

ACH2147 - Desenvolvimento de Sistemas de Informação Distribuídos

Aula 13 – Sincronização e Coordenação

Norton Trevisan Roman

19 de junho de 2022

Sincronização e Coordenação

- Sincronização do *Clock*
- Relógios Lógicos
- Exclusão Mútua

Sincronização e Coordenação

- **Sincronização do Clock**
- Relógios Lógicos
- Exclusão Mútua

Sincronização do *Clock*

Sincronização × Coordenação

- Sincronização de processos
 - Garantimos que um processo espere até outro completar sua operação

Sincronização do *Clock*

Sincronização × Coordenação

- Sincronização de processos
 - Garantimos que um processo espere até outro completar sua operação
- Sincronização de dados
 - Garantimos que dois conjuntos de dados são idênticos

Sincronização do *Clock*

Sincronização × Coordenação

- Sincronização de processos
 - Garantimos que um processo espere até outro completar sua operação
- Sincronização de dados
 - Garantimos que dois conjuntos de dados são idênticos
- Coordenação
 - Tem como objetivo gerenciar as interações e dependências entre as atividades em um SD
 - Coordenação encapsula Sincronização

Sincronização do *Clock*

Relógios Físicos

- Embora a frequência de oscilação do *clock* seja bastante estável, diferentes computadores terão frequências diferentes
- Fazendo com que *clocks* de máquinas distintas saiam de sincronia

Sincronização do *Clock*

Relógios Físicos

- Embora a frequência de oscilação do *clock* seja bastante estável, diferentes computadores terão frequências diferentes
- Fazendo com que *clocks* de máquinas distintas saiam de sincronia
- **Clock skew**: A diferença em valores de tempo entre 2 *clocks*

Sincronização do *Clock*

Relógios Físicos

- Embora a frequência de oscilação do *clock* seja bastante estável, diferentes computadores terão frequências diferentes
 - Fazendo com que *clocks* de máquinas distintas saiam de sincronia
 - **Clock skew**: A diferença em valores de tempo entre 2 *clocks*
- Alguns sistemas, contudo, precisam da hora exata
 - Ex: sistemas de tempo real
 - Precisam de relógios físicos externos

Sincronização do *Clock*

Relógios Físicos: UTC

- Universal Coordinated Time

Sincronização do *Clock*

Relógios Físicos: UTC

- Universal Coordinated Time
- Baseado no número de transições por segundo do átomo de césio 133
 - Bastante preciso

Sincronização do *Clock*

Relógios Físicos: UTC

- Universal Coordinated Time
- Baseado no número de transições por segundo do átomo de cézio 133
 - Bastante preciso
- Atualmente medido como a média de cerca de 50 relógios de cézio espalhados pelo mundo
 - Introduz um *segundo bissexto* de tempos em tempos para compensar o fato de que os dias solares estão se tornando cada vez maiores

Sincronização do *Clock*

Relógios Físicos: UTC

- O valor do UTC é enviado via *broadcast* por satélite e por ondas curtas de rádio

Sincronização do *Clock*

Relógios Físicos: UTC

- O valor do UTC é enviado via *broadcast* por satélite e por ondas curtas de rádio
- Satélites têm um acurácia de ± 0.5 ms

Sincronização do *Clock*

Relógios Físicos: UTC

- O valor do UTC é enviado via *broadcast* por satélite e por ondas curtas de rádio
- Satélites têm um acurácia de ± 0.5 ms
- As estações de rádio têm acurácia de ± 1 ms
 - Contudo, flutuações atmosféricas aleatórias podem afetar o comprimento do caminho do sinal
 - Sua precisão não é melhor que ± 10 ms

Sincronização do *Clock*

Relógios de *Software*

- **Precisão (π)**

- Limite dentro do qual tentamos deixar o desvio **entre dois relógios em quaisquer duas máquinas**:

$$\forall t, p, q : |C_p(t) - C_q(t)| \leq \pi$$

onde $C_p(t)$ é o horário do relógio **calculado** para a máquina p no **horário UTC** t

Sincronização do *Clock*

Relógios de *Software*

- **Precisão (π)**

- Limite dentro do qual tentamos deixar o desvio **entre dois relógios em quaisquer duas máquinas**:

$$\forall t, p, q : |C_p(t) - C_q(t)| \leq \pi$$

onde $C_p(t)$ é o horário do relógio **calculado** para a máquina p no **horário UTC** t

- Calculado?

Relógios de *Software*

- **Precisão (π)**

- Limite dentro do qual tentamos deixar o desvio **entre dois relógios em quaisquer duas máquinas**:

$$\forall t, p, q : |C_p(t) - C_q(t)| \leq \pi$$

onde $C_p(t)$ é o horário do relógio **calculado** para a máquina p no **horário UTC** t

- **Calculado?**

- A partir do número de tiques do relógio desde uma determinada data

Sincronização do *Clock*

Relógios de *Software*

- **Acurácia**

- No caso da **acurácia**, queremos manter o relógio limitado a um valor α :

$$\forall t, p : |C_p(t) - t| \leq \alpha \text{ (depende de referência externa de } t\text{)}$$

Sincronização do *Clock*

Relógios de *Software*

- **Acurácia**

- No caso da **acurácia**, queremos manter o relógio limitado a um valor α :

$$\forall t, p : |C_p(t) - t| \leq \alpha \text{ (depende de referência externa de } t\text{)}$$

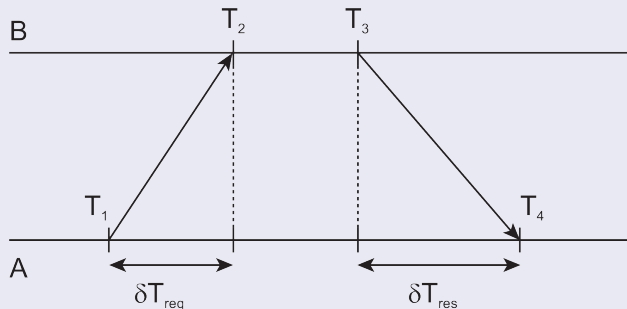
- **Sincronização**

- Interna: manter a **precisão** dos relógios
- Externa: manter a **acurácia** dos relógios
- Um conjunto de relógios com acurácia dentro do limite α estarão precisos dentro do limite $\pi = 2\alpha$

Sincronização do *Clock*

Network Time Protocol

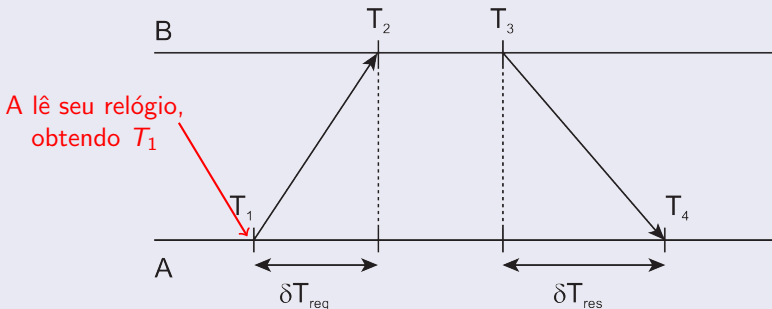
- Uma possibilidade para manter o relógio sincronizado é contatar um servidor de tempo
- O problema é o tempo gasto na transmissão da mensagem



Sincronização do *Clock*

Network Time Protocol

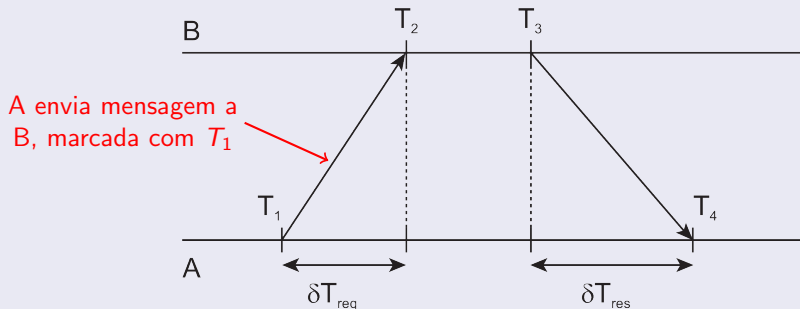
- Uma possibilidade para manter o relógio sincronizado é contatar um servidor de tempo
- O problema é o tempo gasto na transmissão da mensagem



Sincronização do *Clock*

Network Time Protocol

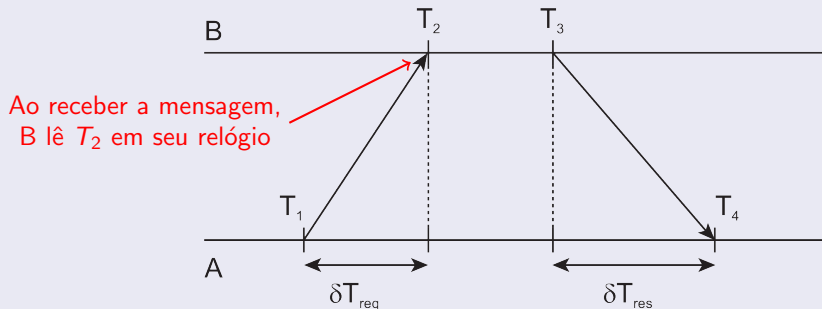
- Uma possibilidade para manter o relógio sincronizado é contatar um servidor de tempo
- O problema é o tempo gasto na transmissão da mensagem



Sincronização do *Clock*

Network Time Protocol

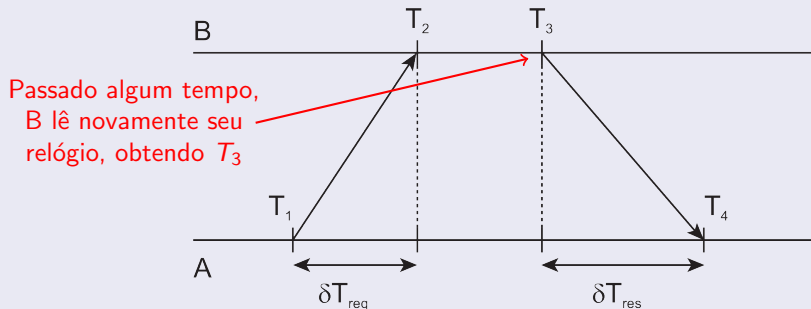
- Uma possibilidade para manter o relógio sincronizado é contatar um servidor de tempo
- O problema é o tempo gasto na transmissão da mensagem



Sincronização do *Clock*

Network Time Protocol

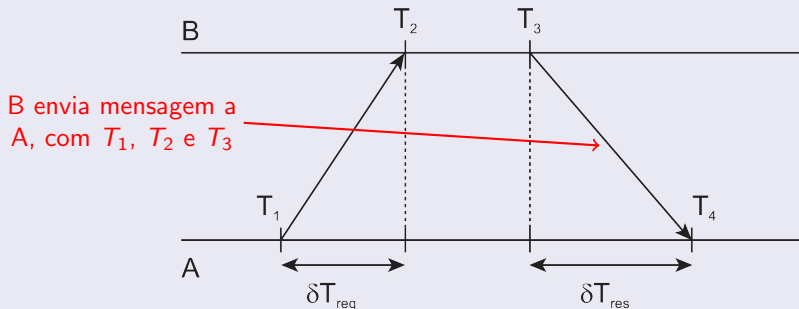
- Uma possibilidade para manter o relógio sincronizado é contatar um servidor de tempo
- O problema é o tempo gasto na transmissão da mensagem



Sincronização do *Clock*

Network Time Protocol

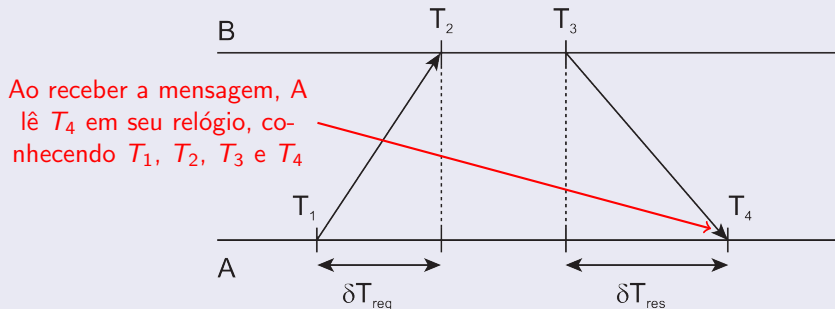
- Uma possibilidade para manter o relógio sincronizado é contatar um servidor de tempo
- O problema é o tempo gasto na transmissão da mensagem



Sincronização do *Clock*

Network Time Protocol

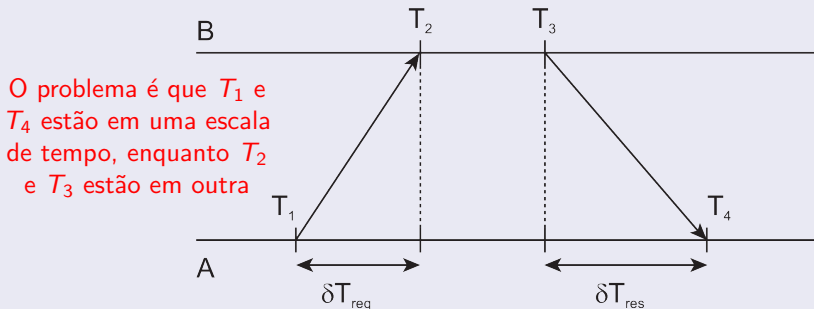
- Uma possibilidade para manter o relógio sincronizado é contatar um servidor de tempo
- O problema é o tempo gasto na transmissão da mensagem



Sincronização do *Clock*

Network Time Protocol

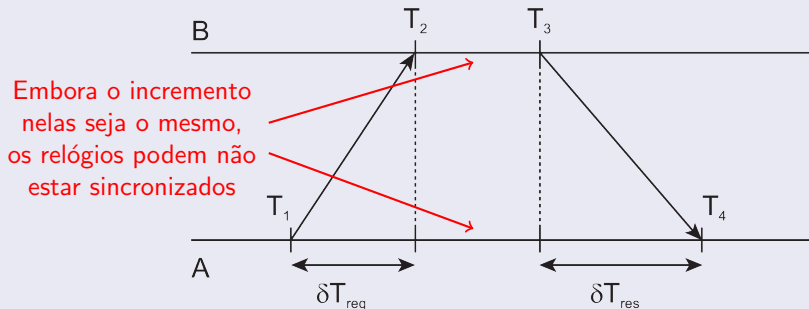
- Uma possibilidade para manter o relógio sincronizado é contatar um servidor de tempo
- O problema é o tempo gasto na transmissão da mensagem



Sincronização do *Clock*

Network Time Protocol

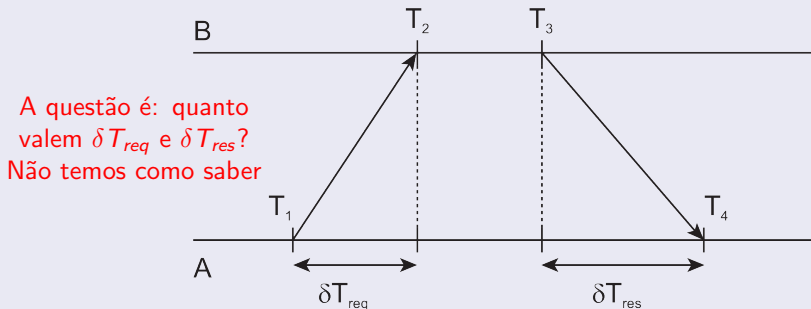
- Uma possibilidade para manter o relógio sincronizado é contatar um servidor de tempo
- O problema é o tempo gasto na transmissão da mensagem



Sincronização do *Clock*

Network Time Protocol

- Uma possibilidade para manter o relógio sincronizado é contatar um servidor de tempo
- O problema é o tempo gasto na transmissão da mensagem

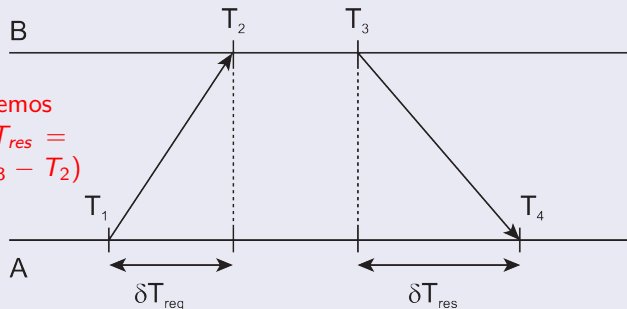


Sincronização do *Clock*

Network Time Protocol

- Uma possibilidade para manter o relógio sincronizado é contatar um servidor de tempo
- O problema é o tempo gasto na transmissão da mensagem

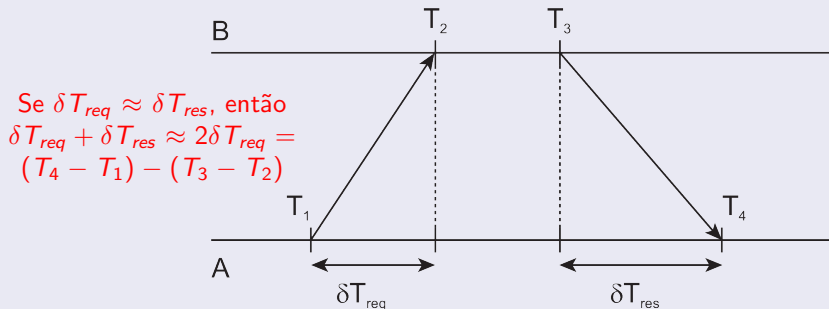
Contudo, sabemos
que $\delta T_{req} + \delta T_{res} =$
 $(T_4 - T_1) - (T_3 - T_2)$



Sincronização do *Clock*

Network Time Protocol

- Uma possibilidade para manter o relógio sincronizado é contatar um servidor de tempo
- O problema é o tempo gasto na transmissão da mensagem

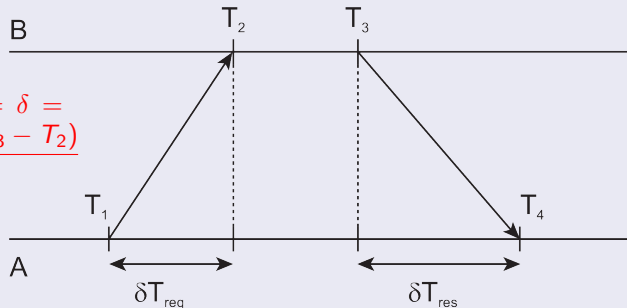


Sincronização do *Clock*

Network Time Protocol

- Uma possibilidade para manter o relógio sincronizado é contatar um servidor de tempo
- O problema é o tempo gasto na transmissão da mensagem

$$\text{Então } \delta T_{req} = \delta = \frac{(T_4 - T_1) - (T_3 - T_2)}{2}$$

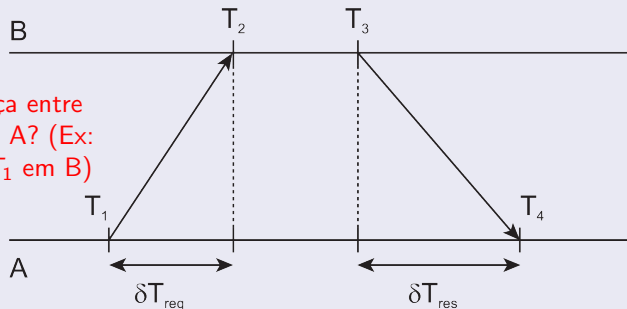


Sincronização do *Clock*

Network Time Protocol

- Uma possibilidade para manter o relógio sincronizado é contatar um servidor de tempo
- O problema é o tempo gasto na transmissão da mensagem

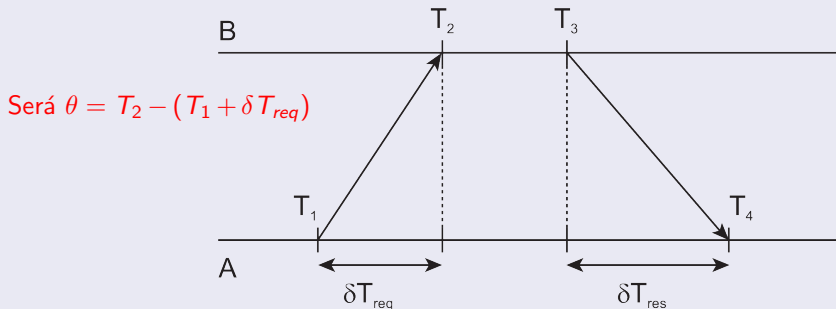
E qual a diferença entre os tempos de B e A? (Ex: o equivalente a T_1 em B)



Sincronização do *Clock*

Network Time Protocol

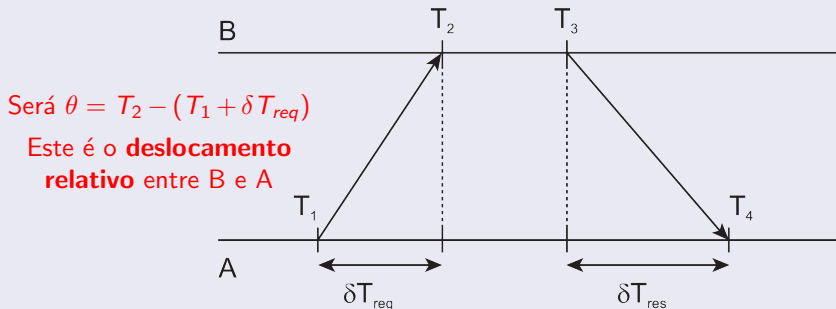
- Uma possibilidade para manter o relógio sincronizado é contatar um servidor de tempo
- O problema é o tempo gasto na transmissão da mensagem



Sincronização do *Clock*

Network Time Protocol

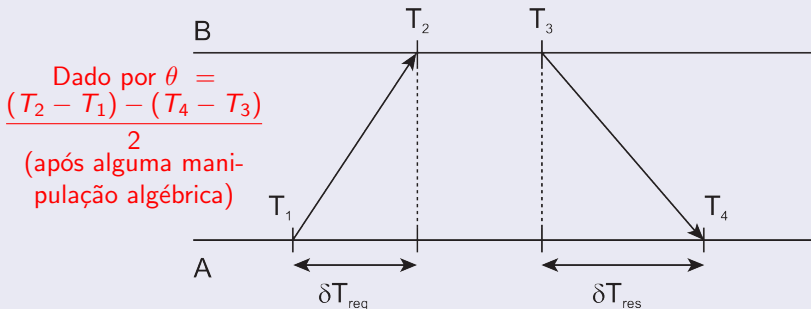
- Uma possibilidade para manter o relógio sincronizado é contatar um servidor de tempo
- O problema é o tempo gasto na transmissão da mensagem



Sincronização do *Clock*

Network Time Protocol

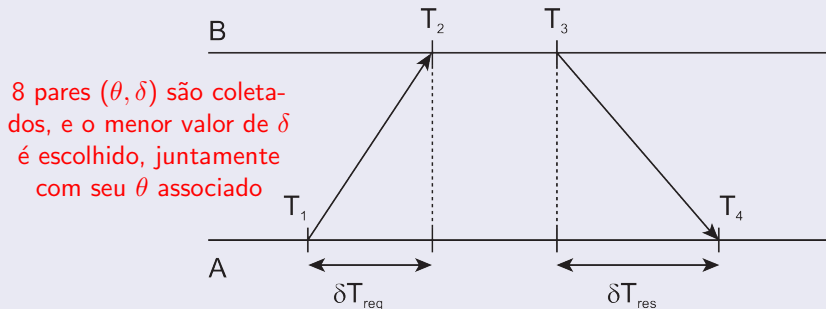
- Uma possibilidade para manter o relógio sincronizado é contatar um servidor de tempo
- O problema é o tempo gasto na transmissão da mensagem



Sincronização do *Clock*

Network Time Protocol

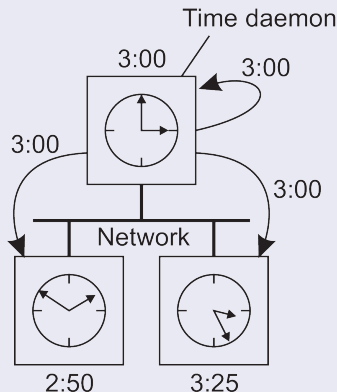
- Uma possibilidade para manter o relógio sincronizado é contatar um servidor de tempo
- O problema é o tempo gasto na transmissão da mensagem



Sincronização do *Clock*

Mantendo o tempo sem UTC

- O algoritmo do Berkeley Unix

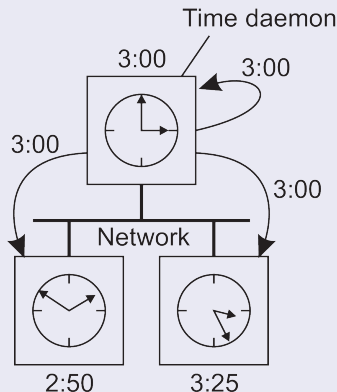


Sincronização do *Clock*

Mantendo o tempo sem UTC

- O algoritmo do Berkeley Unix

No Unix de Berkely, o servidor de tempo (*daemon* de tempo) periodicamente pede o horário local de todas as máquinas, enviando o seu



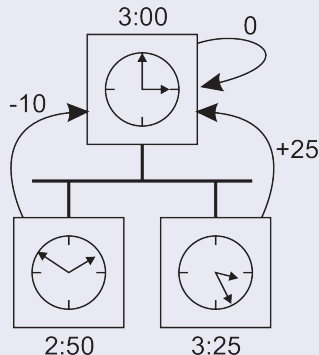
Sincronização do *Clock*

Mantendo o tempo sem UTC

- O algoritmo do Berkeley Unix

No Unix de Berkely, o servidor de tempo (*daemon* de tempo) periodicamente pede o horário local de todas as máquinas, enviando o seu

Estas calculam a diferença entre o horário do servidor e seus horários locais, retornando o resultado ao servidor



Sincronização do *Clock*

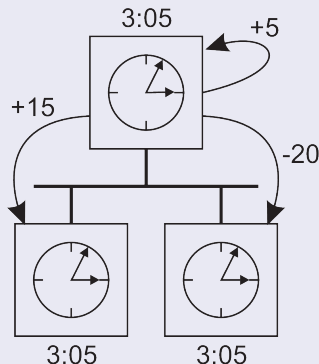
Mantendo o tempo sem UTC

- O algoritmo do Berkeley Unix

No Unix de Berkely, o servidor de tempo (*daemon* de tempo) periodicamente pede o horário local de todas as máquinas, enviando o seu

Estas calculam a diferença entre o horário do servidor e seus horários locais, retornando o resultado ao servidor

Com base nas respostas, ele calcula a diferença média ($\frac{-10+25+0}{3} \approx 5$) e diz a cada máquina como ajustar seu relógio

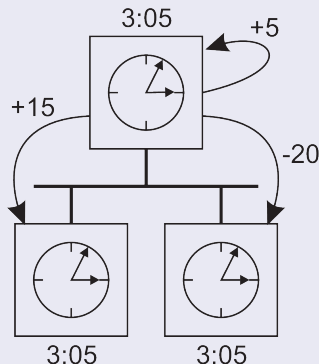


Sincronização do *Clock*

Mantendo o tempo sem UTC

- O algoritmo do Berkeley Unix

Método adequado para sistemas em que nenhuma máquina possui um receptor de UTC



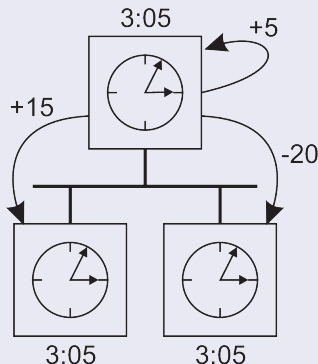
Sincronização do *Clock*

Mantendo o tempo sem UTC

- O algoritmo do Berkeley Unix

Método adequado para sistemas em que nenhuma máquina possui um receptor de UTC

E quando não há a necessidade de que o conjunto esteja de acordo com o tempo real – basta que as máquinas possuam o mesmo horário



Sincronização do *Clock*

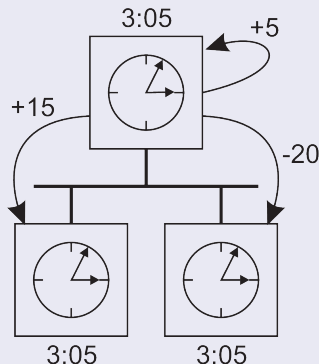
Mantendo o tempo sem UTC

- O algoritmo do Berkeley Unix

Método adequado para sistemas em que nenhuma máquina possui um receptor de UTC

E quando não há a necessidade de que o conjunto esteja de acordo com o tempo real – basta que as máquinas possuam o mesmo horário

Trata-se então de um algoritmo de sincronização interna do relógio

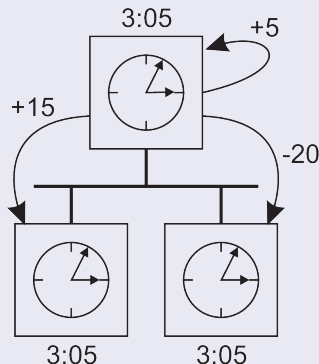


Sincronização do *Clock*

Mantendo o tempo sem UTC

- O algoritmo do Berkeley Unix

É **fundamental**, contudo, que os relógios **nunca** sejam atrasados



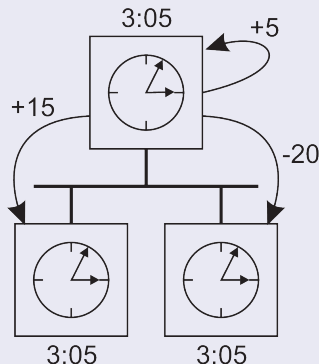
Sincronização do *Clock*

Mantendo o tempo sem UTC

- O algoritmo do Berkeley Unix

É **fundamental**, contudo, que os relógios **nunca** sejam atrasados

Sob risco de termos algum arquivo remoto com data anterior à sua fonte



Sincronização do *Clock*

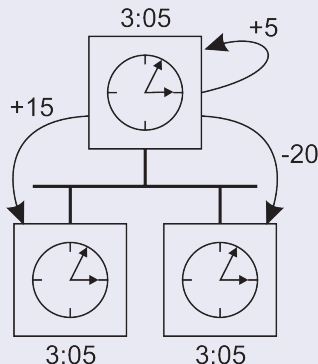
Mantendo o tempo sem UTC

- O algoritmo do Berkeley Unix

É **fundamental**, contudo, que os relógios **nunca** sejam atrasados

Sob risco de termos algum arquivo remoto com data anterior à sua fonte

Qualquer ajuste deve ser feito ou adiantando-se o relógio, ou então o desacelerando, até que a redução desejada seja gradualmente atingida

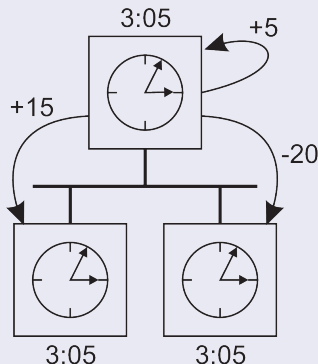


Sincronização do *Clock*

Mantendo o tempo sem UTC

- O algoritmo do Berkeley Unix

Para tal, basta fazer com que a rotina que trata a interrupção do relógio adicione menos ao tempo total



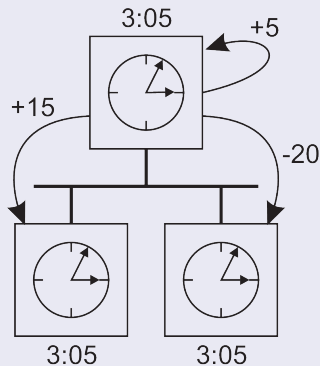
Sincronização do *Clock*

Mantendo o tempo sem UTC

- O algoritmo do Berkeley Unix

Para tal, basta fazer com que a rotina que trata a interrupção do relógio adicione menos ao tempo total

Assim, se 100 interrupções do *clock* são geradas a cada segundo pelo *hardware*, a rotina de tratamento adiciona, por exemplo, 9ms ao contador (em vez de 10ms), até que a correção seja feita



Sincronização e Coordenação

- Sincronização do *Clock*
- **Relógios Lógicos**
- Exclusão Mútua

Precisamos sempre do tempo?

- Se 2 processos não interagem, não é necessário que sincronizem seus relógios
 - A falta de sincronização não causará problemas

Precisamos sempre do tempo?

- Se 2 processos não interagem, não é necessário que sincronizem seus relógios
 - A falta de sincronização não causará problemas
- Na maior parte dos SDs, não importa se os processos concordam exatamente com o horário
 - Mas sim se concordam com **a ordem em que os eventos ocorrem**

Precisamos sempre do tempo?

- Se 2 processos não interagem, não é necessário que sincronizem seus relógios
 - A falta de sincronização não causará problemas
- Na maior parte dos SDs, não importa se os processos concordam exatamente com o horário
 - Mas sim se concordam com **a ordem em que os eventos ocorrem**
- Precisamos apenas de uma noção de ordem entre os eventos

A relação “aconteceu-antes” (*happened-before*)

- Se a e b são dois eventos de um mesmo processo e a ocorre antes de b , então $a \rightarrow b$
- Se a for o evento de envio de uma mensagem e b for o evento de recebimento desta mesma mensagem, então $a \rightarrow b$
- Se $a \rightarrow b$ e $b \rightarrow c$, então $a \rightarrow c$

A relação “aconteceu-antes” (*happened-before*)

- Se a e b são dois eventos de um mesmo processo e a ocorre antes de b , então $a \rightarrow b$
- Se a for o evento de envio de uma mensagem e b for o evento de recebimento desta mesma mensagem, então $a \rightarrow b$
- Se $a \rightarrow b$ e $b \rightarrow c$, então $a \rightarrow c$

Isso introduz uma noção de **ordem parcial dos eventos** em um sistema com processos executando concorrentemente

Relógio Lógico de Lamport

- Para implementar essa relação, associamos uma marcação de tempo $C(e)$ a cada evento e tal que:
 - P1 Se a e b são dois eventos no mesmo processo e $a \rightarrow b$, então $C(a) < C(b)$
 - P2 Se a corresponder ao envio de uma mensagem m e b ao recebimento desta mensagem, então também $C(a) < C(b)$

Relógio Lógico de Lamport

- Para implementar essa relação, associamos uma marcação de tempo $C(e)$ a cada evento e tal que:
 - P1 Se a e b são dois eventos no mesmo processo e $a \rightarrow b$, então $C(a) < C(b)$
 - P2 Se a corresponder ao envio de uma mensagem m e b ao recebimento desta mensagem, então também $C(a) < C(b)$
- Além disso, o tempo de relógio C deve sempre ir para frente, nunca para trás
 - Correções no tempo só podem ser feitas pela adição de um valor positivo

Relógio Lógico de Lamport

- E como podemos associar tempo a eventos quando não há um relógio global?
- Mantendo um conjunto de relógios lógicos **consistentes**, um para cada processo

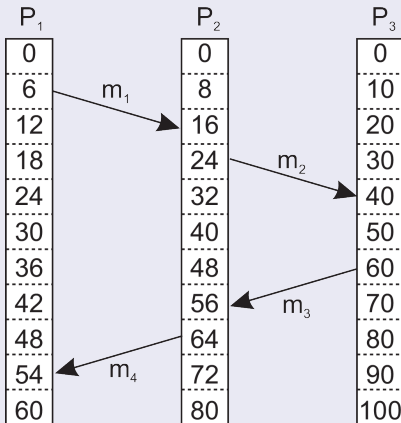
Relógio Lógico de Lamport

- E como podemos associar tempo a eventos quando não há um relógio global?
- Mantendo um conjunto de relógios lógicos **consistentes**, um para cada processo
- Os relógios lógicos de Lamport

Relógios Lógicos

Relógio Lógico de Lamport – Exemplo

Considere três processos em máquinas diferentes, cada uma com seu *clock*

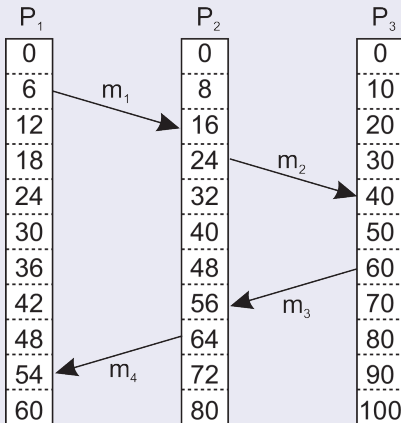


Relógios Lógicos

Relógio Lógico de Lamport – Exemplo

Considere três processos em máquinas diferentes, cada uma com seu *clock*

Suponha que esses *clocks* são implementados como **contadores**



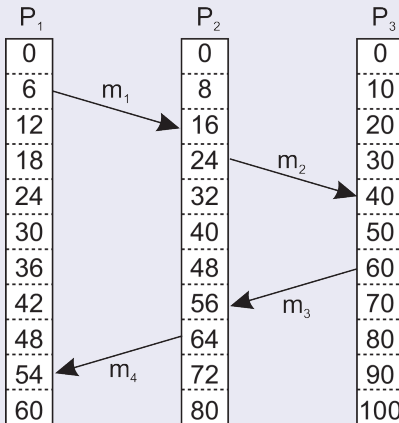
Relógios Lógicos

Relógio Lógico de Lamport – Exemplo

Considere três processos em máquinas diferentes, cada uma com seu *clock*

Suponha que esses *clocks* são implementados como **contadores**

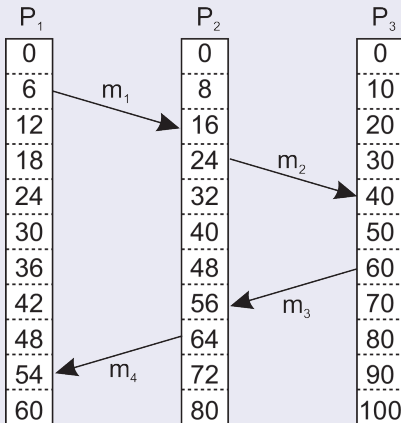
O contador é incrementado por um valor específico a cada T unidades de tempo



Relógios Lógicos

Relógio Lógico de Lamport – Exemplo

O valor desse incremento é diferente em cada máquina (6, 8 e 10)

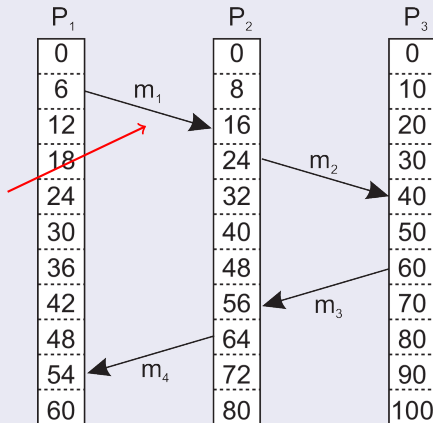


Relógios Lógicos

Relógio Lógico de Lamport – Exemplo

O valor desse incremento é diferente em cada máquina (6, 8 e 10)

No instante 6, P_1 envia a mensagem m_1 a P_2 , contendo esse 6



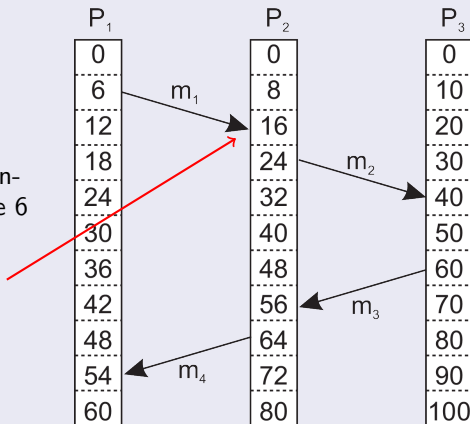
Relógios Lógicos

Relógio Lógico de Lamport – Exemplo

O valor desse incremento é diferente em cada máquina (6, 8 e 10)

No instante 6, P_1 envia a mensagem m_1 a P_2 , contendo esse 6

O *clock* em P_2 está em 16 quando m_1 chega



Relógios Lógicos

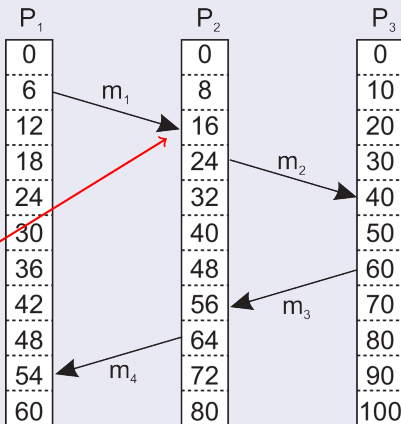
Relógio Lógico de Lamport – Exemplo

O valor desse incremento é diferente em cada máquina (6, 8 e 10)

No instante 6, P_1 envia a mensagem m_1 a P_2 , contendo esse 6

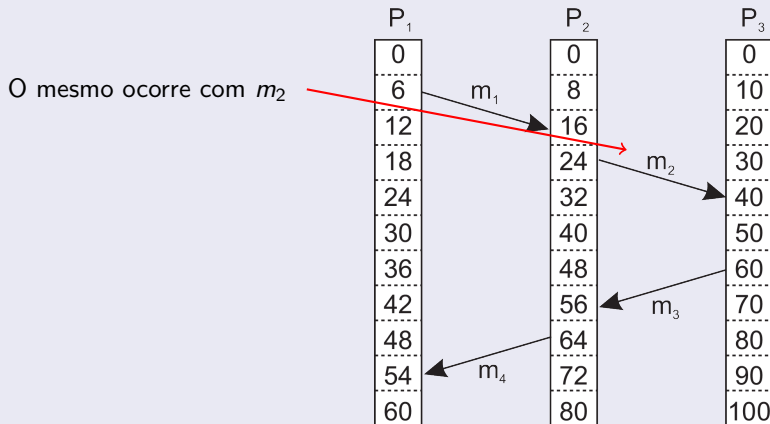
O *clock* em P_2 está em 16 quando m_1 chega

P_2 conclui então que m_1 gastou $16 - 6 = 10$ tiques em trânsito, algo perfeitamente possível



Relógios Lógicos

Relógio Lógico de Lamport – Exemplo

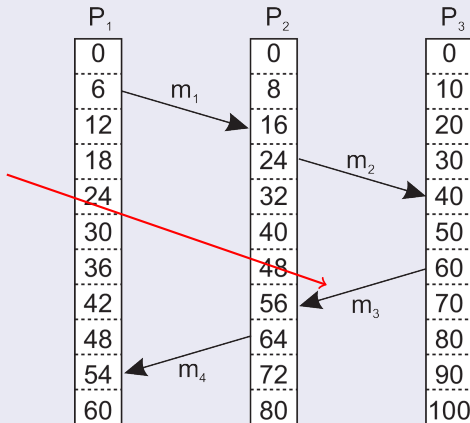


Relógios Lógicos

Relógio Lógico de Lamport – Exemplo

O mesmo ocorre com m_2

Contudo, m_3 deixa P_3 aos 60 e chega em P_2 aos 56, algo impossível



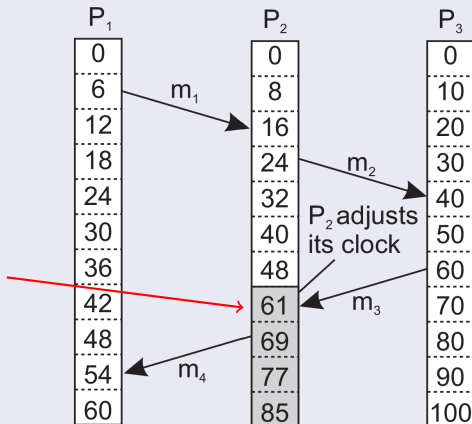
Relógios Lógicos

Relógio Lógico de Lamport – Exemplo

O mesmo ocorre com m_2

Contudo, m_3 deixa P_3 aos 60 e chega em P_2 aos 56, algo impossível

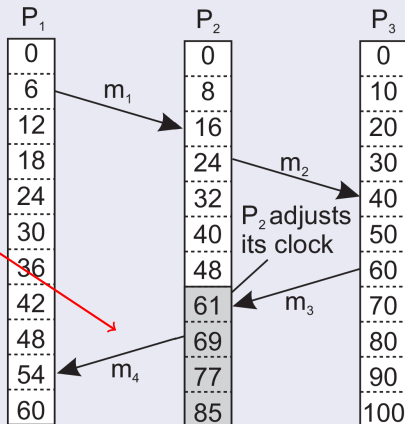
P_2 adianta então seu relógio para $60 + 1 = 61$



Relógios Lógicos

Relógio Lógico de Lamport – Exemplo

O problema se repete com m_4

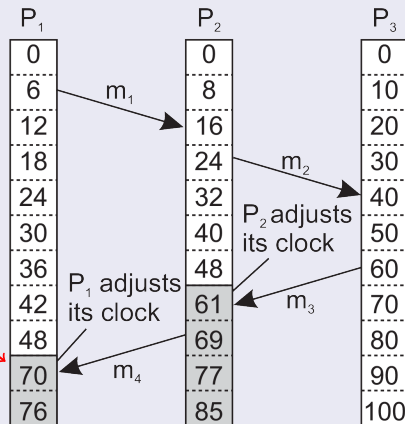


Relógios Lógicos

Relógio Lógico de Lamport – Exemplo

O problema se repete com m_4

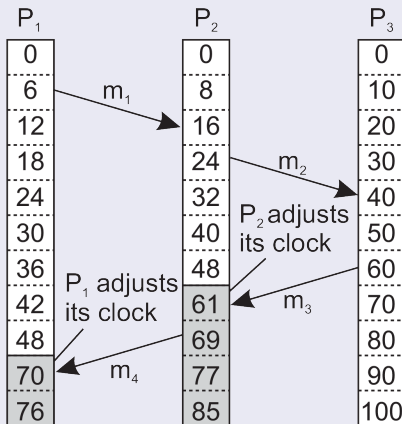
Fazendo com que P_1 ajuste
seu relógio para $69 + 1 = 70$



Relógios Lógicos

Relógio Lógico de Lamport – Exemplo

Temos então que, embora os *clocks* rodem com velocidades diferentes, o algoritmo de Lamport os corrige

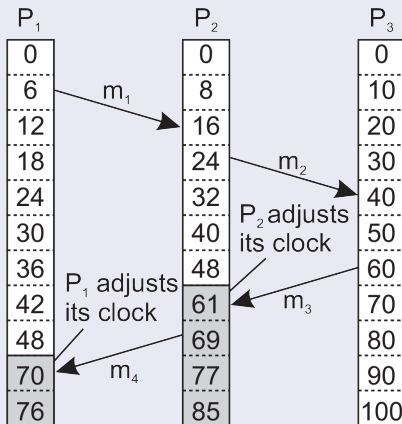


Relógios Lógicos

Relógio Lógico de Lamport – Exemplo

Temos então que, embora os *clocks* rodem com velocidades diferentes, o algoritmo de Lamport os corrige

De fato, os relógios de Lamport são **contadores de eventos**



Relógio Lógico de Lamport – Algoritmo

- Cada processo P_i mantém um contador C_i **local** e o atualiza de acordo com as seguintes regras:
 1. Antes de executar um evento, P_i faz $C_i \leftarrow C_i + 1$
 2. Toda vez que uma mensagem m for **enviada** por P_i , a mensagem deve receber um *timestamp* $ts(m) = C_i$
 3. Sempre que uma mensagem m for **recebida** por P_j , este atualizará seu contador local para $C_j \leftarrow \max\{C_j, ts(m)\}$ e executará o passo 1 antes de repassar m à aplicação

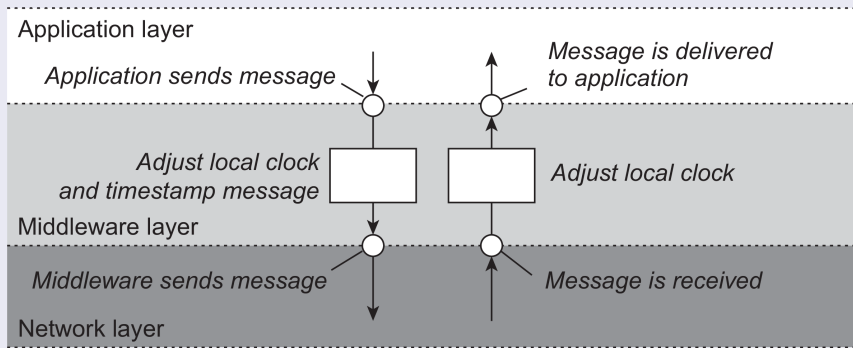
Relógio Lógico de Lamport – Algoritmo

- Note que
 - A propriedade **P1** é satisfeita por (1) e **P2** por (2) e (3)
 - Ainda assim pode acontecer de dois eventos ocorrerem ao mesmo tempo. **Desempate usando os IDs dos processos**
1. Antes de executar um evento, P_i faz $C_i \leftarrow C_i + 1$
 2. Toda vez que uma mensagem m for **enviada** por P_i , a mensagem deve receber um *timestamp* $ts(m) = C_i$
 3. Sempre que uma mensagem m for **recebida** por P_j , este atualizará seu contador local para $C_j \leftarrow \max\{C_j, ts(m)\}$ e executará o passo 1 antes de repassar m à aplicação

Relógios Lógicos

Relógio Lógico de Lamport

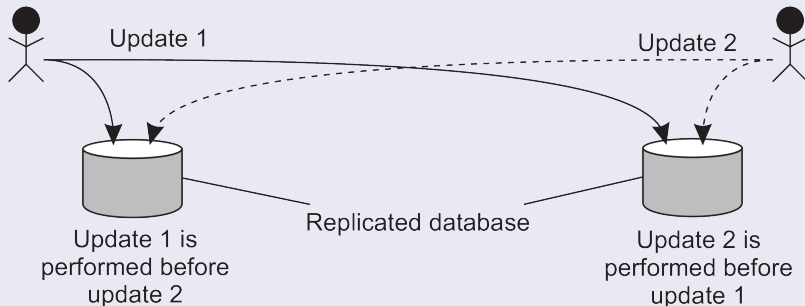
- Esses ajustes ocorrem na camada do *middleware*, onde os relógios de Lamport residem



Relógios Lógicos de Lamport

Multicasting totalmente ordenado

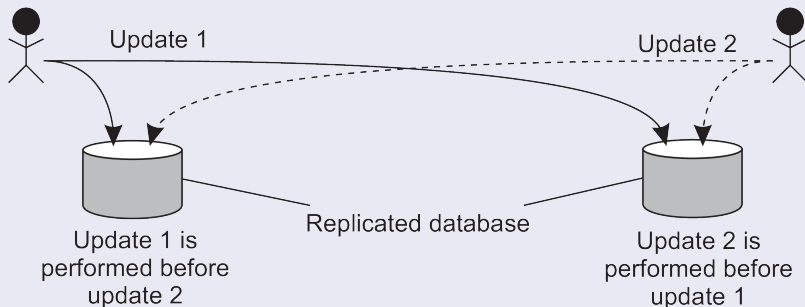
Considere a situação na qual uma base de dados foi replicada



Relógios Lógicos de Lamport

Multicasting totalmente ordenado

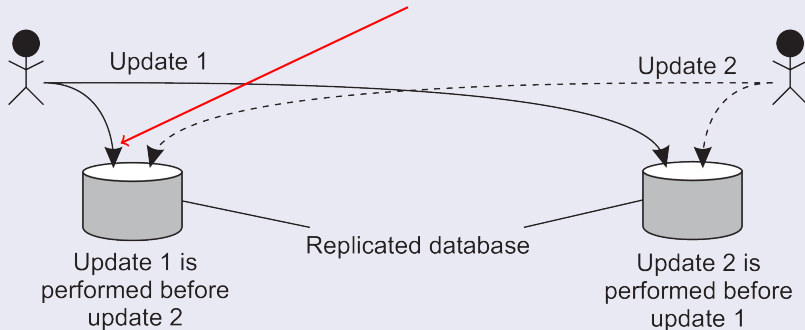
Atualizações concorrentes precisam ser vistas na mesma ordem por todos



Relógios Lógicos de Lamport

Multicasting totalmente ordenado

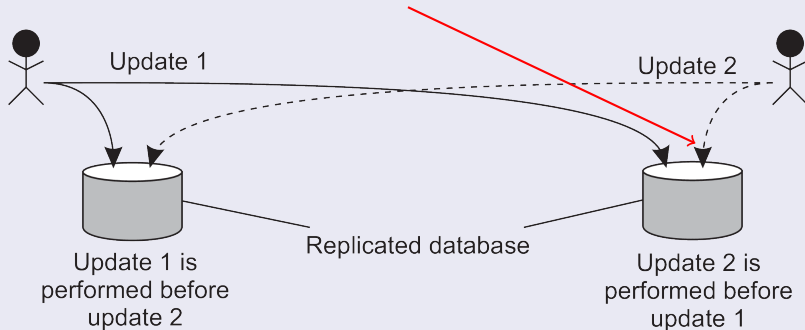
Imagine que alguém deposite R\$ 100,00 em sua conta, que conta com um saldo de R\$ 1.000,00



Relógios Lógicos de Lamport

Multicasting totalmente ordenado

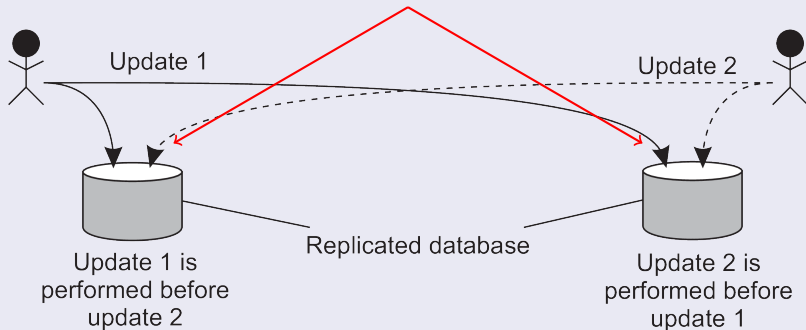
Ao mesmo tempo, um empregado do banco atualiza a conta, adicionando 1% de juros ao saldo



Relógios Lógicos de Lamport

Multicasting totalmente ordenado

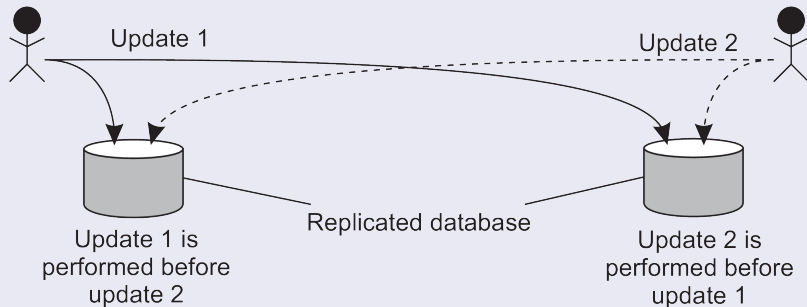
Ambas operações, contudo, levam um certo tempo para serem atualizadas em todas as réplicas



Relógios Lógicos de Lamport

Multicasting totalmente ordenado

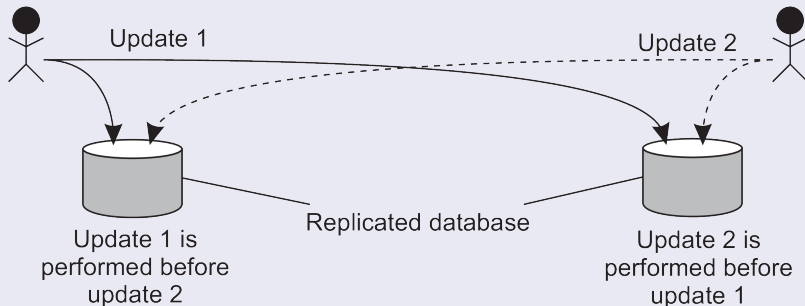
O que deixa as bases em um estado inconsistente, com R\$ 1.111,00 e R\$ 1.110,00, respectivamente



Relógios Lógicos de Lamport

Multicasting totalmente ordenado

Multicast totalment ordenado: *multicast* em que todas as mensagens são entregues na mesma ordem a cada receptor



Relógios Lógicos de Lamport

Multicasting totalmente ordenado

- E como conseguimos isso?

Relógios Lógicos de Lamport

Multicasting totalmente ordenado

- E como conseguimos isso?
 - O processo P_i envia uma mensagem m_i para todos os outros
 - A mensagem possui um *timestamp* com o horário lógico atual de P_i
 - A mensagem é colocada em sua fila local f_i

Relógios Lógicos de Lamport

Multicasting totalmente ordenado

- E como conseguimos isso?
 - O processo P_i envia uma mensagem m_i para todos os outros
 - A mensagem possui um *timestamp* com o horário lógico atual de P_i
 - A mensagem é colocada em sua fila local f_i
 - Toda mensagem que chegar a P_j é colocada na fila f_j
ordenada conforme seu timestamp
 - O receptor então faz o *multicasting* de uma confirmação a todos os outros processos

Relógios Lógicos de Lamport

Multicasting totalmente ordenado

- E como conseguimos isso?
 - O processo P_i envia uma mensagem m_i para todos os outros
 - A mensagem possui um *timestamp* com o horário lógico atual de P_i
 - A mensagem é colocada em sua fila local f_i
 - Toda mensagem que chegar a P_j é colocada na fila f_j
ordenada conforme seu timestamp
 - O receptor então faz o *multicasting* de uma confirmação a todos os outros processos
- Note que, se seguirmos o algoritmo de Lamport, o *timestamp* da mensagem recebida é menor que o da confirmação

Relógios Lógicos de Lamport

Multicasting totalmente ordenado

- Um processo P_j passa uma mensagem m_i à aplicação que ele está rodando somente se:
 - A mensagem estiver no início de sua fila f_j (é a mais antiga);
e

Relógios Lógicos de Lamport

Multicasting totalmente ordenado

- Um processo P_j passa uma mensagem m_i à aplicação que ele está rodando somente se:
 - A mensagem estiver no início de sua fila f_j (é a mais antiga); e
 - Foi confirmada por todos os demais processos
 - Ou seja, para cada processo P_k , existe uma mensagem m_k em f_j com um *timestamp* maior

Relógios Lógicos de Lamport

Multicasting totalmente ordenado

- Um processo P_j passa uma mensagem m_i à aplicação que ele está rodando somente se:
 - A mensagem estiver no início de sua fila f_j (é a mais antiga); e
 - Foi confirmada por todos os demais processos
 - Ou seja, para cada processo P_k , existe uma mensagem m_k em f_j com um *timestamp* maior
- A mensagem é então removida de f_j , e entregue à aplicação
 - Suas confirmações associadas são também removidas

Multicasting totalmente ordenado – Observações

- Se uma mensagem m ficar pronta em um servidor S , m foi recebida por todos os outros servidores (que enviaram ACKs dizendo que m foi recebida)

Relógios Lógicos de Lamport

Multicasting totalmente ordenado – Observações

- Se uma mensagem m ficar pronta em um servidor S , m foi recebida por todos os outros servidores (que enviaram ACKs dizendo que m foi recebida)
- Se n é uma mensagem originada no mesmo lugar que m e for enviada antes de m , então todos receberão n antes de m e n ficará no topo da fila antes de m
- Terá um *timestamp* menor, e a ordem será propagada pelo algoritmo de Lamport

Multicasting totalmente ordenado – Observações

- Se n , contudo, for originada em outro lugar, é um pouco mais complicado
 - Pode ser que m e n cheguem em ordem diferente nos servidores
 - Mas é certeza que antes de tirar um deles da fila, ele terá que receber os ACKs de todos os outros servidores
 - E isso permite comparar os valores dos relógios e entregar as mensagens na ordem geral dos relógios

Relógios vetoriais

- Os relógios de Lamport fazem com que todos os eventos em um SD estejam ordenados
- De modo que, se a ocorre antes de b , então $C(a) < C(b)$, e este será posicionado antes de b

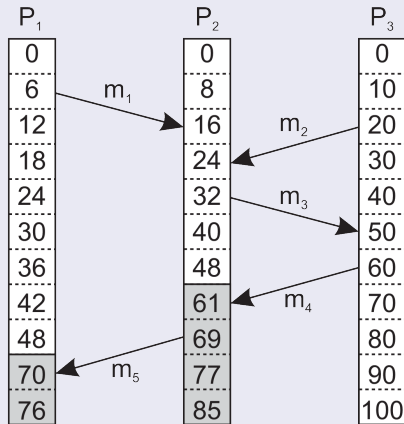
Relógios vetoriais

- Os relógios de Lamport fazem com que todos os eventos em um SD estejam ordenados
 - De modo que, se a ocorre antes de b , então $C(a) < C(b)$, e este será posicionado antes de b
- Mas $A \Rightarrow B$ não significa também que $B \Rightarrow A$
 - Ou seja, $C(a) < C(b)$ não implica a ter acontecido antes de b
 - Mesmo que o contrário seja verdade

Relógios Lógicos

Relógios vetoriais

Seja $T_e(m_i)$ o horário lógico de envio de m_i , e $T_r(m_i)$ o horário lógico de seu recebimento

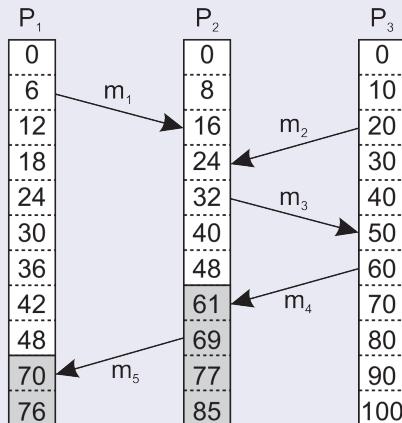


Relógios Lógicos

Relógios vetoriais

Seja $T_e(m_i)$ o horário lógico de envio de m_i , e $T_r(m_i)$ o horário lógico de seu recebimento

Por construção, temos que $T_e(m_i) < T_r(m_i)$



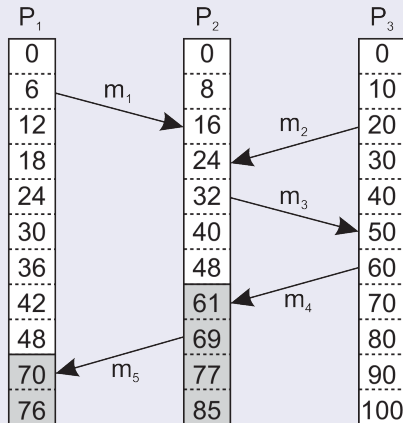
Relógios Lógicos

Relógios vetoriais

Seja $T_e(m_i)$ o horário lógico de envio de m_i , e $T_r(m_i)$ o horário lógico de seu recebimento

Por construção, temos que $T_e(m_i) < T_r(m_i)$

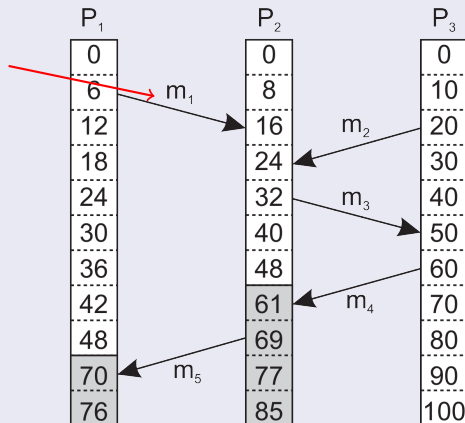
Mas podemos realmente afirmar que $T_e(m_i) < T_r(m_j)$ quando $m_i \neq m_j$?



Relógios Lógicos

Relógios vetoriais

Por exemplo, sabemos
que $T_e(m_1) < T_r(m_1)$

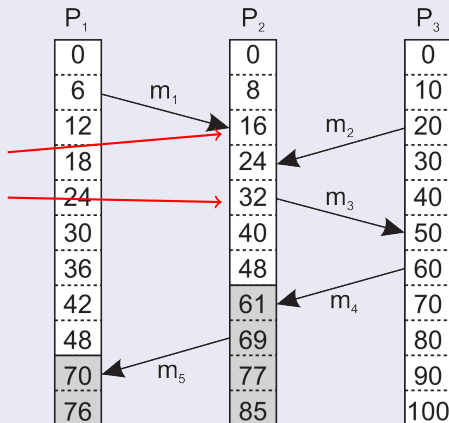


Relógios Lógicos

Relógios vetoriais

Por exemplo, sabemos
que $T_e(m_1) < T_r(m_1)$

E que $T_r(m_1) < T_e(m_3)$,
dado que estes eventos
ocorreram em P_2



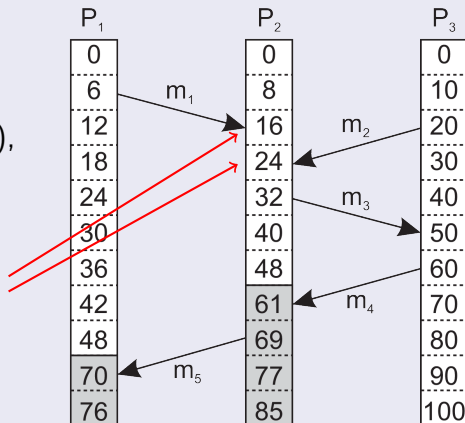
Relógios Lógicos

Relógios vetoriais

Por exemplo, sabemos
que $T_e(m_1) < T_r(m_1)$

E que $T_r(m_1) < T_e(m_3)$,
dado que estes eventos
ocorreram em P_2

Também sabemos que
 $T_r(m_1) < T_r(m_2)$



Relógios Lógicos

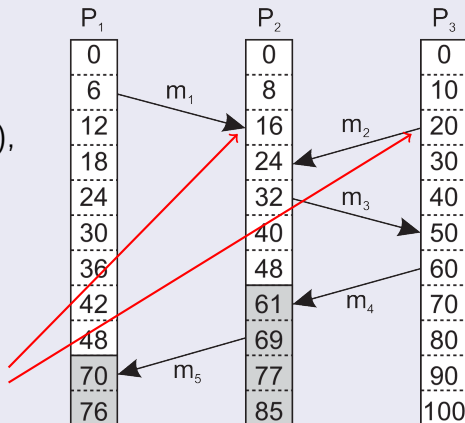
Relógios vetoriais

Por exemplo, sabemos
que $T_e(m_1) < T_r(m_1)$

E que $T_r(m_1) < T_e(m_3)$,
dado que estes even-
tos ocorreram em P_2

Também sabemos que
 $T_r(m_1) < T_r(m_2)$

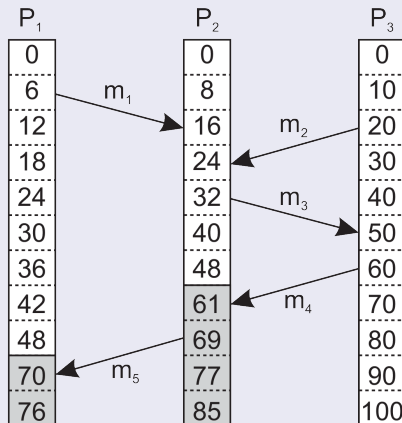
Mas o que dizer de
 $T_r(m_1)$ e $T_e(m_2)$



Relógios Lógicos

Relógios vetoriais

O problema é que os relógios de Lamport não capturam **causalidade**

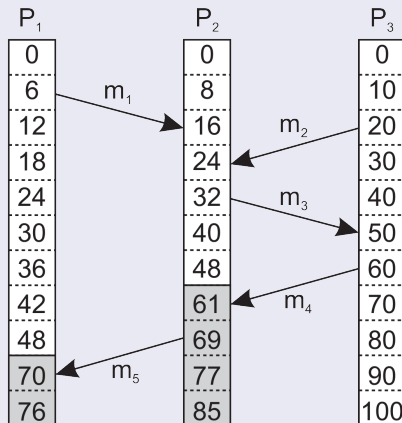


Relógios Lógicos

Relógios vetoriais

O problema é que os relógios de Lamport não capturam **causalidade**

Para isso precisamos de **relógios vetoriais**



Relógios vetoriais

- Cada processo P_i mantém um vetor VC_i tal que
 - $VC_i[i]$ é o número de eventos que ocorreram até o momento em P_i (seu relógio lógico)
 - Se $VC_i[j] = k$, então P_i sabe que k eventos ocorreram em P_j (seu conhecimento do tempo local em P_j)

Relógios vetoriais

- Cada processo P_i mantém um vetor VC_i tal que
 - $VC_i[i]$ é o número de eventos que ocorreram até o momento em P_i (seu relógio lógico)
 - Se $VC_i[j] = k$, então P_i sabe que k eventos ocorreram em P_j (seu conhecimento do tempo local em P_i)

VC_3 :

3	0	1	0
1	2	3	4

Relógios Lógicos

Relógios vetoriais

- Cada processo P_i mantém um vetor VC_i tal que
 - $VC_i[i]$ é o número de eventos que ocorreram até o momento em P_i (seu relógio lógico)
 - Se $VC_i[j] = k$, então P_i sabe que k eventos ocorreram em P_j (seu conhecimento do tempo local em P_j)

VC_3 :

3	0	1	0
1	2	3	4

Número de eventos ocorridos em P_3


Relógios Lógicos

Relógios vetoriais

- Cada processo P_i mantém um vetor VC_i tal que
 - $VC_i[i]$ é o número de eventos que ocorreram até o momento em P_i (seu relógio lógico)
 - Se $VC_i[j] = k$, então P_i sabe que k eventos ocorreram em P_j (seu conhecimento do tempo local em P_j)

VC_3 :

3	0	1	0
1	2	3	4



Número de eventos em P_1 conhecidos por P_3

Relógios vetoriais: Manutenção

1. Antes da execução de um evento, P_i faz
 $VC_i[i] \leftarrow VC_i[i] + 1$
 - Registra assim um novo evento em P_i

Relógios vetoriais: Manutenção

1. Antes da execução de um evento, P_i faz
 $VC_i[i] \leftarrow VC_i[i] + 1$
 - Registra assim um novo evento em P_i
2. Quando o processo P_i envia uma mensagem m para P_j , ele define o vetor de *timestamps* de m – $ts(m)$ – como sendo VC_i (após executar o passo 1)
 - Ou seja, $ts(m) \leftarrow VC_i$
 - Assim ele também registra o envio da mensagem como um evento em P_i

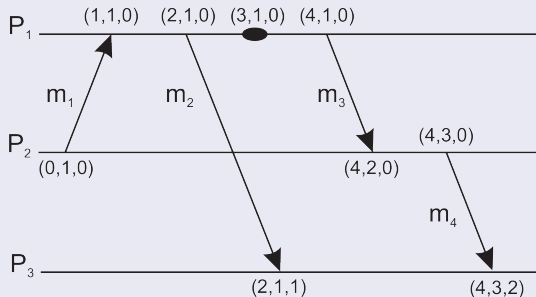
Relógios vetoriais: Manutenção

3. Ao receber uma mensagem m , o processo P_j faz
$$VC_j[k] \leftarrow \max(VC_j[k], ts(m)[k]), \forall k$$
 - Após isso executa o passo 1, registrando o recebimento da mensagem, e então a envia à aplicação

Relógios Lógicos

Relógios vetoriais: Manutenção

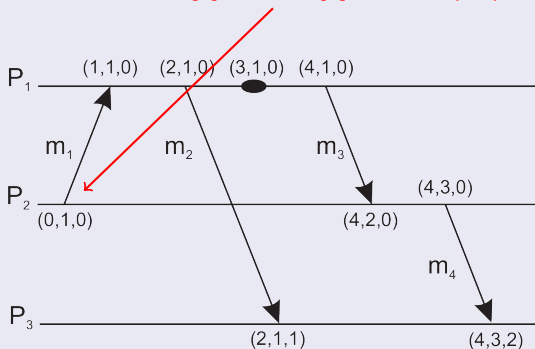
Considere os seguintes processos, com VC_1 , VC_2 e $VC_3 = (0, 0, 0)$



Relógios Lógicos

Relógios vetoriais: Manutenção

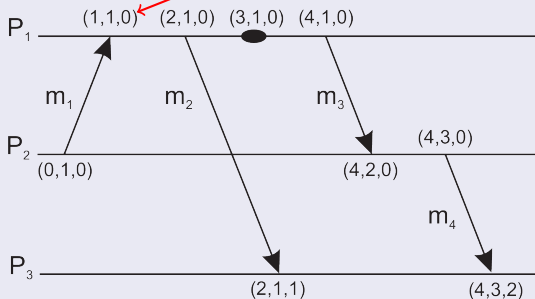
Ao enviar m_1 , P_2 faz $VC_2[2] \leftarrow VC_2[2] + 1$ e $ts(m_1) \leftarrow (0, 1, 0)$



Relógios Lógicos

Relógios vetoriais: Manutenção

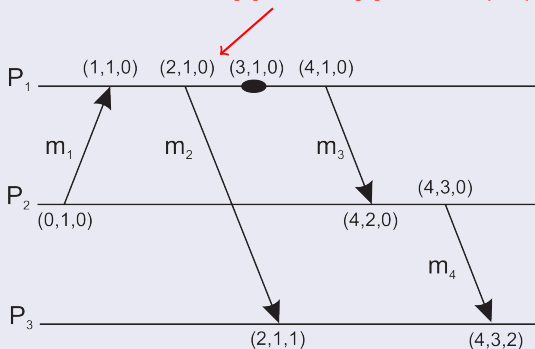
Ao receber m_1 , P_1 faz $VC_1[1] \leftarrow VC_1[1] + 1$, atualiza VC_1 com $VC_1[k] \leftarrow \max(VC_1[k], ts(m)[k]), 1 \leq k \leq 3$, e entrega m_1 à aplicação



Relógios Lógicos

Relógios vetoriais: Manutenção

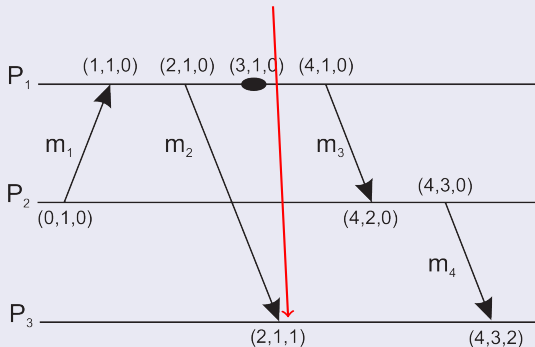
Antes de enviar m_2 , P_1 faz $VC_1[1] \leftarrow VC_1[1] + 1$ e $ts(m_2) \leftarrow (2, 1, 0)$



Relógios Lógicos

Relógios vetoriais: Manutenção

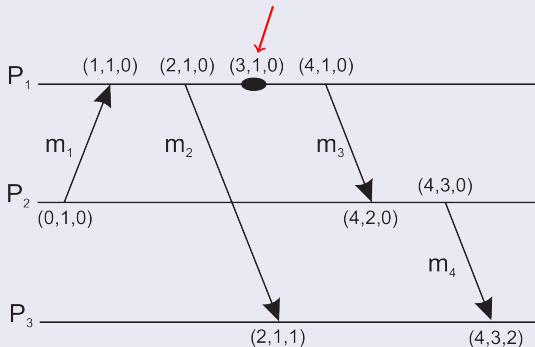
Ao receber m_2 , P_3 faz $VC_3[3] \leftarrow VC_3[3] + 1$, atualiza VC_3 com $VC_3[k] \leftarrow \max(VC_3[k], ts(m)[k]), 1 \leq k \leq 3$, e entrega m_2 à aplicação



Relógios Lógicos

Relógios vetoriais: Manutenção

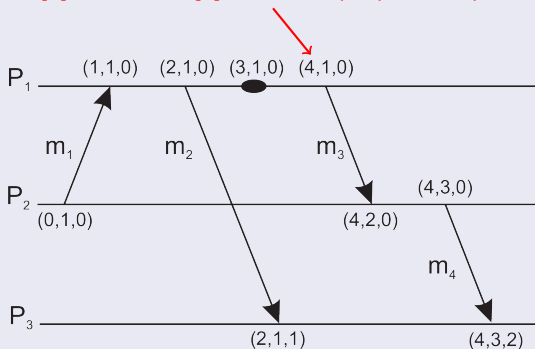
Enquanto isso, outro evento ocorre em P_1 , fazendo com que $VC_1[1] \leftarrow VC_1[1] + 1 \Rightarrow VC_1 : (3, 1, 0)$



Relógios Lógicos

Relógios vetoriais: Manutenção

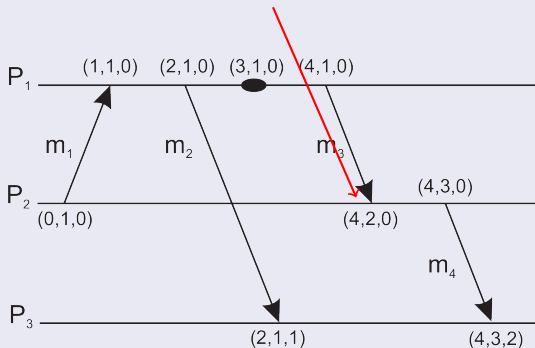
Na sequência, e antes de enviar m_3 , P_1 faz
 $VC_1[1] \leftarrow VC_1[1] + 1$ e $ts(m_3) \leftarrow (4, 1, 0)$



Relógios Lógicos

Relógios vetoriais: Manutenção

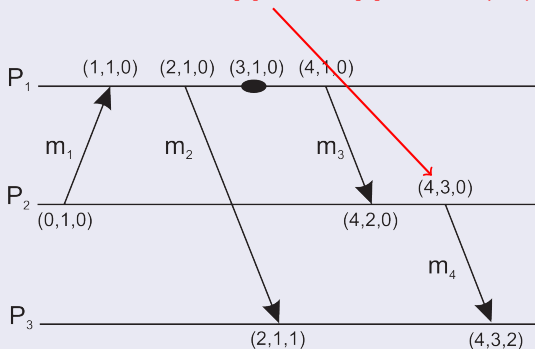
Ao receber m_3 , P_2 faz $VC_2[2] \leftarrow VC_2[2] + 1$, $VC_2[k] \leftarrow \max(VC_2[k], ts(m)[k]), 1 \leq k \leq 3$, e entrega m_3 à aplicação



Relógios Lógicos

Relógios vetoriais: Manutenção

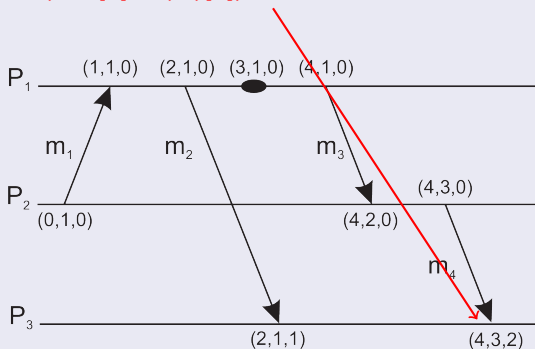
Antes de enviar m_4 , P_2 faz $VC_2[2] \leftarrow VC_2[2] + 1$ e $ts(m_4) \leftarrow (4, 3, 0)$



Relógios Lógicos

Relógios vetoriais: Manutenção

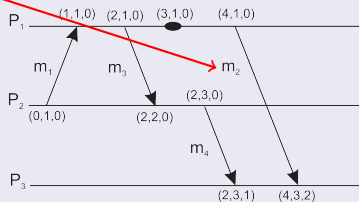
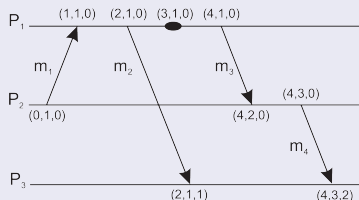
Finalmente, ao receber m_4 , P_3 faz $VC_3[3] \leftarrow VC_3[3] + 1$,
 $VC_3[k] \leftarrow \max(VC_3[k], ts(m)[k]), 1 \leq k \leq 3$, e entrega m_4 à aplicação



Relógios Lógicos

Relógios vetoriais: Precedência \times Dependência

Imagine agora que atrasamos o envio de m_2 para após o envio de m_3

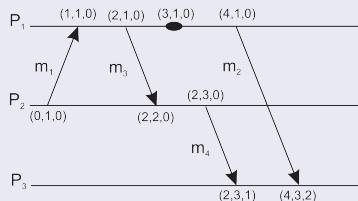
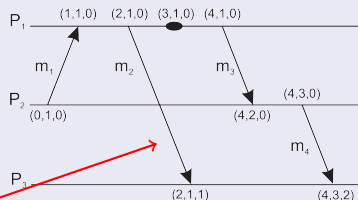


Relógios Lógicos

Relógios vetoriais: Precedência \times Dependência

Imagine agora que atrasamos o envio de m_2 para após o envio de m_3

Temos então que $ts(m_2) < ts(m_4)$, e m_2 pode preceder causalmente m_4



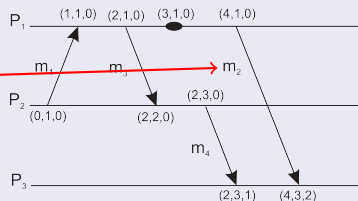
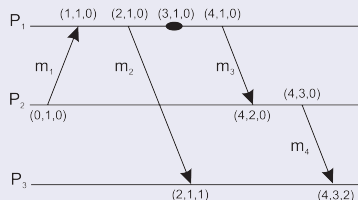
Relógios Lógicos

Relógios vetoriais: Precedência \times Dependência

Imagine agora que atrasamos o envio de m_2 para após o envio de m_3

Temos então que $ts(m_2) < ts(m_4)$, e m_2 pode preceder causalmente m_4

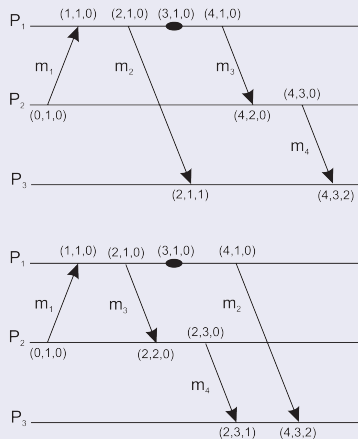
Equanto que $ts(m_4) < ts(m_2)$, e m_2 e m_4 podem estar em conflito



Relógios Lógicos

Relógios vetoriais: Precedência \times Dependência

E o que significa dizer que *a* precede causalmente (ou potencialmente causa) *b*?

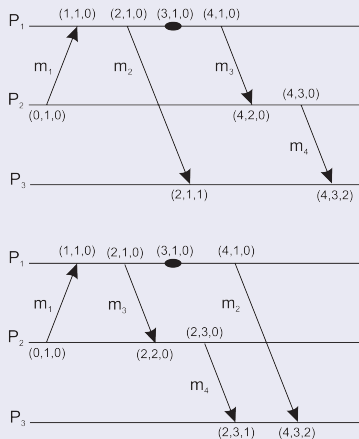


Relógios Lógicos

Relógios vetoriais: Precedência \times Dependência

E o que significa dizer que a **precede causalmente** (ou potencialmente causa) b ?

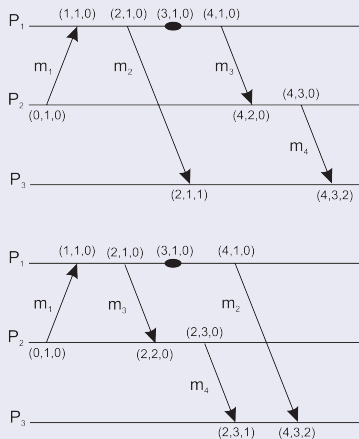
Significa que b **pode** depender causalmente de a , já que há informação de a que pode ter sido propagada para b



Relógios Lógicos

Relógios vetoriais: Precedência \times Dependência

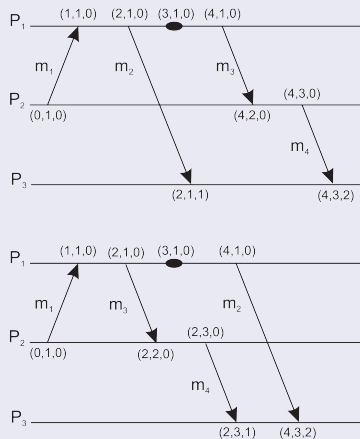
Dizemos que b pode **depende causalmente** de a se $ts(a) < ts(b)$, tal que:



Relógios vetoriais: Precedência \times Dependência

Dizemos que b pode **depende causalmente** de a se $ts(a) < ts(b)$, tal que:

- Para todo k , $ts(a)[k] \leq ts(b)[k]$; e

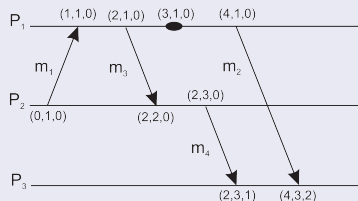
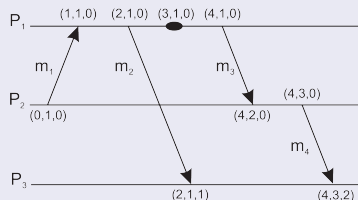


Relógios Lógicos

Relógios vetoriais: Precedência \times Dependência

Dizemos que b pode **depende causalmente** de a se $ts(a) < ts(b)$, tal que:

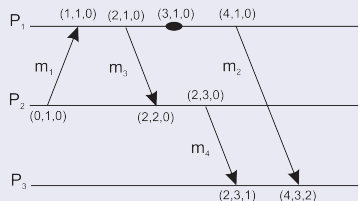
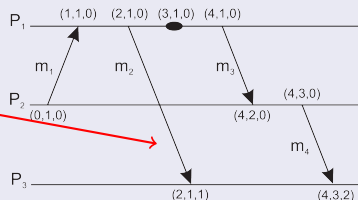
- Para todo k , $ts(a)[k] \leq ts(b)[k]$; e
- Existe pelo menos um índice k' para o qual $ts(a)[k'] < ts(b)[k']$



Relógios Lógicos

Relógios vetoriais: Precedência \times Dependência

Ao usar relógios vetoriais, P_3 pode detectar se m_4 pode ser causalmente dependente de m_2

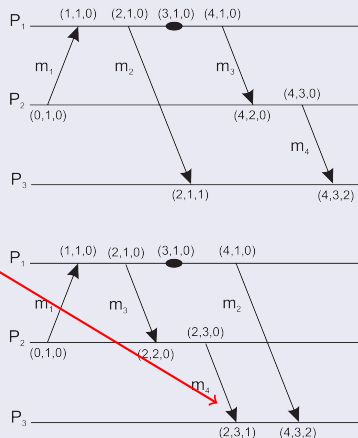


Relógios Lógicos

Relógios vetoriais: Precedência \times Dependência

Ao usar relógios vetoriais, P_3 pode detectar se m_4 pode ser causalmente dependente de m_2

Ou se pode ser um conflito potencial



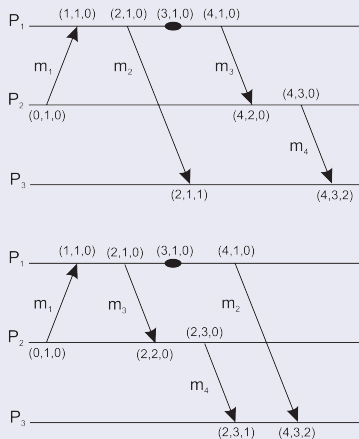
Relógios Lógicos

Relógios vetoriais: Precedência \times Dependência

Ao usar relógios vetoriais, P_3 pode detectar se m_4 pode ser causalmente dependente de m_2

Ou se pode ser um conflito potencial

Sem saber o conteúdo da mensagem, contudo, não é realmente possível afirmar com certeza a existência de conflito ou relação causal



Relógios vetoriais

- Note que
 - Quando P_j recebe uma mensagem m de P_i , com *timestamp* $ts(m)$, ele sabe o número de eventos que precedem o envio de m em P_i
 - Além de saber quantos eventos em outros processos, conhecidos por P_i , precederam o envio de m

Relógios vetoriais

- Note que
 - Quando P_j recebe uma mensagem m de P_i , com *timestamp* $ts(m)$, ele sabe o número de eventos que precedem o envio de m em P_i
 - Além de saber quantos eventos em outros processos, conhecidos por P_i , precederam o envio de m

$ts(m)$ diz então ao receptor quantos eventos em outros processos precederam o envio de m , e dos quais m pode depender causalmente

Multicast ordenado causalmente

- Podemos agora garantir que uma mensagem seja entregue somente se todas as mensagens que a causalmente precedem tiverem sido entregues

Multicast ordenado causalmente

- Podemos agora garantir que uma mensagem seja entregue somente se todas as mensagens que a causalmente precedem tiverem sido entregues
- Para isso, assumimos que os relógios são ajustados no envio e entrega da mensagem

Multicast ordenado causalmente

- Podemos agora garantir que uma mensagem seja entregue somente se todas as mensagens que a causalmente precedem tiverem sido entregues
- Para isso, assumimos que os relógios são ajustados no envio e entrega da mensagem
 - Ao enviar m , P_i incrementa $VC_i[i]$

Multicast ordenado causalmente

- Podemos agora garantir que uma mensagem seja entregue somente se todas as mensagens que a causalmente precedem tiverem sido entregues
- Para isso, assumimos que os relógios são ajustados no envio e entrega da mensagem
 - Ao enviar m , P_i incrementa $VC_i[i]$
 - Ao entregar m à aplicação, faz
$$VC_i[k] \leftarrow \max(VC_i[k], ts(m)[k]), \forall k$$

Multicast ordenado causalmente

- Podemos agora garantir que uma mensagem seja entregue somente se todas as mensagens que a causalmente precedem tiverem sido entregues
- Para isso, assumimos que os relógios são ajustados no envio e entrega da mensagem
 - Ao enviar m , P_i incrementa $VC_i[i]$
 - Ao entregar m à aplicação, faz
$$VC_i[k] \leftarrow \max(VC_i[k], ts(m)[k]), \forall k$$
 - Não há ajuste no recebimento da mensagem, apenas quando da entrega à aplicação

Multicast ordenado causalmente

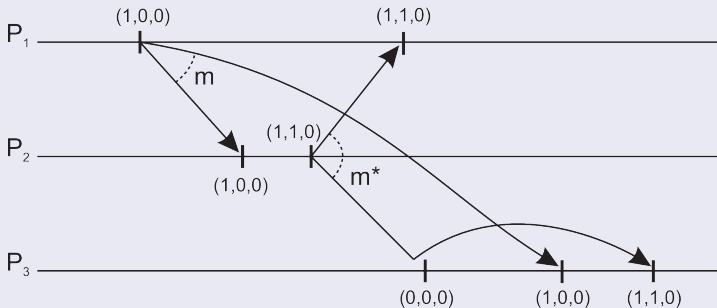
- Quando P_j recebe uma mensagem m de P_i , ele atrasa sua entrega à camada de aplicação até que:
 - $ts(m)[i] = VC_j[i] + 1$ (m é a próxima mensagem que P_j espera de P_i); e

Multicast ordenado causalmente

- Quando P_j recebe uma mensagem m de P_i , ele atrasa sua entrega à camada de aplicação até que:
 - $ts(m)[i] = VC_j[i] + 1$ (m é a próxima mensagem que P_j espera de P_i); e
 - $ts(m)[k] \leq VC_j[k], \forall k \neq i$ (P_j já entregou todas as mensagens enviadas por P_i no momento em que este enviou m)

Relógios Vetoriais

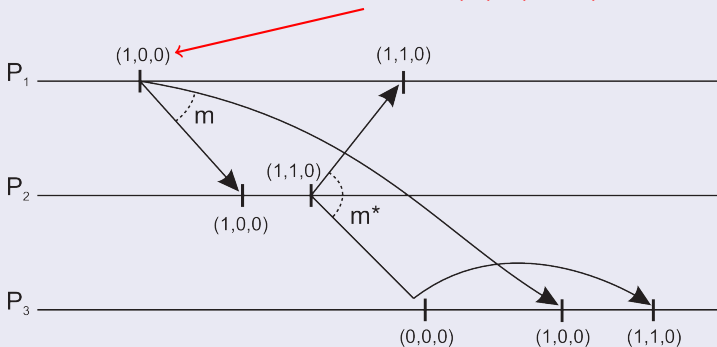
Forçando comunicação causal



Relógios Vetoriais

Forçando comunicação causal

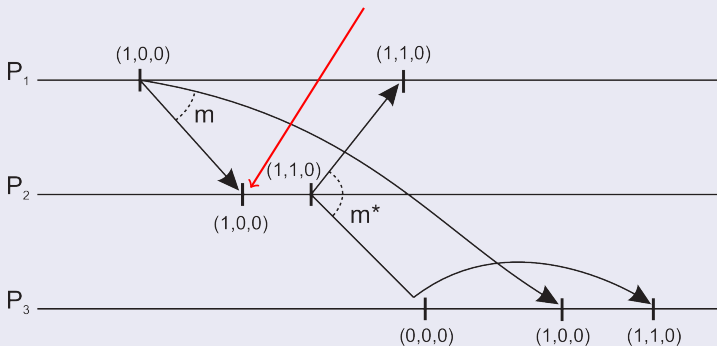
Em $(1, 0, 0)$ (horário local), P_1 envia m a dois outros processos, com $ts(m) = (1, 0, 0)$



Relógios Vetoriais

Forçando comunicação causal

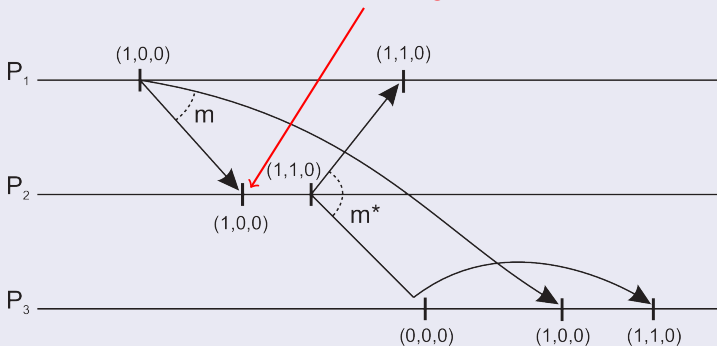
Após recebê-la, P_2 a entrega à aplicação,
atualizando seu relógio para $(1, 0, 0)$



Relógios Vetoriais

Forçando comunicação causal

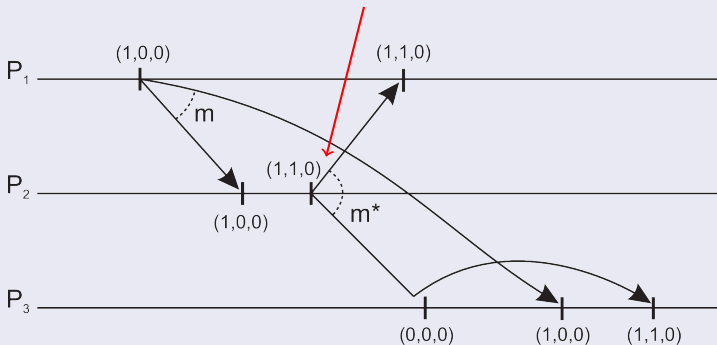
Indicando que P_2 recebeu 1 mensagem de P_1 e nenhuma de P_3 , não tendo enviado mensagens até o momento



Relógios Vetoriais

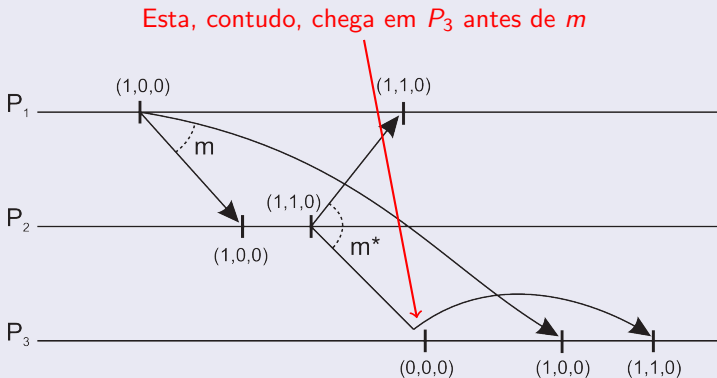
Forçando comunicação causal

P_2 decide enviar uma mensagem m^* aos demais processos, fazendo $VC_2[2] \leftarrow VC_2[2] + 1$ e $ts(m^*) \leftarrow (1, 1, 0)$



Relógios Vetoriais

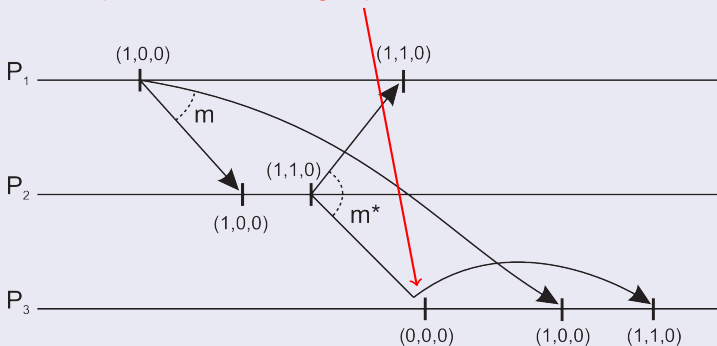
Forçando comunicação causal



Relógios Vetoriais

Forçando comunicação causal

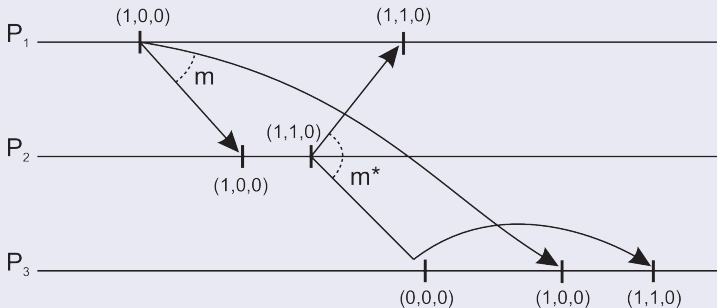
Ao comparar $ts(m^*) = (1, 1, 0)$ com seu tempo atual $(0, 0, 0) - P_3$ conclui que não recebeu uma mensagem de P_1 , aparentemente entregue por P_2 antes do envio de m^*



Relógios Vetoriais

Forçando comunicação causal

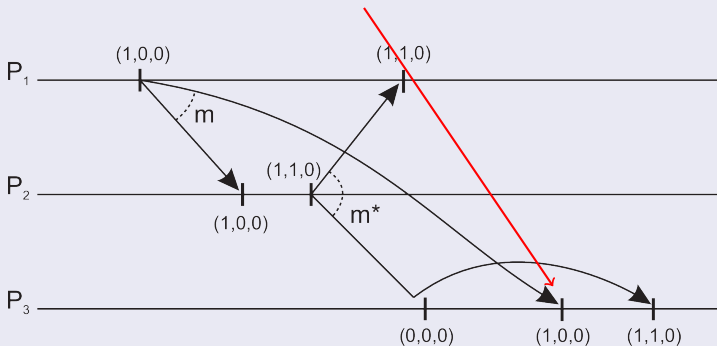
Ele decide então postergar a entrega de m^* , não ajustando seu relógio



Relógios Vetoriais

Forçando comunicação causal

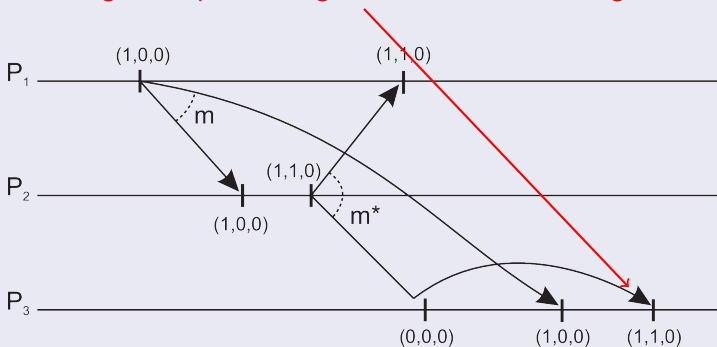
Após receber m e entregá-la, P_3 atualiza $VC_3 \leftarrow (1,0,0)$



Relógios Vetoriais

Forçando comunicação causal

Agora ele pode entregar m^* e atualizar seu relógio



Sincronização e Coordenação

- Sincronização do *Clock*
- Relógios Lógicos
- **Exclusão Mútua**

Exclusão Mútua

- Processos em SDs podem requerer acesso exclusivo a algum recurso
 - Problema da exclusão mútua

Exclusão Mútua

- Processos em SDs podem requerer acesso exclusivo a algum recurso
 - Problema da exclusão mútua
- Possíveis soluções
 - **Baseadas em permissão:** um processo que quiser entrar na região crítica (ou acessar um recurso) precisa da permissão dos outros processos

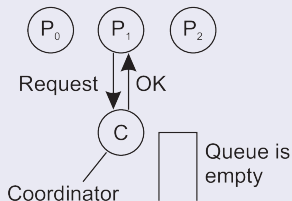
Exclusão Mútua

- Processos em SDs podem requerer acesso exclusivo a algum recurso
 - Problema da exclusão mútua
- Possíveis soluções
 - **Baseadas em permissão:** um processo que quiser entrar na região crítica (ou acessar um recurso) precisa da permissão dos outros processos
 - **Baseadas em tokens:** uma mensagem especial – *token* – é passada entre processos. Há um único *token* disponível, e o processo que o possuir pode entrar na região crítica ou passá-lo para frente quando não estiver interessado

Exclusão Mútua

Baseadas em permissão: Algoritmo centralizado

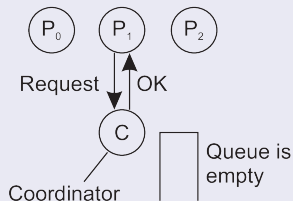
- Eleja um dos processos como coordenador



Exclusão Mútua

Baseadas em permissão: Algoritmo centralizado

- Eleja um dos processos como coordenador

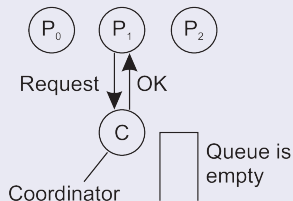


Quando um processo (P_1) quer acessar um recurso compartilhado, ele envia uma mensagem ao coordenador

Exclusão Mútua

Baseadas em permissão: Algoritmo centralizado

- Eleja um dos processos como coordenador

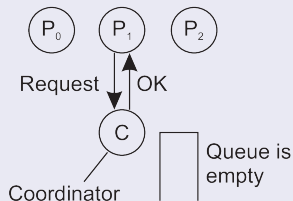


Se nenhum outro processo estiver acessando o recurso, o coordenador responde, garantindo a permissão

Exclusão Mútua

Baseadas em permissão: Algoritmo centralizado

- Eleja um dos processos como coordenador

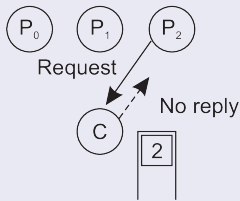
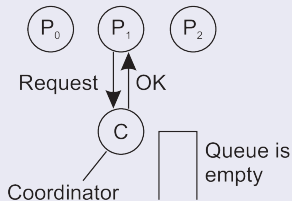


Quando a resposta chega, o processo pode seguir para a região crítica

Exclusão Mútua

Baseadas em permissão: Algoritmo centralizado

- Eleja um dos processos como coordenador

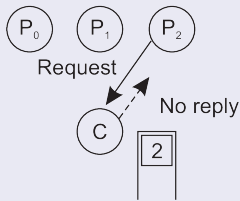
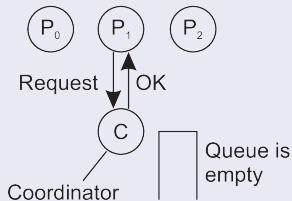


Se outro processo (P_2) pedir permissão, o coordenador não responde (bloqueando assim P_2 , que aguarda a resposta)

Exclusão Mútua

Baseadas em permissão: Algoritmo centralizado

- Eleja um dos processos como coordenador

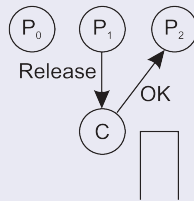
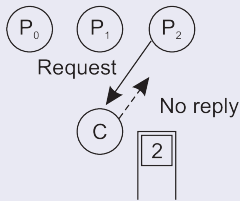
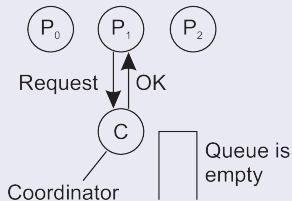


O coordenador enfileira então a requisição de P_2 e aguarda novas mensagens

Exclusão Mútua

Baseadas em permissão: Algoritmo centralizado

- Eleja um dos processos como coordenador

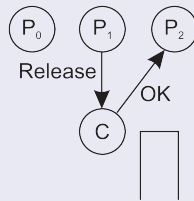
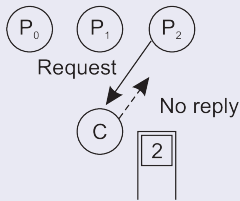
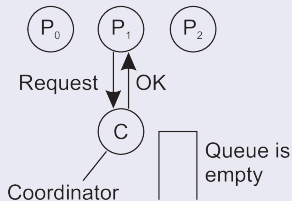


Quando P_1 termina com o recurso, envia uma mensagem ao coordenador

Exclusão Mútua

Baseadas em permissão: Algoritmo centralizado

- Eleja um dos processos como coordenador

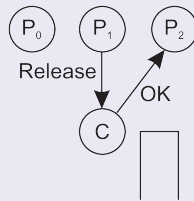
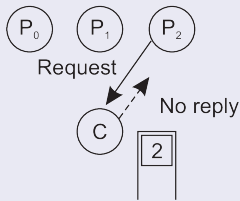
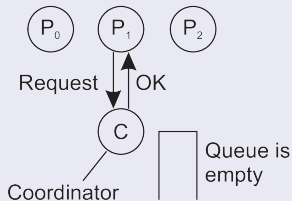


Este então verifica a fila e responde a P_2 , garantindo seu acesso

Exclusão Mútua

Baseadas em permissão: Algoritmo centralizado

- Eleja um dos processos como coordenador

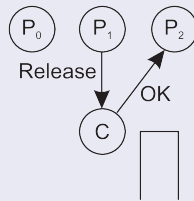
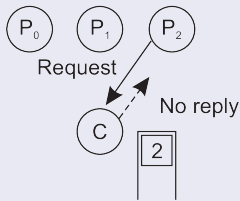
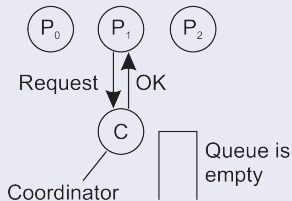


Problema: se o coordenador cair, todo o sistema pode travar (os processos ficarão bloqueados indefinidamente)

Exclusão Mútua

Baseadas em permissão: Algoritmo centralizado

- Eleja um dos processos como coordenador



Além disso, em sistemas grandes um único coordenador pode ser um gargalo de desempenho

Exclusão Mútua

Baseadas em permissão: Algoritmo distribuído

- Usa os relógios lógicos de Lamport

Exclusão Mútua

Baseadas em permissão: Algoritmo distribuído

- Usa os relógios lógicos de Lamport
- Quando um processo quer acessar um recurso compartilhado, ele
 - Constrói uma mensagem (requisição), com o nome do recurso, seu id de processo, e o tempo (lógico) atual
 - Envia a requisição a todos os processos (incluindo ele mesmo)

Exclusão Mútua

Baseadas em permissão: Algoritmo distribuído

- Quando um processo P_r recebe uma requisição m de outro processo P_e :

Exclusão Mútua

Baseadas em permissão: Algoritmo distribuído

- Quando um processo P_r recebe uma requisição m de outro processo P_e :
 - Se P_r não estiver acessando o recurso, e não quiser acessar, envia um OK a P_e

Exclusão Mútua

Baseadas em permissão: Algoritmo distribuído

- Quando um processo P_r recebe uma requisição m de outro processo P_e :
 - Se P_r não estiver acessando o recurso, e não quiser acessar, envia um OK a P_e
 - Se P_r já possuir acesso ao recurso, ele não responde, enfileirando a requisição

Exclusão Mútua

Baseadas em permissão: Algoritmo distribuído

- Quando um processo P_r recebe uma requisição m de outro processo P_e :
 - Se P_r não estiver acessando o recurso, e não quiser acessar, envia um OK a P_e
 - Se P_r já possuir acesso ao recurso, ele não responde, enfileirando a requisição
 - Se P_r quiser também acessar o recurso (e ainda não o fez)
 - Ele compara o *timestamp* de m com o da mensagem que enviou a todos
 - Se m tiver o menor *timestamp*, P_r responde com OK
 - Do contrário, ele enfileira a requisição, não respondendo a P_e

Exclusão Mútua

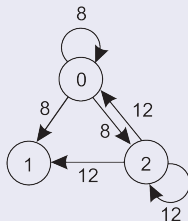
Baseadas em permissão: Algoritmo distribuído

- Após o envio das requisições, P_e espera até que todos tenham respondido
- Quando tiver terminado, remove todos os processos de sua fila, enviando OK eles

Exclusão Mútua

Baseadas em permissão: Algoritmo distribuído

- Após o envio das requisições, P_e espera até que todos tenham respondido
- Quando tiver terminado, remove todos os processos de sua fila, enviando OK eles

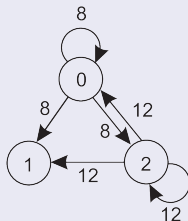


P_0 envia a todos uma requisição m_0 com $ts(m_0) = 8$

Exclusão Mútua

Baseadas em permissão: Algoritmo distribuído

- Após o envio das requisições, P_e espera até que todos tenham respondido
- Quando tiver terminado, remove todos os processos de sua fila, enviando OK eles

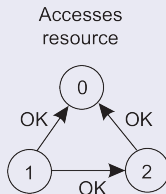
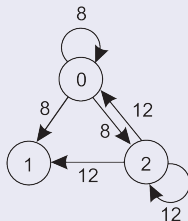


Ao mesmo tempo, P_2 envia m_2 , com $ts(m_2) = 12$

Exclusão Mútua

Baseadas em permissão: Algoritmo distribuído

- Após o envio das requisições, P_e espera até que todos tenham respondido
- Quando tiver terminado, remove todos os processos de sua fila, enviando OK eles

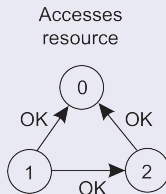
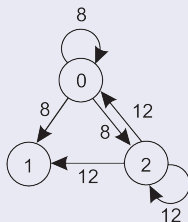


P_1 não está interessado no recurso, então envia OK a todos

Exclusão Mútua

Baseadas em permissão: Algoritmo distribuído

- Após o envio das requisições, P_e espera até que todos tenham respondido
- Quando tiver terminado, remove todos os processos de sua fila, enviando OK eles

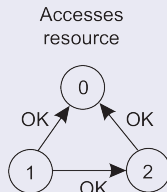
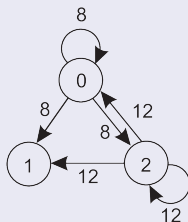


P_0 e P_2 vêem o conflito e comparam os *timestamps*

Exclusão Mútua

Baseadas em permissão: Algoritmo distribuído

- Após o envio das requisições, P_e espera até que todos tenham respondido
- Quando tiver terminado, remove todos os processos de sua fila, enviando OK eles

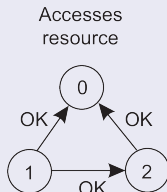
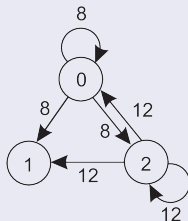


P_2 vê que perdeu, então envia OK a P_0

Exclusão Mútua

Baseadas em permissão: Algoritmo distribuído

- Após o envio das requisições, P_e espera até que todos tenham respondido
- Quando tiver terminado, remove todos os processos de sua fila, enviando OK eles

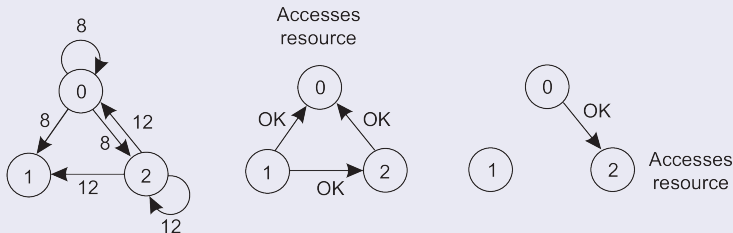


P_0 enfileira a requisição de P_2 e acessa o recurso

Exclusão Mútua

Baseadas em permissão: Algoritmo distribuído

- Após o envio das requisições, P_e espera até que todos tenham respondido
- Quando tiver terminado, remove todos os processos de sua fila, enviando OK eles

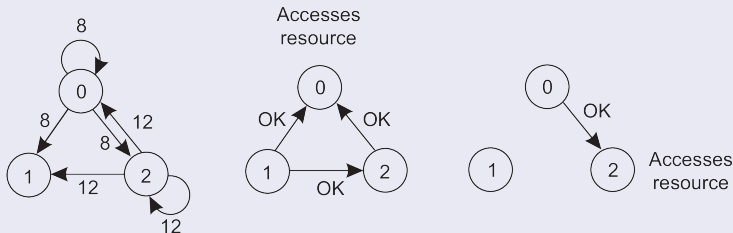


Ao terminar, P_0 desenfilera a requisição de P_2 , enviando OK a ele

Exclusão Mútua

Baseadas em permissão: Algoritmo distribuído

- Após o envio das requisições, P_e espera até que todos tenham respondido
- Quando tiver terminado, remove todos os processos de sua fila, enviando OK eles

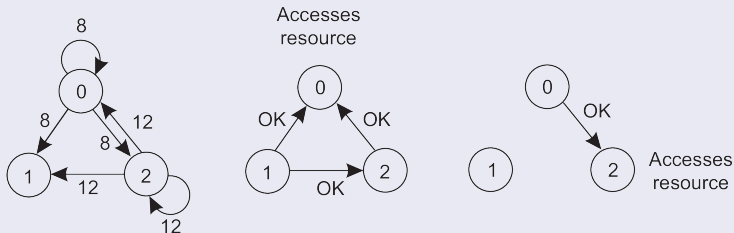


P_2 pode então usar o recurso

Exclusão Mútua

Baseadas em permissão: Algoritmo distribuído

- Após o envio das requisições, P_e espera até que todos tenham respondido
- Quando tiver terminado, remove todos os processos de sua fila, enviando OK eles

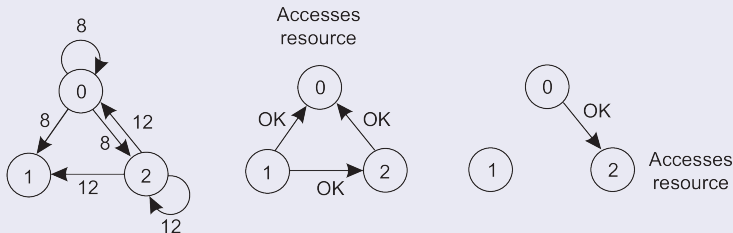


Problema: se um processo cair, não responderá a requisições

Exclusão Mútua

Baseadas em permissão: Algoritmo distribuído

- Após o envio das requisições, P_e espera até que todos tenham respondido
- Quando tiver terminado, remove todos os processos de sua fila, enviando OK eles

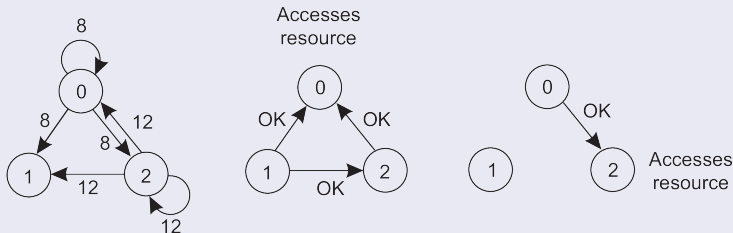


Isso será interpretado como negação, e ninguém acessa o recurso

Exclusão Mútua

Baseadas em permissão: Algoritmo distribuído

- Após o envio das requisições, P_e espera até que todos tenham respondido
- Quando tiver terminado, remove todos os processos de sua fila, enviando OK eles

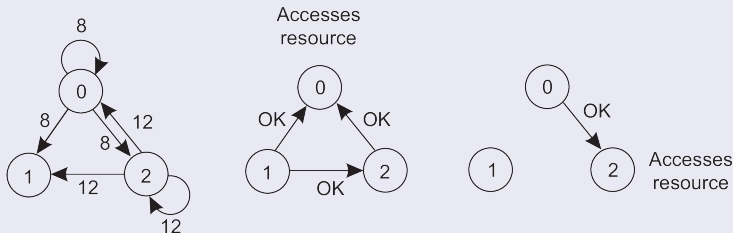


A solução é sempre responder, ou com OK ou negando

Exclusão Mútua

Baseadas em permissão: Algoritmo distribuído

- Após o envio das requisições, P_e espera até que todos tenham respondido
- Quando tiver terminado, remove todos os processos de sua fila, enviando OK eles



Após uma requisição ser negada, o emissor bloqueia até ter um OK

Exclusão Mútua

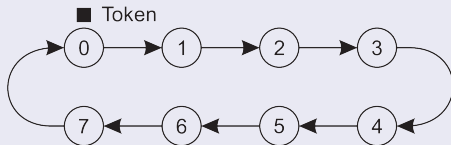
Baseadas em *token*: *Token ring algorithm*

- Organize os processos em um anel **lógico**, passando um *token* entre eles
- Aquele que estiver com o *token* pode entrar na região crítica (se quiser)

Exclusão Mútua

Baseadas em *token*: *Token ring algorithm*

- Organize os processos em um anel **lógico**, passando um *token* entre eles
- Aquele que estiver com o *token* pode entrar na região crítica (se quiser)



Quando o anel inicia, P_0 recebe um *token*, que é passado ao longo do anel

Exclusão Mútua

Algoritmo descentralizado

- Assuma que todo recurso é replicado N vezes
 - Cada réplica possui seu próprio coordenador

Exclusão Mútua

Algoritmo descentralizado

- Assuma que todo recurso é replicado N vezes
 - Cada réplica possui seu próprio coordenador
- Quando um processo quer acessar o recurso, precisará do voto da maioria dos coordenadores
 - De $m > N/2$ coordenadores
 - Se não conseguir m votos, o processo espera um tempo aleatório e tenta novamente

Exclusão Mútua

Algoritmo descentralizado

- Assuma que todo recurso é replicado N vezes
 - Cada réplica possui seu próprio coordenador
- Quando um processo quer acessar o recurso, precisará do voto da maioria dos coordenadores
 - De $m > N/2$ coordenadores
 - Se não conseguir m votos, o processo espera um tempo aleatório e tenta novamente
- Um coordenador sempre responde imediatamente a uma requisição, aceitando ou negando-a

Exclusão Mútua

Comparação

- Assumindo mensagens ponto-a-ponto
 - Multicast* para N processos conta como N mensagens

Algoritmo	# msgs por entrada/saída	Atraso para entrar (em qde msgs)	Problemas
Centralizado	3	2	Morte do coordenador
Distribuído	$2(N - 1)$	$2(N - 1)$	Morte de qualquer
Token ring	1 a ∞	0 a $N - 1$	Perder token, morrer
Decentralizado	$2mk + m, k = 1, 2, \dots$	$2mk$	<i>Starvation</i> , ineficiente

Exclusão Mútua

Comparação

- Assumindo mensagens ponto-a-ponto
 - Multicast* para N processos conta como N mensagens

Algoritmo	# msgs por entrada/saída	Atraso para entrar (em qde msgs)	Problemas
Centralizado	3	2	Morte do coordenador
Distribuído	$2(N - 1)$	$2(N - 1)$	Morte de qualquer
Token ring	1 a ∞	0 a $N - 1$	Perder token, morrer
Decentralizado	$2mk + m, k = 1, 2, \dots$	$2mk$	<i>Starvation</i> , ineficiente

Temos aqui o número de mensagens necessárias para um processo acessar e liberar um recurso e o atraso antes que um acesso possa ocorrer

Exclusão Mútua

Comparação

- Assumindo mensagens ponto-a-ponto
 - Multicast* para N processos conta como N mensagens

Algoritmo	# msgs por entrada/saída	Atraso para entrar (em qde msgs)	Problemas
Centralizado	3	2	Morte do coordenador
Distribuído	$2(N - 1)$	$2(N - 1)$	Morte de qualquer
Token ring	1 a ∞	0 a $N - 1$	Perder token, morrer
Decentralizado	$2mk + m, k = 1, 2, \dots$	$2mk$	<i>Starvation</i> , ineficiente

O algoritmo centralizado requer somente 3 mensagens para entrar e sair da região crítica: uma requisição, uma confirmação e uma liberação do recurso

Exclusão Mútua

Comparação

- Assumindo mensagens ponto-a-ponto
 - Multicast* para N processos conta como N mensagens

Algoritmo	# msgs por entrada/saída	Atraso para entrar (em qde msgs)	Problemas
Centralizado	3	2	Morte do coordenador
Distribuído	$2(N - 1)$	$2(N - 1)$	Morte de qualquer
Token ring	1 a ∞	0 a $N - 1$	Perder token, morrer
Decentralizado	$2mk + m, k = 1, 2, \dots$	$2mk$	<i>Starvation</i> , ineficiente

O distribuído requer $N - 1$ requisições, uma para cada um dos demais processos, além de $N - 1$ confirmações adicionais

Exclusão Mútua

Comparação

- Assumindo mensagens ponto-a-ponto
 - Multicast* para N processos conta como N mensagens

Algoritmo	# msgs por entrada/saída	Atraso para entrar (em qde msgs)	Problemas
Centralizado	3	2	Morte do coordenador
Distribuído	$2(N - 1)$	$2(N - 1)$	Morte de qualquer
Token ring	1 a ∞	0 a $N - 1$	Perder token, morrer
Decentralizado	$2mk + m, k = 1, 2, \dots$	$2mk$	<i>Starvation</i> , ineficiente

Com token ring, se cada processo quiser entrar na região crítica, então cada passada do token resulta em uma entrada e uma saída (média de uma mensagem)

Exclusão Mútua

Comparação

- Assumindo mensagens ponto-a-ponto
 - Multicast* para N processos conta como N mensagens

Algoritmo	# msgs por entrada/saída	Atraso para entrar (em qde msgs)	Problemas
Centralizado	3	2	Morte do coordenador
Distribuído	$2(N - 1)$	$2(N - 1)$	Morte de qualquer
Token ring	1 a ∞	0 a $N - 1$	Perder token, morrer
Decentralizado	$2mk + m, k = 1, 2, \dots$	$2mk$	<i>Starvation</i> , ineficiente

No outro extremo, se ninguém quiser entrar, o token pode circular por horas, e não há limite para o número de mensagens trocadas

Exclusão Mútua

Comparação

- Assumindo mensagens ponto-a-ponto
 - Multicast* para N processos conta como N mensagens

Algoritmo	# msgs por entrada/saída	Atraso para entrar (em qde msgs)	Problemas
Centralizado	3	2	Morte do coordenador
Distribuído	$2(N - 1)$	$2(N - 1)$	Morte de qualquer
Token ring	1 a ∞	0 a $N - 1$	Perder token, morrer
Descentralizado	$2mk + m, k = 1, 2, \dots$	$2mk$	<i>Starvation</i> , ineficiente

Por fim, no modelo descentralizado a mensagem deve ser enviada a m coordenadores, que a respondem

Exclusão Mútua

Comparação

- Assumindo mensagens ponto-a-ponto
 - Multicast* para N processos conta como N mensagens

Algoritmo	# msgs por entrada/saída	Atraso para entrar (em qde msgs)	Problemas
Centralizado	3	2	Morte do coordenador
Distribuído	$2(N - 1)$	$2(N - 1)$	Morte de qualquer
Token ring	1 a ∞	0 a $N - 1$	Perder token, morrer
Decentralizado	$2mk + m$, $k = 1, 2, \dots$	$2mk$	<i>Starvation</i> , ineficiente

É possível também que k tentativas precisem ser feitas. Deixar a RC requer o envio de mensagem adicional a cada um dos m coordenadores

Exclusão Mútua

Comparação

- Assumindo mensagens ponto-a-ponto
 - Multicast* para N processos conta como N mensagens

Algoritmo	# msgs por entrada/saída	Atraso para entrar (em qde msgs)	Problemas
Centralizado	3	2	Morte do coordenador
Distribuído	$2(N - 1)$	$2(N - 1)$	Morte de qualquer
Token ring	1 a ∞	0 a $N - 1$	Perder token, morrer
Decentralizado	$2mk + m, k = 1, 2, \dots$	$2mk$	<i>Starvation</i> , ineficiente

Virtualmente todos esses algoritmos sofrem muito em caso de *crashes*