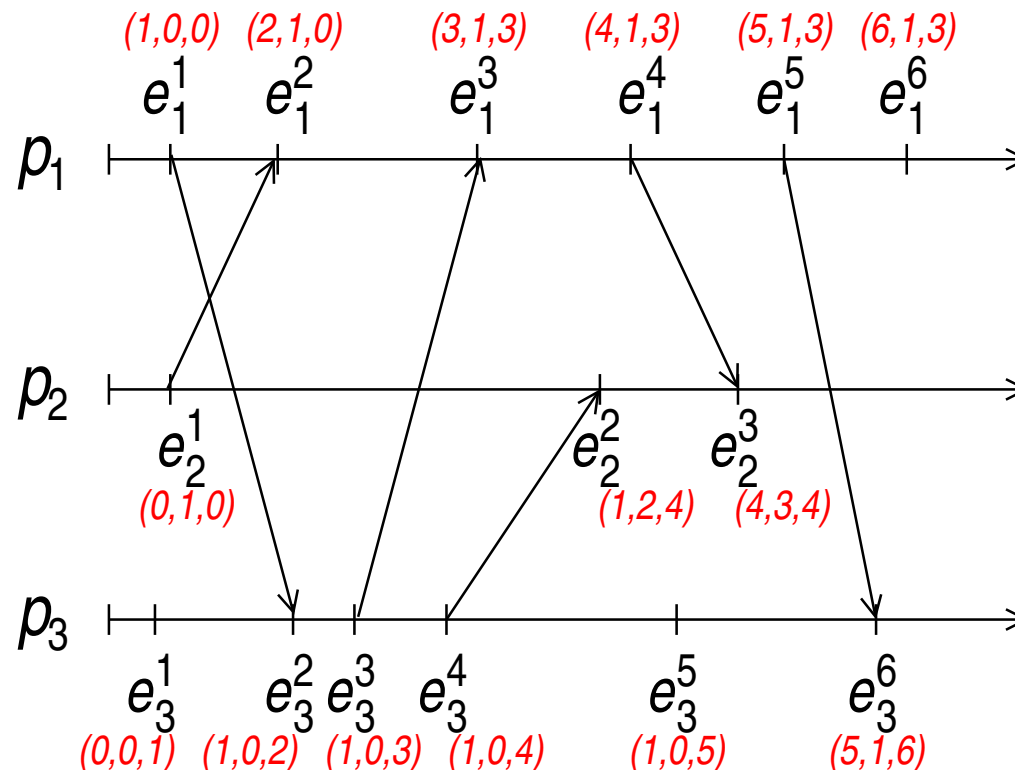
Vector Clock

- Questão: É possível que:

$$e_i \rightarrow e_j \wedge \exists j, j = 1 \dots n, j \neq i | VC(e_i)[j] > VC(e_j)[j]$$

No caso de $e_i \rightarrow e_j$, temos que

$$VC(e_i)[i] \leq VC(e_j)[i]$$



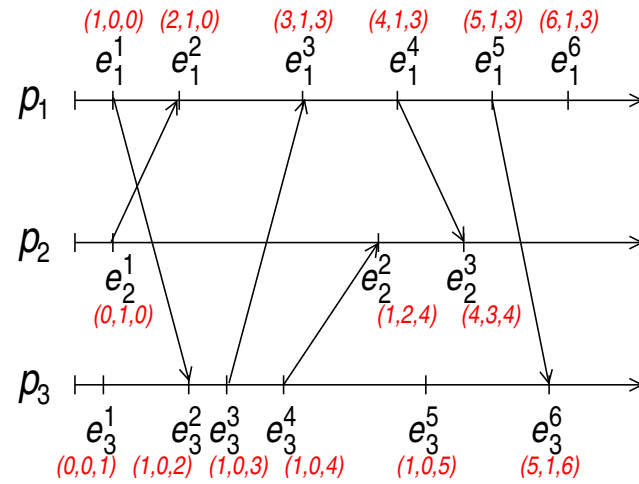
Vector Clock

Propriedade 3 Concorrência: Sejam os eventos e_i no processo p_i e e_j no processo p_j , para $i \neq j$:

$$e_i || e_j \equiv (VC(e_i)[i] > VC(e_j)[i]) \wedge (VC(e_j)[j] > VC(e_i)[j])$$

Vector clock

Exemplo da propriedade 3
Concorrência



	e_1^1	e_1^2	e_1^3	e_1^4	e_1^5	e_1^6	e_2^1	e_2^2	e_2^3	e_3^1	e_3^2	e_3^3	e_3^4	e_3^5	e_3^6
	100	210	313	413	513	613	010	124	434	001	102	103	104	105	516
e_1^1															
e_1^2															
e_1^3															
e_1^4															
e_1^5															
e_1^6															
e_2^1															
e_2^2															
e_2^3															
e_3^1															
e_3^2															
e_3^3															
e_3^4															
e_3^5															
e_3^6															

Vector Clock

- Consistência de cortes de uma computação distribuída pode ser verificada em termos de *vector clocks*
- Eventos e_i e e_j são ditos serem inconsistentes par-a-par se eles não podem pertencer a fronteira do mesmo corte consistente

Propriedade 4 Inconsistente par-a-par: Evento e_i no processo p_i é inconsistente par-a-par com evento e_j no processo p_j , para $i \neq j$, sse:

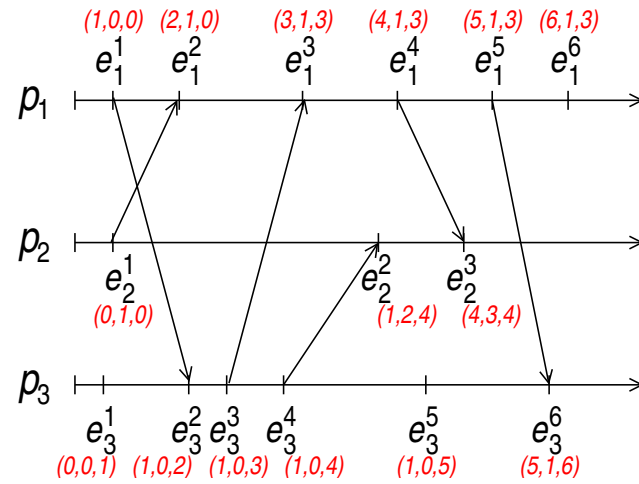
$$VC(e_i)[i] \leq VC_j(e_j)[i] (\equiv e_i \rightarrow e_j)$$

e e_i não pertence ao corte e e_j sim

- O termo identifica a possibilidade do corte incluir pelo menos um evento *receive* sem incluir o evento *send* correspondente

Vector clock

Exemplo da propriedade 4
Inconsistente par-a-par



	e_1^1	e_1^2	e_1^3	e_1^4	e_1^5	e_1^6	e_2^1	e_2^2	e_2^3	e_3^1	e_3^2	e_3^3	e_3^4	e_3^5	e_3^6
	100	210	313	413	513	613	010	124	434	001	102	103	104	105	516
e_1^1								⌊	⌊		⌊	⌊	⌊	⌊	⌊
e_1^2									⌊						⌊
e_1^3									⌊						⌊
e_1^4									⌊						⌊
e_1^5															⌊
e_1^6															⌊
e_2^1		⌊	⌊	⌊	⌊	⌊									⌊
e_2^2															⌊
e_2^3															⌊
e_3^1			⌊	⌊	⌊	⌊		⌊	⌊						
e_3^2			⌊	⌊	⌊	⌊		⌊	⌊						
e_3^3			⌊	⌊	⌊	⌊		⌊	⌊						
e_3^4								⌊	⌊						
e_3^5															
e_3^6															

Vector Clock

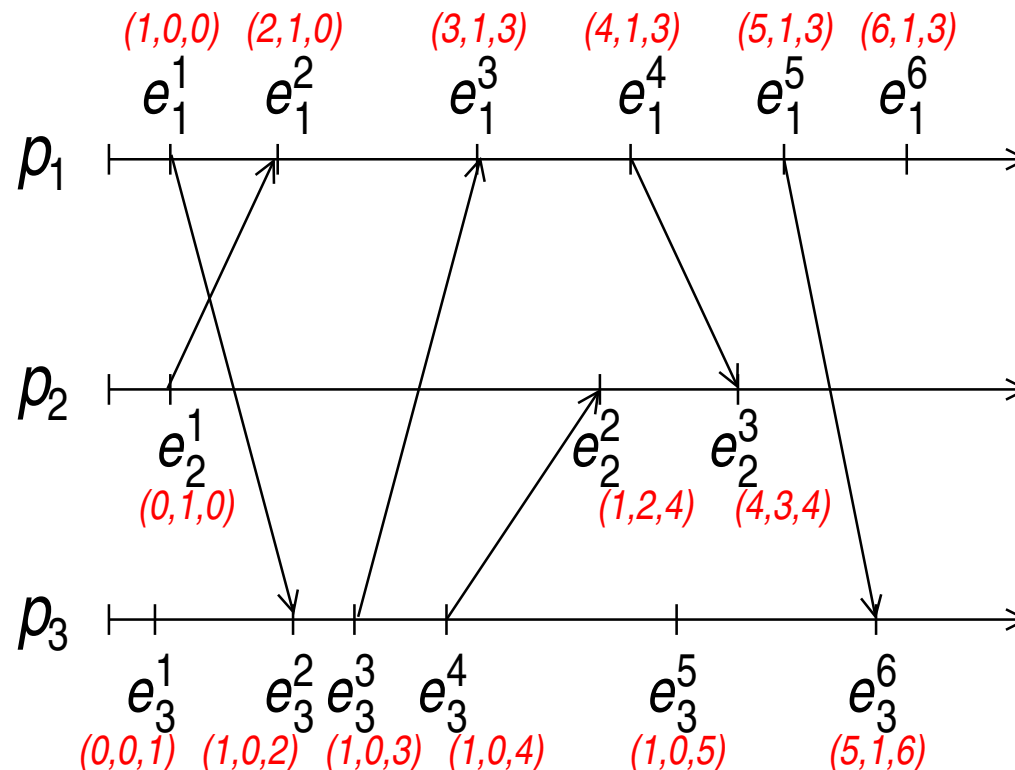
- Um corte consistente não inclui na sua fronteira eventos inconsistentes par-a-par
- Dada a definição de um corte, basta verificar a inconsistência par-a-par dos eventos que estão na fronteira do corte
- Eventos e_i e e_j são ditos serem inconsistentes par-a-par se eles não podem pertencer a fronteira do mesmo corte consistente

Vector Clock

Propriedade 5 Corte consistente: Um corte definido por (c_1, \dots, c_n) é consistente se e somente se

$$\forall i, j, 1 \leq i, j \leq n \mid VC(e_i^{c_i})[i] \leq VC(e_j^{c_j})[i]$$

e e_i pertence ao corte e e_j pode pertencer ou não



Vector Clock

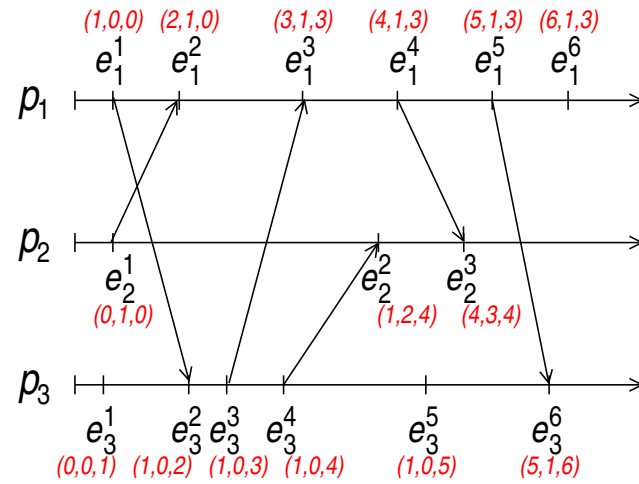
- $VC(e_i)[j]$, $j = 1 \dots n \wedge j \neq i$ representa o número de eventos em p_j que causalmente precedem evento e_i em p_i
- Seja $\#(e_i)$ o número total de eventos que causalmente precedem o evento e_i em toda a computação, que é dado por

$$\#(e_i) = \left(\sum_{j=1}^n VC(e_i)[j] \right) - 1$$

Propriedade 6 Contagem: Dado um evento e_i no processo p_i e o valor de seu *vector clock* $VC(e_i)$, o número de eventos e tal que $e \rightarrow e'$ (ou ainda $VC(e) < VC(e')$) é dado por $\#(e_i)$

Vector clock

Exemplo da propriedade 6
Contagem



Evento	e_1^1	e_1^2	e_1^3	e_1^4	e_1^5	e_1^6	e_2^1	e_2^2	e_2^3	e_3^1	e_3^2	e_3^3	e_3^4	e_3^5	e_3^6
VC	100	210	313	413	513	613	010	124	434	001	102	103	104	105	516
#(e)	0	2	6	7	8	9	0	6	10	0	2	3	4	5	11

Vector Clock

- A propriedade abaixo pode ser usada para determinar se o *gap* causal entre dois eventos admite um terceiro evento

Propriedade 7 Weak gap-detection: Dado um evento e_i no processo p_i e um evento e_j no processo p_j

$$\text{se } VC(e_i)[k] < VC(e_j)[k]$$

para algum $k \neq j$, então existe um evento e_k tal que

$$\neg(e_k \rightarrow e_i) \wedge (e_k \rightarrow e_j)$$

- A propriedade é fraca (*weak*) no sentido que para processos p_i , p_j e p_k não é possível concluir se os três eventos formam a cadeia causal $e_i \rightarrow e_k \rightarrow e_j$
- Para o caso especial $i = k$, a propriedade de fato identifica a condição suficiente para tal conclusão

Vector Clock

Propriedade 7 Weak gap-detection: Dado um evento e_i no processo p_i e um evento e_j no processo p_j se $VC(e_i)[k] < VC(e_j)[k]$ para algum $k \neq j$, então existe um evento e_k tal que

$$\neg(e_k \rightarrow e_i) \wedge (e_k \rightarrow e_j)$$

