Computação Distribuída

Relógios Físicos, Lógicos e Vetoriais

Vladimir Rocha (Vladi)

CMCC - Universidade Federal do ABC

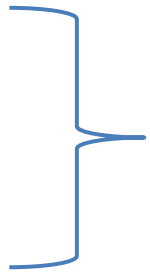
Disclaimer

- Estes slides foram baseados nos do professor **Emilio Francesquini** para o curso de Sistemas Distribuídos na UFABC.
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- Estes slides foram adaptados daqueles originalmente preparados (e gentilmente cedidos) pelo professor **Daniel Cordeiro, da EACH-USP** que por sua vez foram baseados naqueles disponibilizados online pelos autores do livro "Distributed Systems", 3ª Edição em: https://www.distributed-systems.net.

Agenda

Sincronização de relógios

- Relógios físicos
- Relógios lógicos (de Lamport)
  - → Multicast de Ordem Total
- Relógios vetoriais
  - → Multicast de Ordem Causal



Capítulo 4

Agenda

Sincronização de relógios

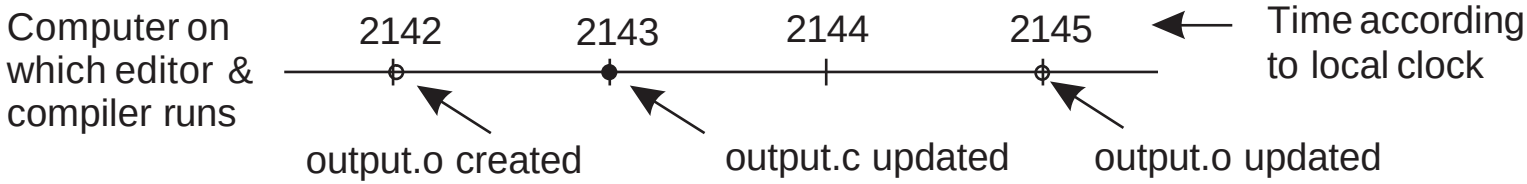
- Relógios físicos
- Relógios lógicos (de lamport)
- Relógios vetoriais

Relógios Físicos: centralizado

- Em sistemas centralizados a definição do horário não é ambígua

Exemplo de problema: Make sabe da alteração do .c

Assumindo e.g. output.c criado no tempo 2140

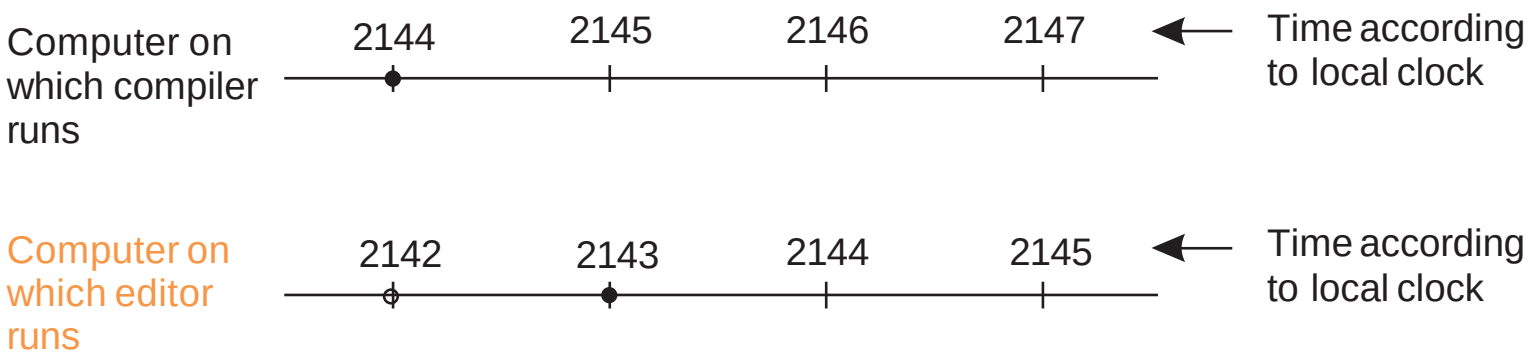


Relógios Físicos: distribuído

- Em sistemas distribuídos isto não é verdade (é ambígua)

  Mesmo em sistemas multi-processador pode não ser verdade

Exemplo de problema: Make não sabe da alteração do .c

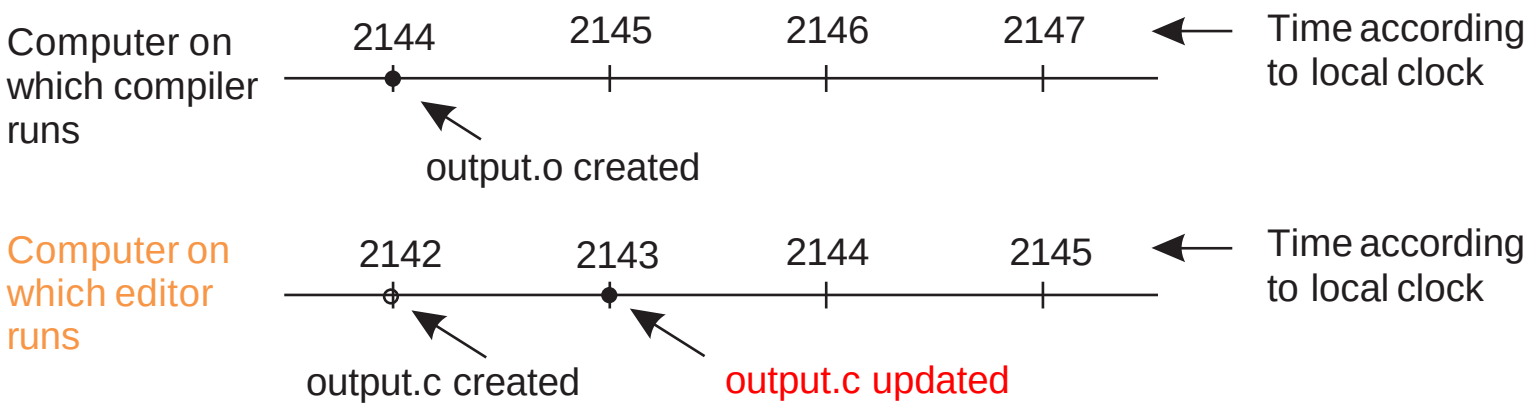


Relógios Físicos: distribuído

- Em sistemas distribuídos isto não é verdade (é ambígua)

  Mesmo em sistemas multi-processador pode não ser verdade

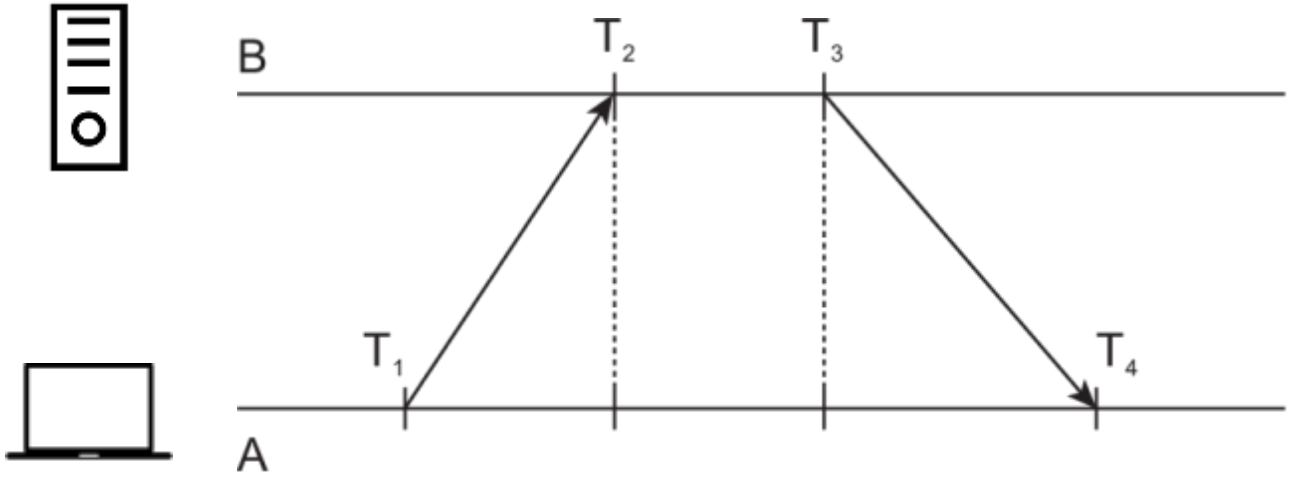
Exemplo de problema: Make não sabe da alteração do .c



Computador onde é executada a compilação não criará o output.o da atualização realizada no tempo 2143 !

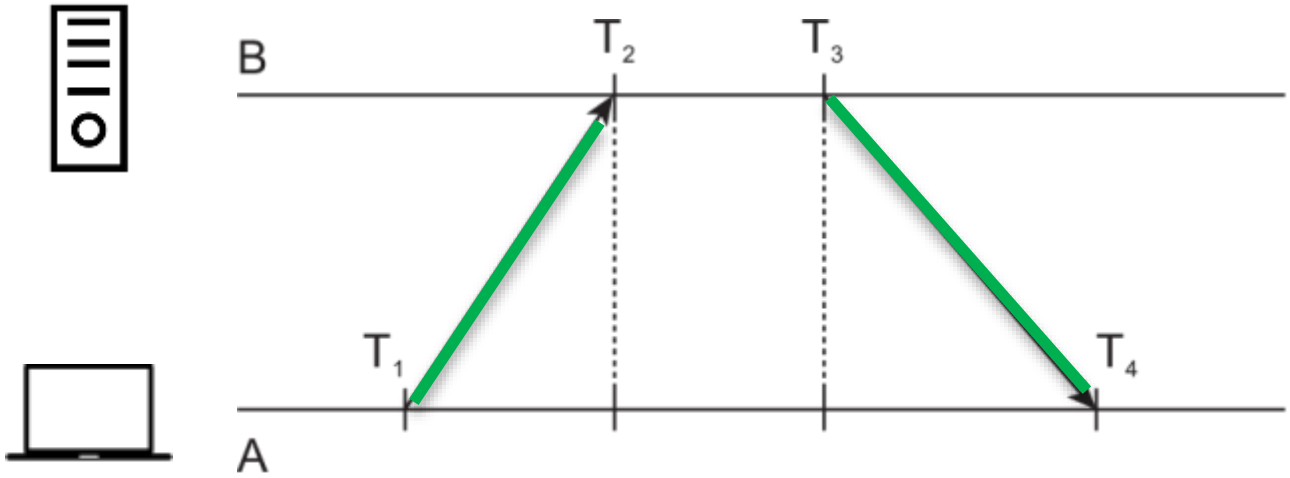
Relógios Físicos: ajuste de horários

Recuperação do horário atual de um servidor



Relógios Físicos: ajuste de horários

Recuperação do horário atual de um servidor

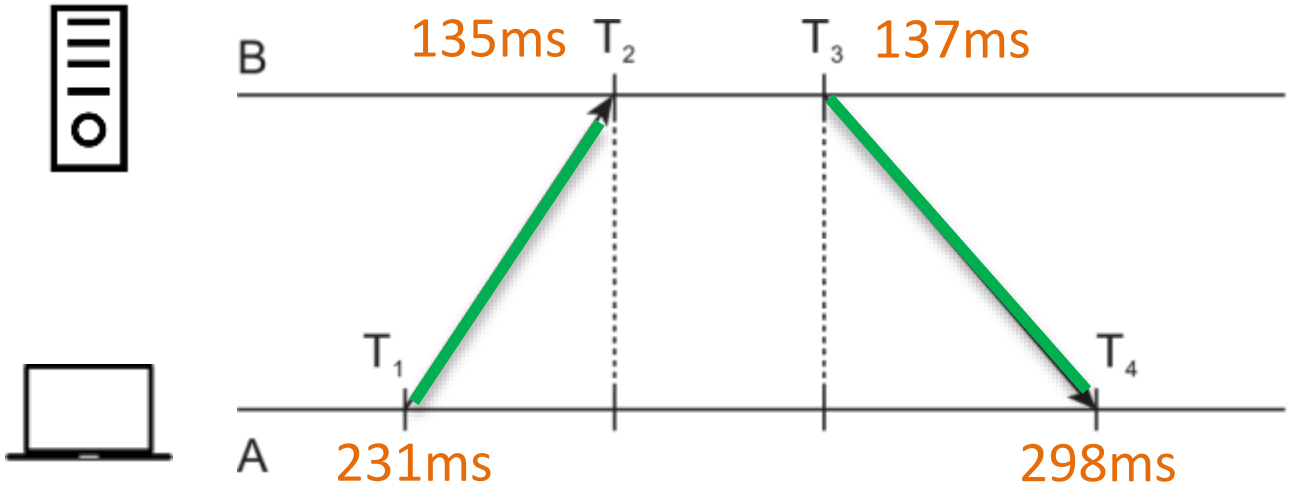


Cálculo do atraso na propagação δ (*round trip time - RTT*)

$$\delta = (T4 - T1) - (T3 - T2)$$

Relógios Físicos: ajuste de horários

Recuperação do horário atual de um servidor



Cálculo do atraso na propagação δ (*round trip time - RTT*)

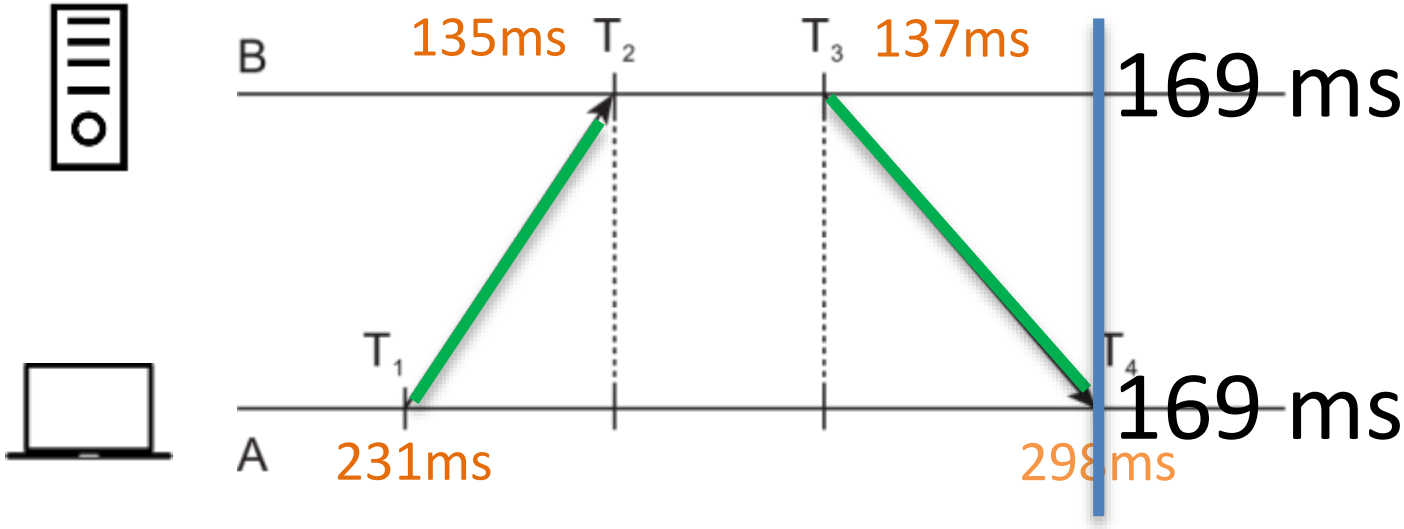
$$\delta = (T4 - T1) - (T3 - T2)$$

$$\delta = (298 - 231) - (137 - 135)$$

$$\delta = 65 \text{ ms}$$

Relógios Físicos: ajuste de horários

Recuperação do horário atual de um servidor



Cálculo do atraso na propagação δ (*round trip time - RTT*)

$$\delta = (T4 - T1) - (T3 - T2)$$

$$\delta = (298 - 231) - (137 - 135)$$

$$\delta = 65 \text{ ms}$$

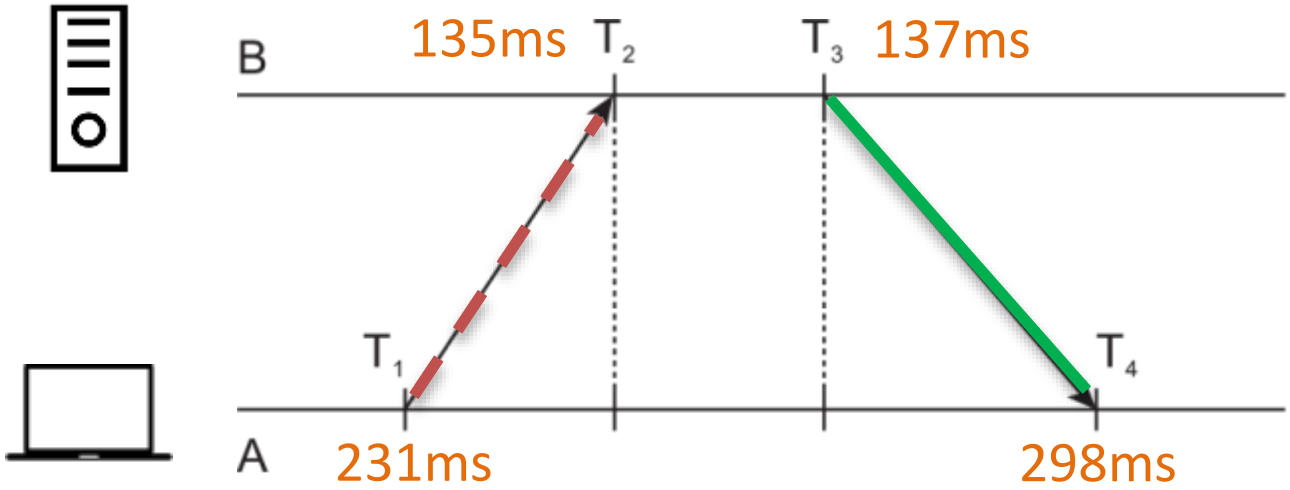
Algoritmo de Cristian [1989]

$T_A = T_3 + \delta/2$

$T_A = 137 + 32 = 169$ ms

Relógios Físicos: ajuste de horários

Recuperação do horário atual de um servidor



Cálculo da diferença relativa θ entre servidores (*offset*)

$$\theta = [(T2 - T1) + (T3 - T4)] / 2$$

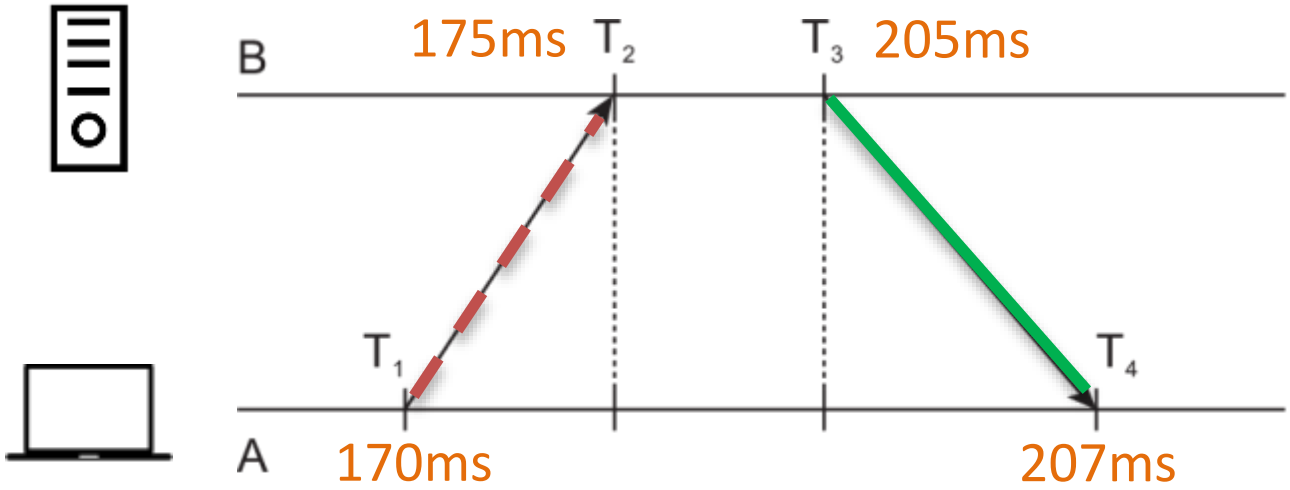
$$\theta = [(135 - 231) + (137 - 298)] / 2$$

$$\theta = -128.5 \text{ ms}$$

Um nº negativo indica que nosso relógio está mais rápido e um positivo indica que está mais lento

Relógios Físicos: ajuste de horários

Recuperação do horário atual de um servidor



Cálculo da diferença relativa θ entre servidores (*offset*)

$$\theta = [(T2 - T1) + (T3 - T4)] / 2$$

$$\theta = ?$$

Relógios Físicos: Sincronização externa (NTP)

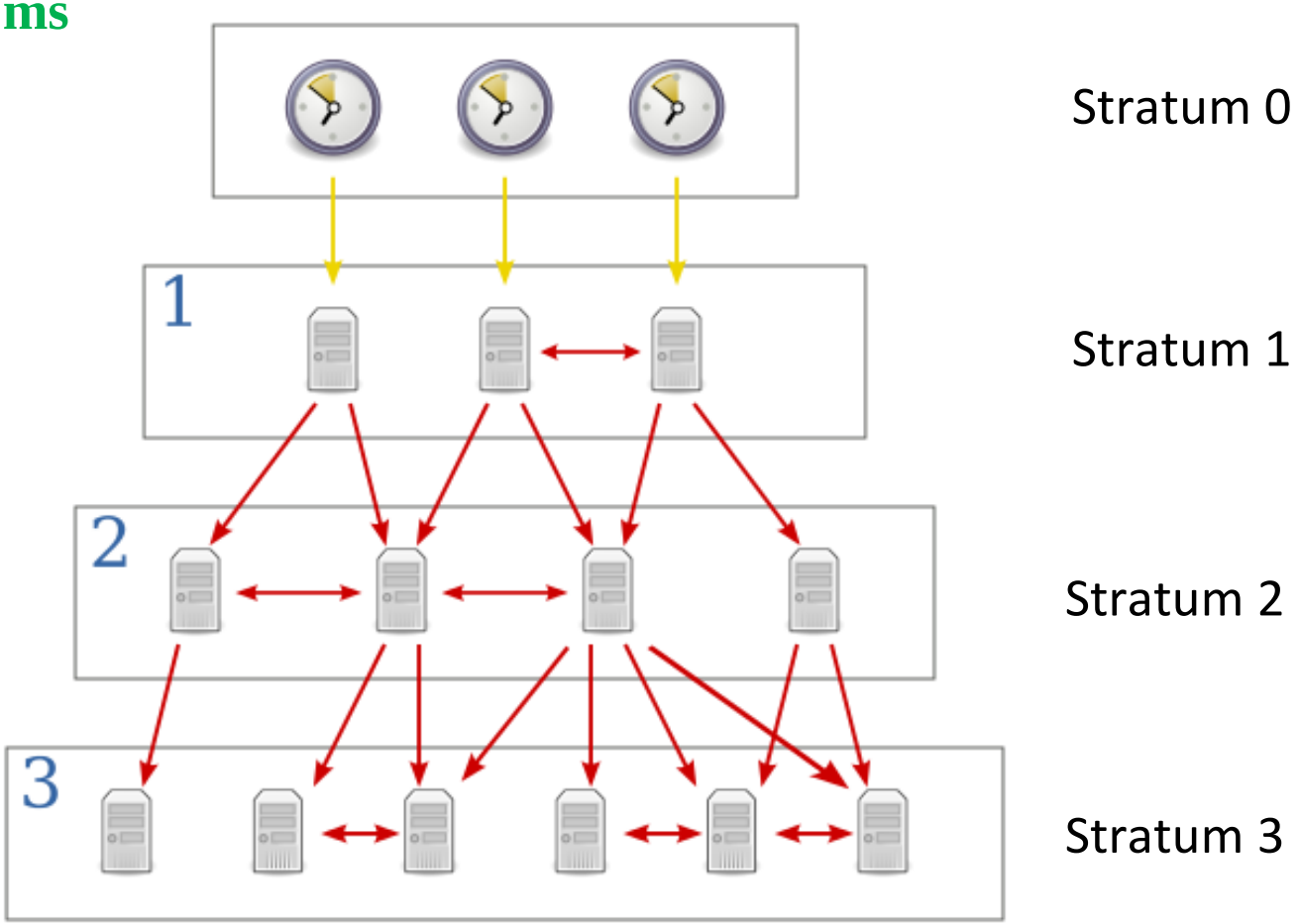
Network Time Protocol [1975 - 1985 - 1992 - 2010]

Colete oito (θ,δ) e escolha os *offset* θ cujos atrasos RTT δ são minimais.

Sincronia de **1-50 ms**

direta →

rede →



https://en.wikipedia.org/wiki/Network_Time_Protocol

Relógios Físicos: Sincronização externa (NTP)

Big Techs pedem fim de segundo extra que ajusta relógios à rotação da Terra

Google, Amazon, Microsoft, Meta se juntaram para tentar acabar com o segundo bissexto, quando se acrescenta um segundo extra aos relógios do mundo todo para sincronizá-los com a rotação da Terra.

https://www.uol.com.br/tilt/noticias/redacao/2022/07/28/big-techs-x-tempo-a-briga-das-gigantes-contra-os-segundos-bissextos.htm

Relógios Físicos: Sincronização interna (Berkeley)

Algoritmo de Berkeley [Gusella 1989]

Permita o servidor de hora sonde todas as máquinas periodicamente, calcule uma média e informe cada máquina como ela deve ajustar o seu horário relativo ao seu horário atual.

Nota:

Você terá todas as máquinas em sincronia entre **20 e 25 ms**. Você nem precisa propagar o horário UTC (Universal Coordinated Time, horário real medido em 50 relógios atômicos no mundo).

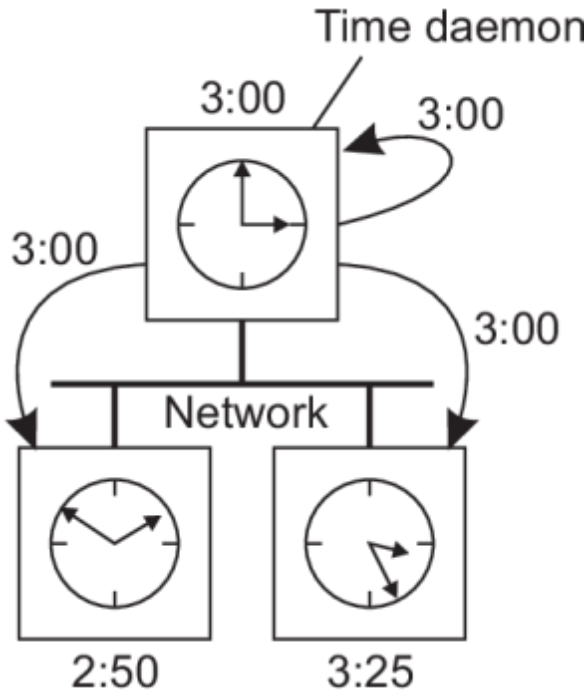
É fundamental

saber que atrasar o relógio nunca é permitido. Você deve fazer ajustes suaves.

Relógios Físicos: Sincronização interna (Berkeley)

Algoritmo de Berkeley

O daemon pergunta a diferença do horário respeito dele

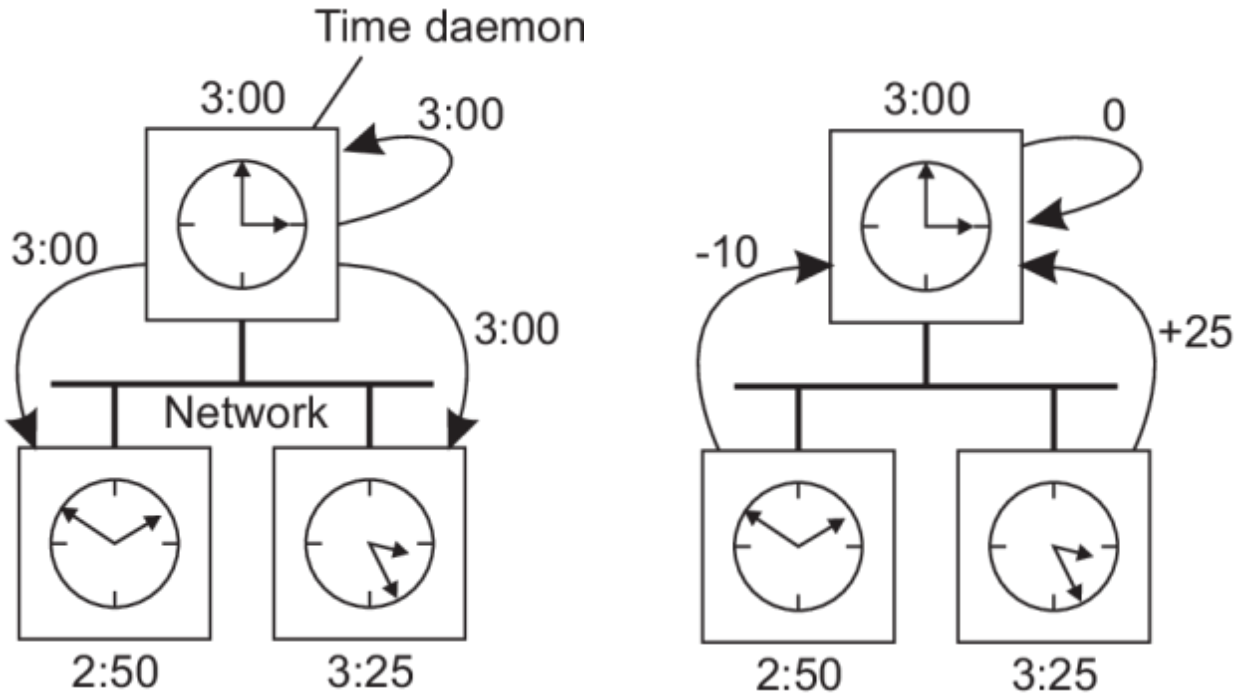


Relógios Físicos: Sincronização interna (Berkeley)

Algoritmo de Berkeley

O daemon pergunta a diferença do horário respeito dele

Os nós respondem a diferença



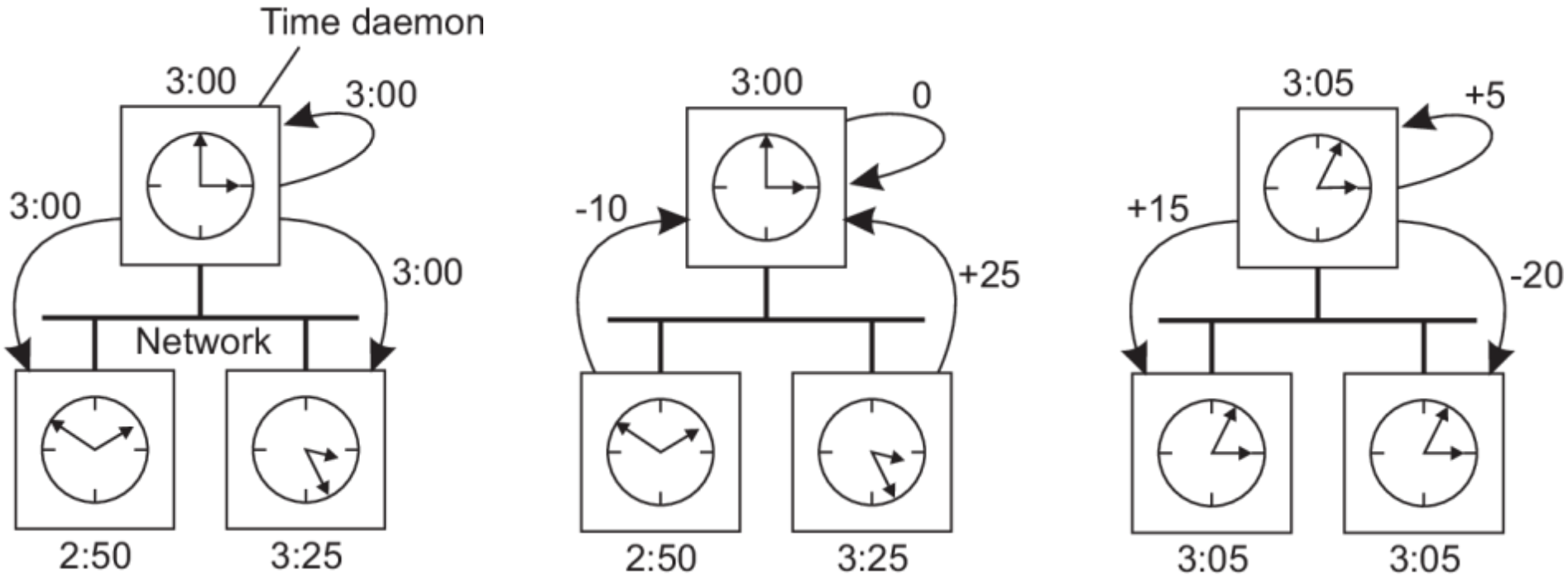
Relógios Físicos: Sincronização interna (Berkeley)

Algoritmo de Berkeley

O daemon pergunta a diferença do horário respeito dele

Os nós respondem a diferença

O daemon calcula e informa individualmente como devem ser ajustados

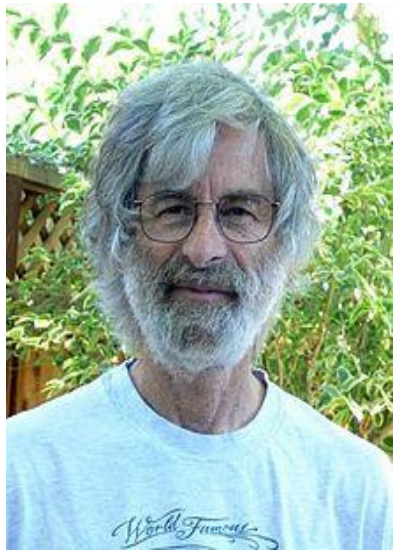


Agenda

Sincronização de relógios

- Relógios físicos
- Relógios lógicos (de Lamport)
- Relógios vetoriais

Relógio Lógico: Problema



Leslie Lamport
(pai dos Sist. Distrib.)

O que importa na maior parte dos sistemas distribuídos não é fazer com que todos os processos concordem com a hora exata (e.g., como NTP)

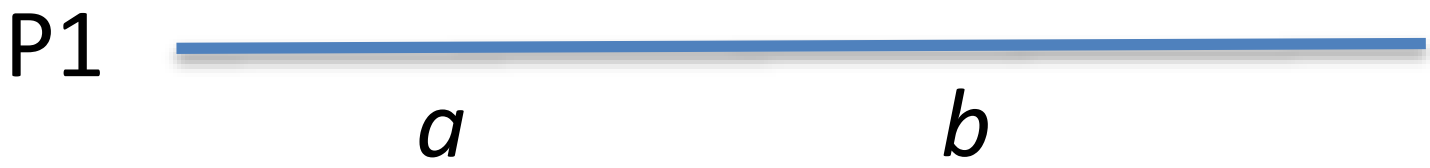
mas sim fazer com que eles concordem com a ordem em que os eventos ocorreram.

Ou seja, precisamos de uma noção de **ordem** entre os eventos.

Relógio Lógico: A relação "aconteceu-antes"

A relação "aconteceu-antes" (*happened-before*) [Lamport 1978]

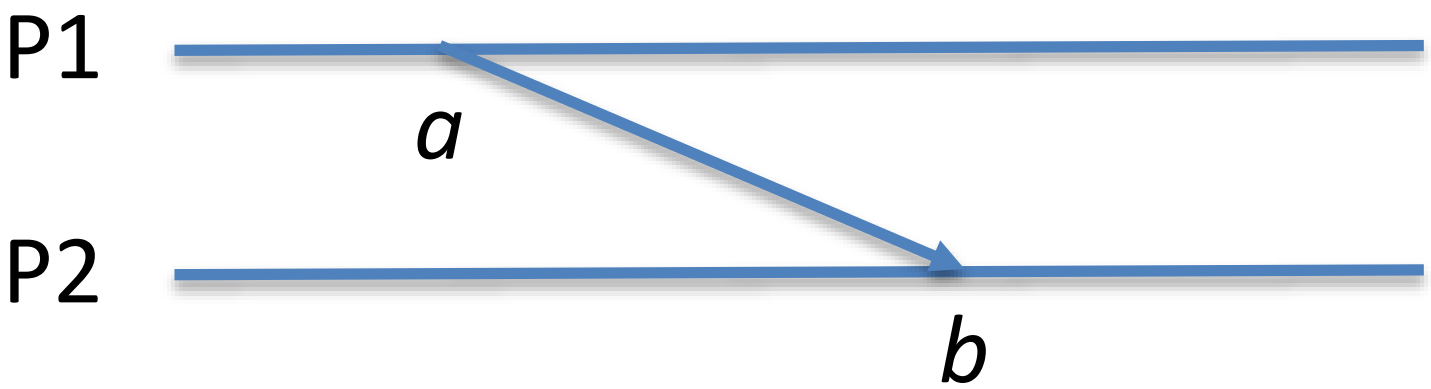
- se a e b são dois eventos de um mesmo processo e a ocorreu antes de b, então $a \rightarrow b$



Relógio Lógico: A relação "aconteceu-antes"

A relação "aconteceu-antes" (*happened-before*) [Lamport 1978]

- se a e b são dois eventos de um mesmo processo e a ocorreu 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 $a \rightarrow b$



Relógio Lógico: A relação “aconteceu-antes”

A relação “aconteceu-antes” (*happened-before*) [Lamport 1978]

- se a e b são dois eventos de um mesmo processo e a ocorreu 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 $a \rightarrow c$

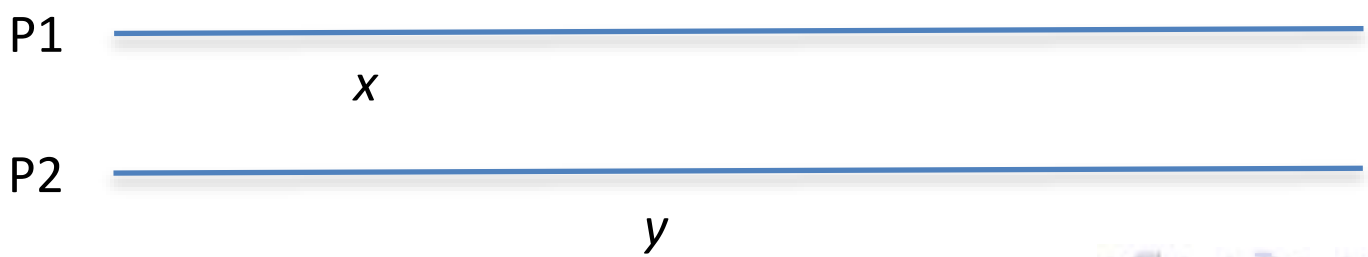
Relógio Lógico: A relação "aconteceu-antes"

A relação "aconteceu-antes" (*happened-before*) [Lamport 1978]

- se a e b são dois eventos de um mesmo processo e a ocorreu 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 $a \rightarrow c$

Eventos concorrentes

Se dois eventos x e y ocorrem em processos distintos e esses processos nunca interagem (mesmo que indiretamente) então nem $x \rightarrow y$ nem $y \rightarrow x$ são verdade. São eventos concorrentes. Quando se diz que dois eventos são concorrentes na verdade quer dizer que *nada pode (ou precisa) ser dito sobre a sua ordem*.



Relógio Lógico: A relação "aconteceu-antes"

A relação "aconteceu-antes" (*happened-before*) [Lamport 1978]

- se a e b são dois eventos de um mesmo processo e a ocorreu 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 $a \rightarrow c$

Eventos concorrentes

Se dois eventos x e y ocorrem em processos distintos e esses processos nunca interagem (mesmo que indiretamente) então nem $x \rightarrow y$ nem $y \rightarrow x$ são verdade. São eventos concorrentes. Quando se diz que dois eventos são concorrentes na verdade quer dizer que *nada pode (ou precisa) ser dito sobre a sua ordem*.

Nota:

Isso introduz uma noção de ordem PARCIAL dos eventos em um sistema com processos executando concorrentemente.

Relógio lógico de Lamport

Problema

Como fazemos para manter uma visão global do comportamento do sistema que seja consistente com a relação aconteceu-antes?

Relógio lógico de Lamport

Problema

Como fazemos para manter uma visão global do comportamento do sistema que seja consistente com a relação aconteceu-antes?

Solução

Associar um *timestamp* $C(e)$ a cada evento e tal que:

P1 se a e b são dois eventos no mesmo processo e $a \rightarrow b$, então é obrigatório que $C(a) < C(b)$

P2 se a corresponde ao envio de uma mensagem m e b ao recebimento desta mensagem, então também é válido que $C(a) < C(b)$

Relógio lógico de Lamport

Problema

Como fazemos para manter uma visão global do comportamento do sistema que seja consistente com a relação aconteceu-antes?

Solução

Associar um *timestamp* $C(e)$ a cada evento e tal que:

P1 se a e b são dois eventos no mesmo processo e $a \rightarrow b$, então é obrigatório que $C(a) < C(b)$

P2 se a corresponde ao envio de uma mensagem m e b ao recebimento desta mensagem, então também é válido que $C(a) < C(b)$

Como associar um *timestamp* a um evento quando não há um relógio global? Solução: manter um conjunto de relógios lógicos consistentes, um para cada processo

Relógio lógico de Lamport

Solução

Cada processo P_i mantém um **contador** C_i local e o ajusta de acordo com as seguintes regras:

Relógio lógico de Lamport

Solução

Cada processo P_i mantém um **contador** C_i local e o ajusta de acordo com as seguintes regras:

1. para quaisquer dois **eventos sucessivos** que ocorrer em P_i, C_i é incrementado em 1

Relógio lógico de Lamport

Solução

Cada processo P_i mantém um **contador** C_i local e o ajusta de acordo com as seguintes regras:

1. para quaisquer dois **eventos sucessivos** que ocorrer em P_i, C_i é incrementado em 1
2. toda vez que uma mensagem m for enviada por um processo P_i, a mensagem deve receber um *timestamp* $ts(m) = C_i$

Relógio lógico de Lamport

Solução

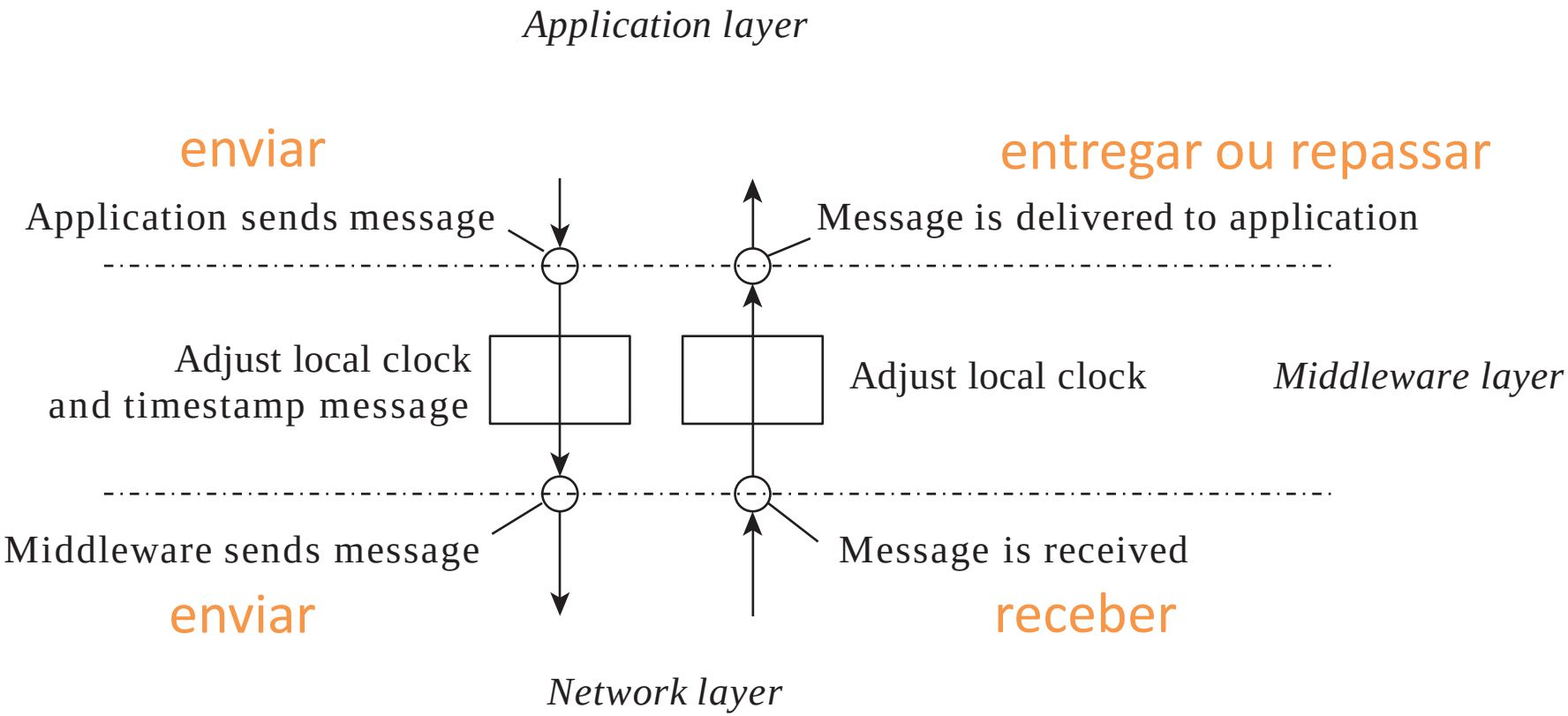
Cada processo P_i mantém um **contador** C_i local e o ajusta de acordo com as seguintes regras:

1. para quaisquer dois **eventos sucessivos** que ocorrer em P_i , C_i é incrementado em 1
2. toda vez que uma mensagem m for enviada por um processo P_i , a mensagem deve receber um *timestamp* $ts(m) = C_i$
3. sempre que uma mensagem m for recebida por um processo P_j , P_j **ajustará** seu contador local C_j para $\max\{C_j , ts(m)\}$ e executará o passo 1 antes de **repassar** m para a aplicação

Relógio lógico de Lamport: nota

Nota

Os ajustes ocorrem na camada do *middleware*



Relógio lógico de Lamport

Solução

Cada processo P_i mantém um **contador** C_i local e o ajusta de acordo com as seguintes regras:

1. para quaisquer dois **eventos sucessivos** que ocorrer em P_i , C_i é incrementado em 1
2. toda vez que uma mensagem m for enviada por um processo P_i , a mensagem deve receber um *timestamp* $ts(m) = C_i$
3. sempre que uma mensagem m for recebida por um processo P_j , P_j ajustará seu contador local C_j para $\max\{C_j , ts(m)\}$ e executará o passo 1 antes de repassar m para a aplicação

Observações:

- a propriedade **P1** é satisfeita por (1); propriedade **P2** por (2) e (3)
- ainda assim pode acontecer de dois eventos ocorrerem ao mesmo tempo (concorrente). Desempate usando os IDs dos processos criam uma ordem TOTAL dos eventos. Sem isso, é ordem parcial.

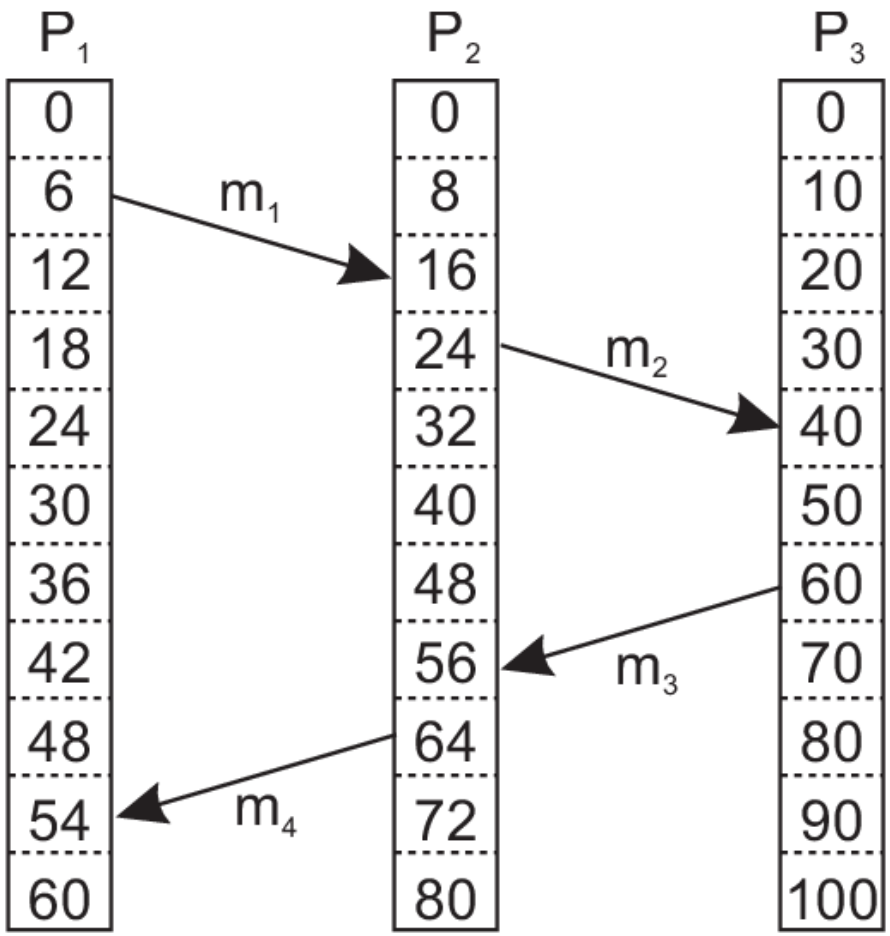
Relógio lógico de Lamport – exemplo

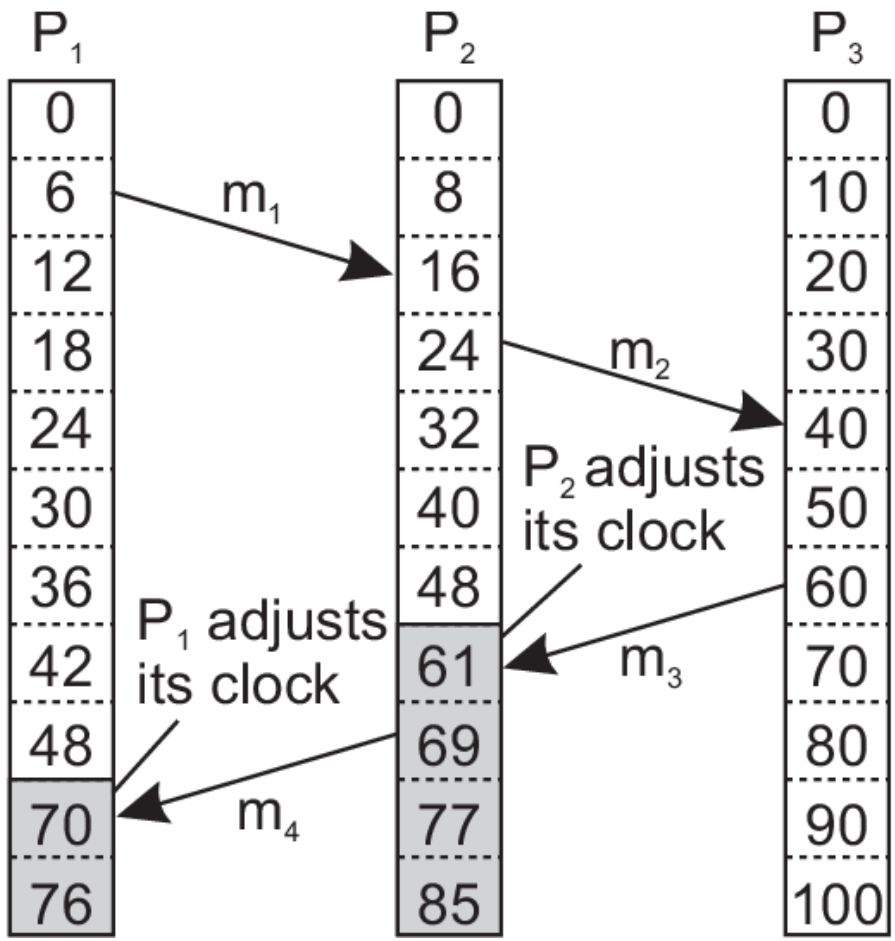
Considere três processos com contadores de eventos funcionando a velocidades diferentes.

P_1	P_2	P_3
0	0	0
6	8	10
12	16	20
18	24	30
24	32	40
30	40	50
36	48	60
42	56	70
48	64	80
54	72	90
60	80	100

Relógio lógico de Lamport – exemplo

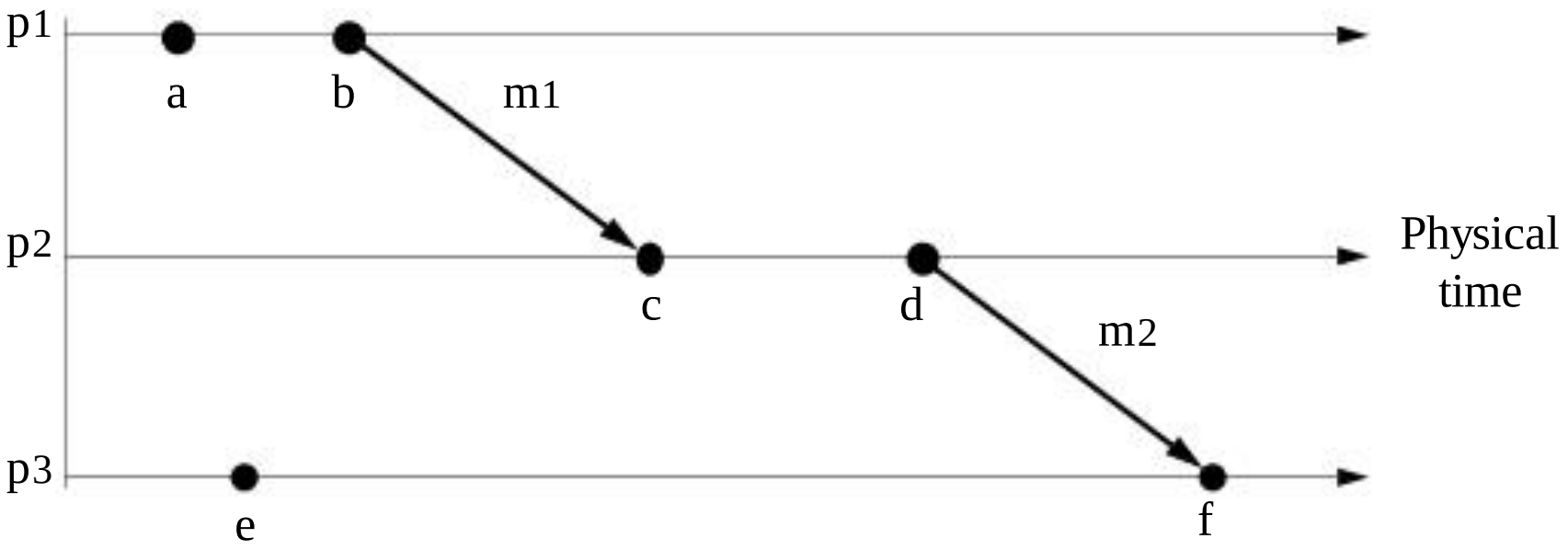
Considere três processos com contadores de eventos funcionando a velocidades diferentes.





Certo ou errado segundo Lamport?

Relógio lógico: Exercício



Fonte: CDKB

Exercício: No contexto do happens-before que se pode dizer sobre:

1. **a** e **b**? 2. **b** e **c**? 3. **a** e **f**?
4. **a** e **e**? 5. **c** e **e**?

Exercício: Insira os relógios lógicos para cada evento (relógios começam em 0)

Relógio lógico: multicast

Relógio lógico: multicast

- Relógios lógicos (de Lamport)

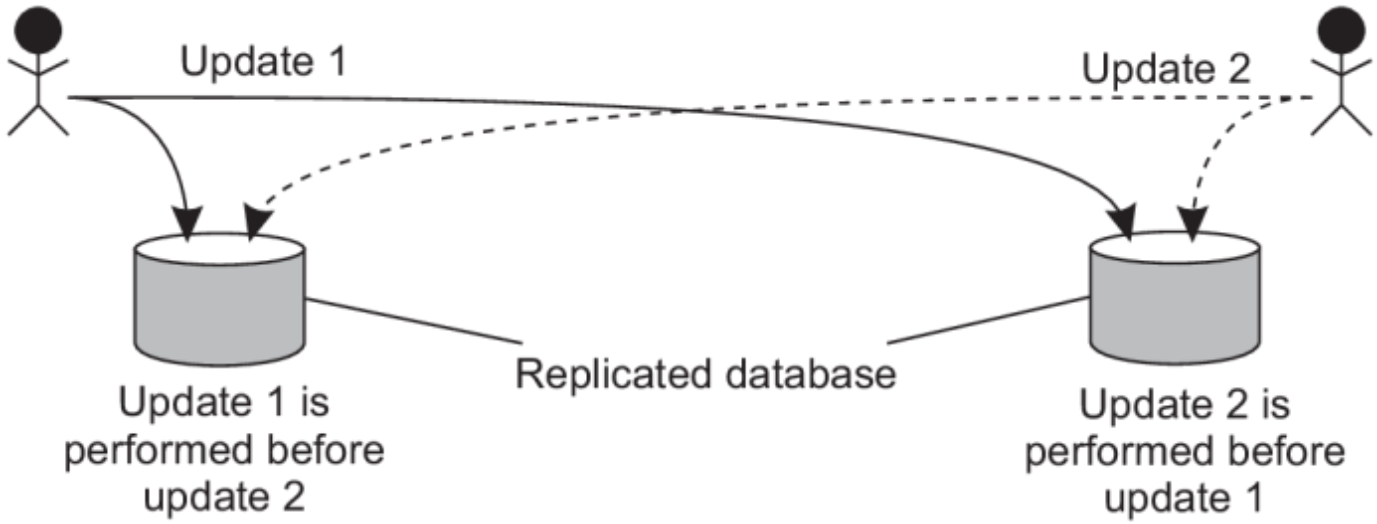
  → Multicast de Ordem Total

Relógio lógico: multicast com ordem total

Problema

Algumas vezes precisamos garantir que atualizações concorrentes em um banco de dados replicado sejam vistas por todos como se tivessem ocorrido na mesma ordem.

- P_1 adiciona R$ 100 a uma conta (valor inicial: R$ 1000)
- P_2 incrementa a conta em 1%



Resultado

Na ausência de sincronização correta,

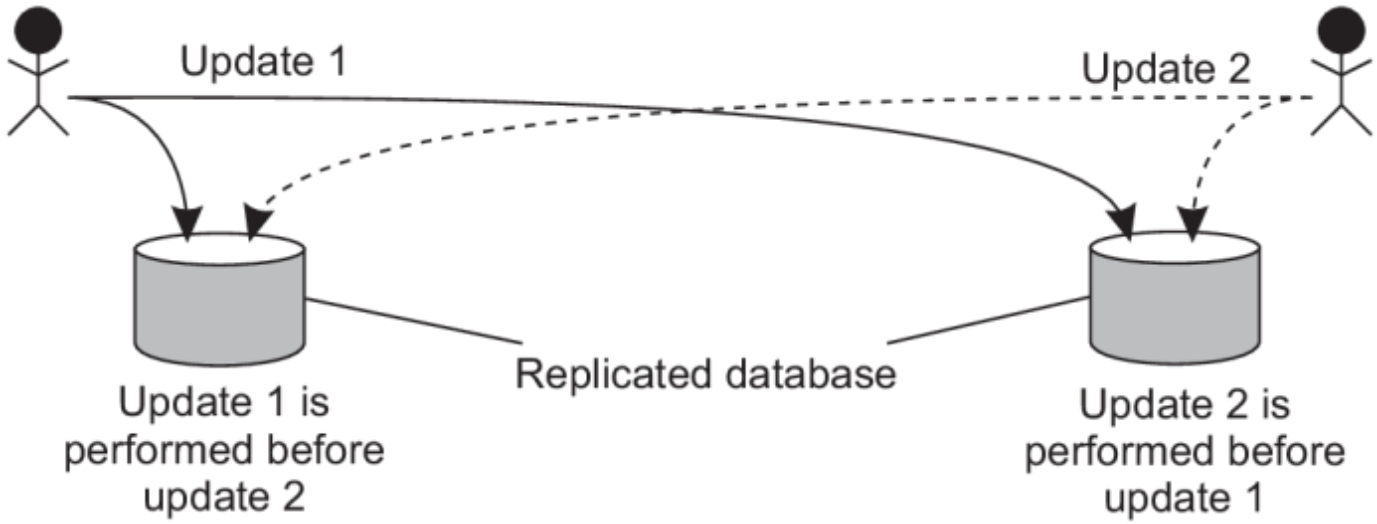
réplica #1 ← R$? enquanto que na réplica #2 ← R$?

Relógio lógico: multicast com ordem total

Problema

Algumas vezes precisamos garantir que atualizações concorrentes em um banco de dados replicado sejam vistas por todos como se tivessem ocorrido na mesma ordem.

- P_1 adiciona R$ 100 a uma conta (valor inicial: R$ 1000)
- P_2 incrementa a conta em 1%



Resultado

Na ausência de sincronização correta,

réplica #1 ← R$ 1111, enquanto que na réplica #2 ← R$ 1110.

Relógio lógico: multicast com ordem total

Solução

- P_i envia uma mensagem com timestamp m_i para todos os outros. A mensagem é colocada em sua fila local $queue_i$.
- Toda mensagem que chegar em P_j é colocada na fila $queue_j$ priorizada pelo seu timestamp e **confirmada (*acknowledged ACK*)** por todos os outros processos

P_j repassa a mensagem m_i para a sua aplicação somente se:

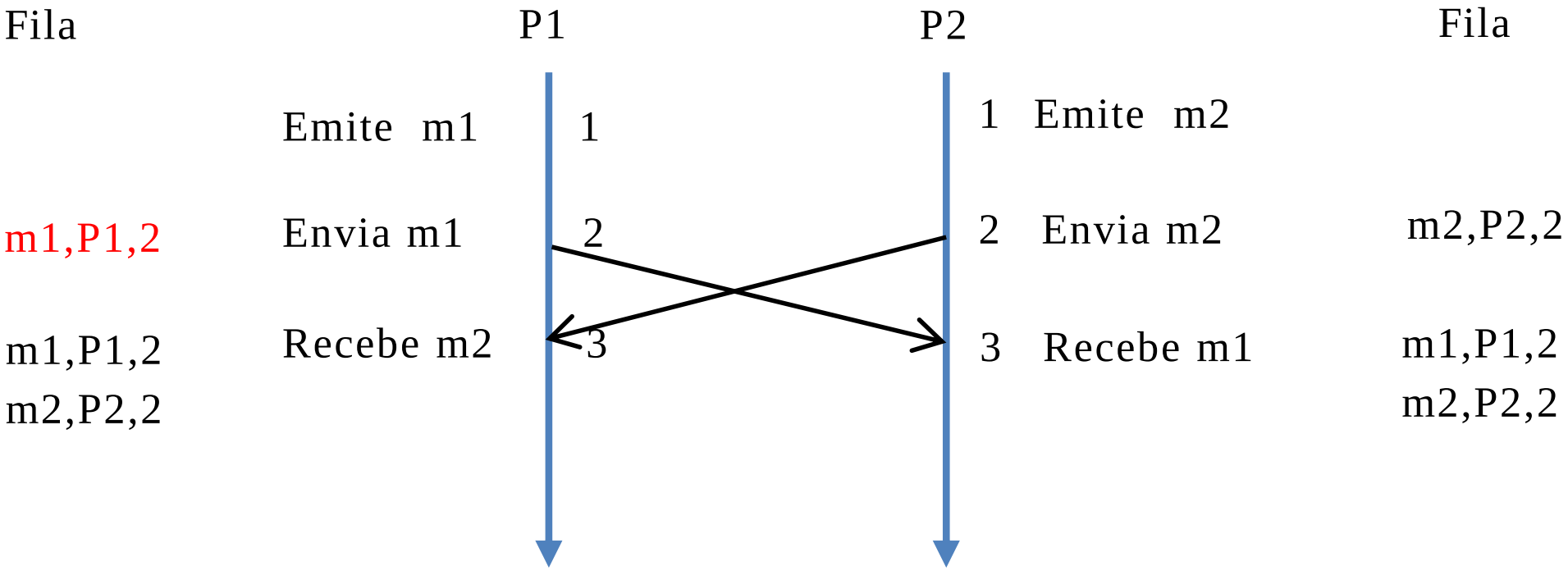
(1) m_i estiver no topo da fila $queue_j$

(2) todos os acknowledgements foram recebidos para m_i

Nota

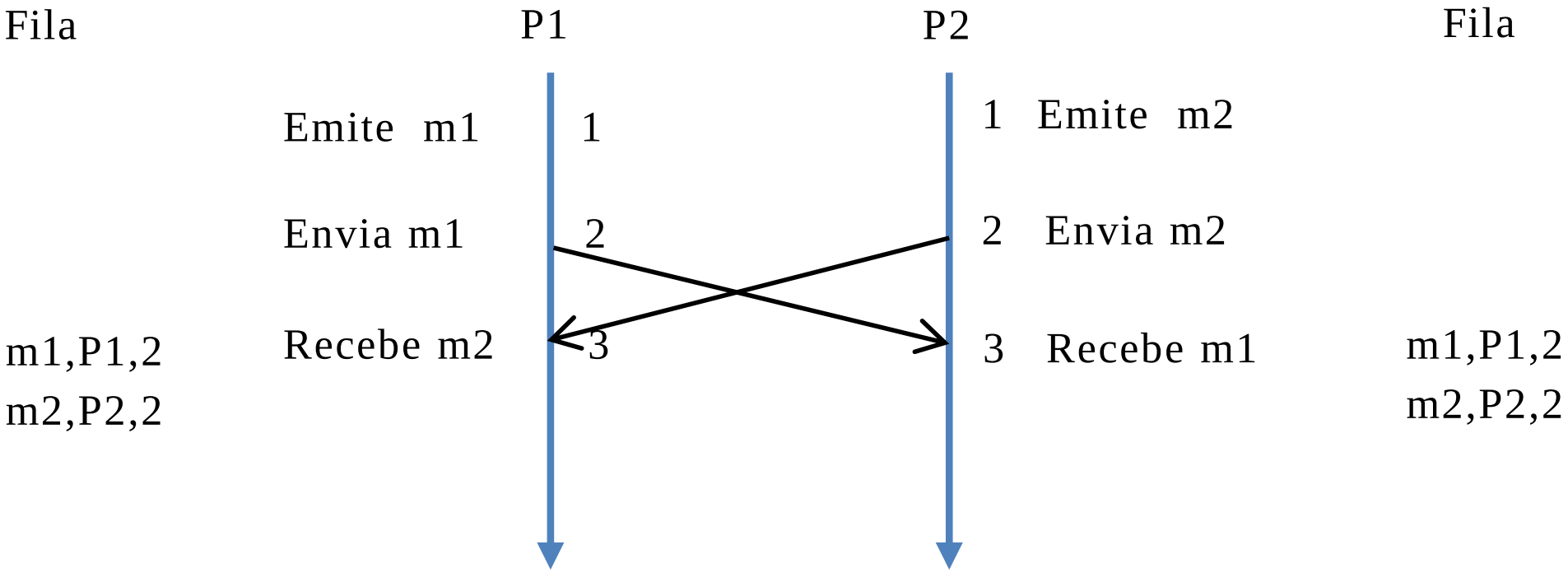
Assumimos que a comunicação é confiável e que a ordem FIFO (entre as mensagens enviadas por um processo) é respeitada.

Relógio lógico: multicast com ordem total



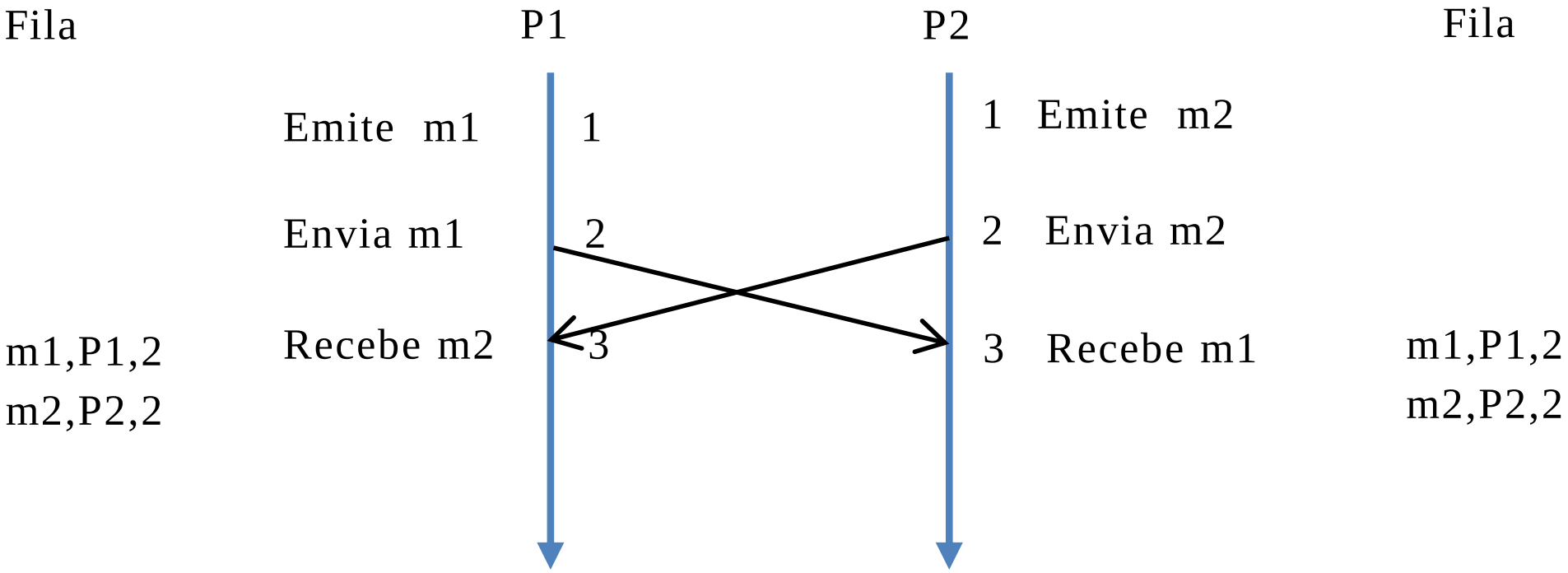
- Fila ordenada por prioridade (mais acima, mais prioridade)
- Emite m, para P1 ou P2, significa enviar a operação de update ao BD. Não quer dizer que essa operação será realizada imediatamente (pois depende do outro processo).
- Ao mesmo tempo que P1 emite m1, P2 emite m2
- m1 tem mais prioridade que m2 [= timestamp, porém pr(P1) > pr(P2)]

Relógio lógico: multicast com ordem total



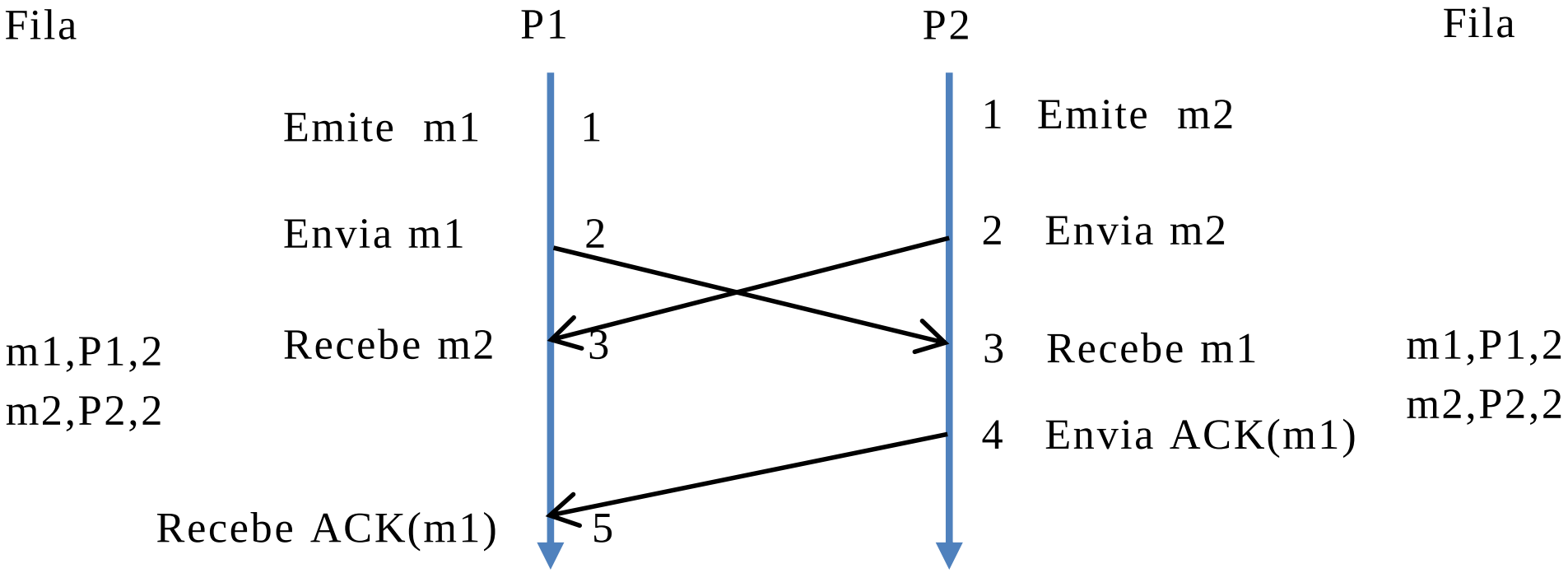
- Agora, P1 deve analisar se pode enviar ou não o ACK para cada mensagem da fila.
- P1 enviará um ACK (m1,P1,2) para P2 ? Sim, porque o processo associado a m1 corresponde ao mesmo processo que está analisando.
- P1 enviará um ACK (m2,P2,2) para P2? NÃO, porque:
    - o processo associado a m2 tem menor prioridade que P1
    - P1 não recebeu ainda o ACK para (m1,P1,2) vindo de P2.

Relógio lógico: multicast com ordem total



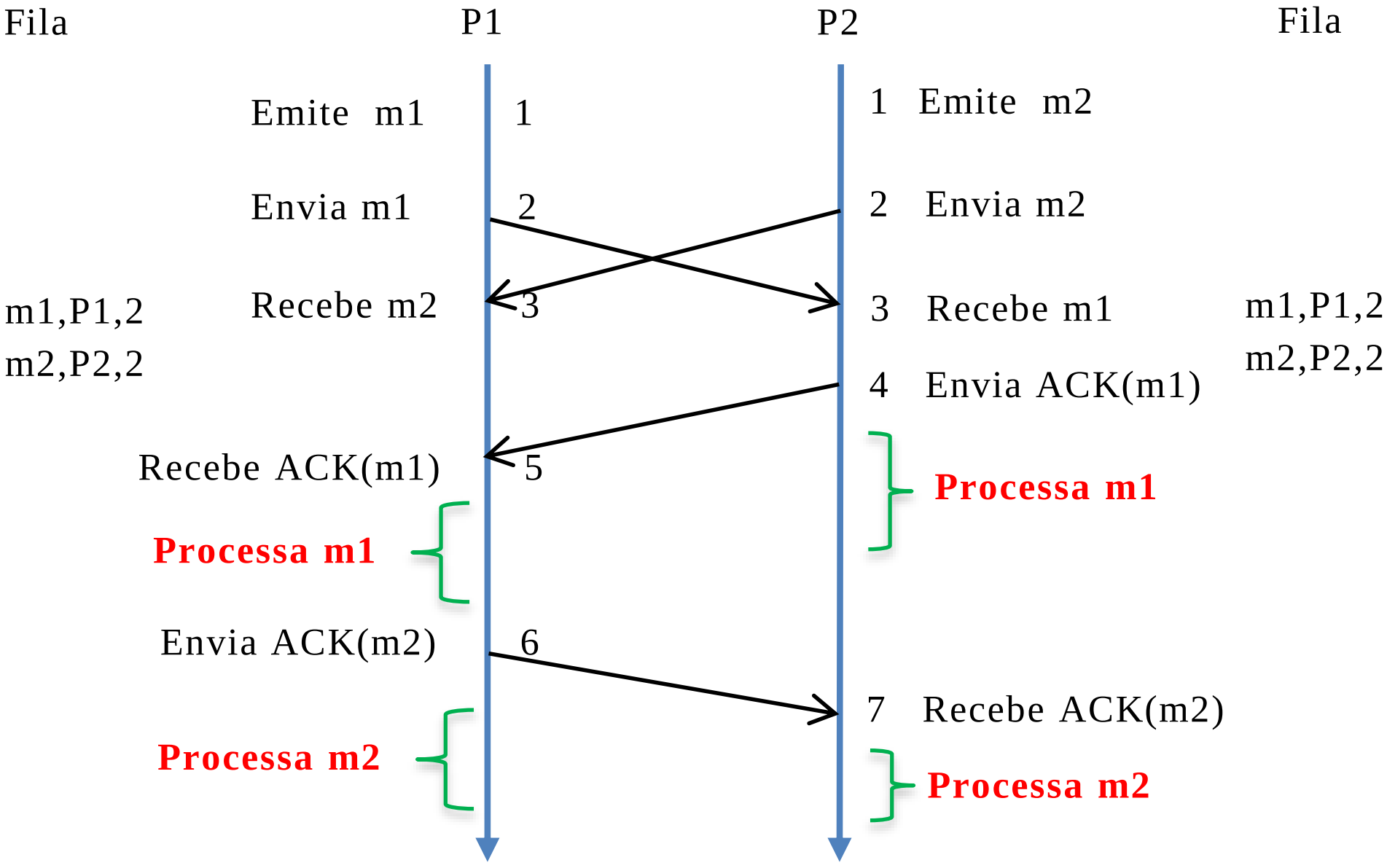
- Por sua vez, P2 também deve analisar se pode enviar ou não o ACK para cada mensagem da fila.
- P2 enviará um ACK (m1,P1,2) para P1 ? Sim, porque o processo associado a m1 tem mais prioridade que P2.
- P2 enviará um ACK (m2,P2,2) para P1? Sim porque o processo associado a m2 corresponde ao mesmo processo que está analisando

Relógio lógico: multicast com ordem total



- Resumindo:
- P1 enviou ACK(m1) e recebeu de P2 ACK(m1) e ACK(m2)
- P2 enviou ACK(m1), ACK(m2) e recebeu de P1 somente ACK(m1)

Relógio lógico: multicast com ordem total



- Quando P1 processa m1, pode enviar ACK(m2,P2,2)

Relógio lógico: o algoritmo de multicast funciona?

Observe que:

- se uma mensagem *m* ficar pronta em um processo *S*, *m* foi recebida por todos os outros processos (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 (pela ordem FIFO)*

- se *n* for originada em outro lugar que *m* é um pouco mais complicado. Pode ser que *m* e *n* cheguem em ordem diferente em *S*, mas é certeza de que antes de tirar um deles da fila, *S* terá que receber os ACKs de todos os outros processos, o que permitirá comparar os valores dos relógios e *repassar* à aplicação as mensagens na ordem total

Agenda

Sincronização de relógios

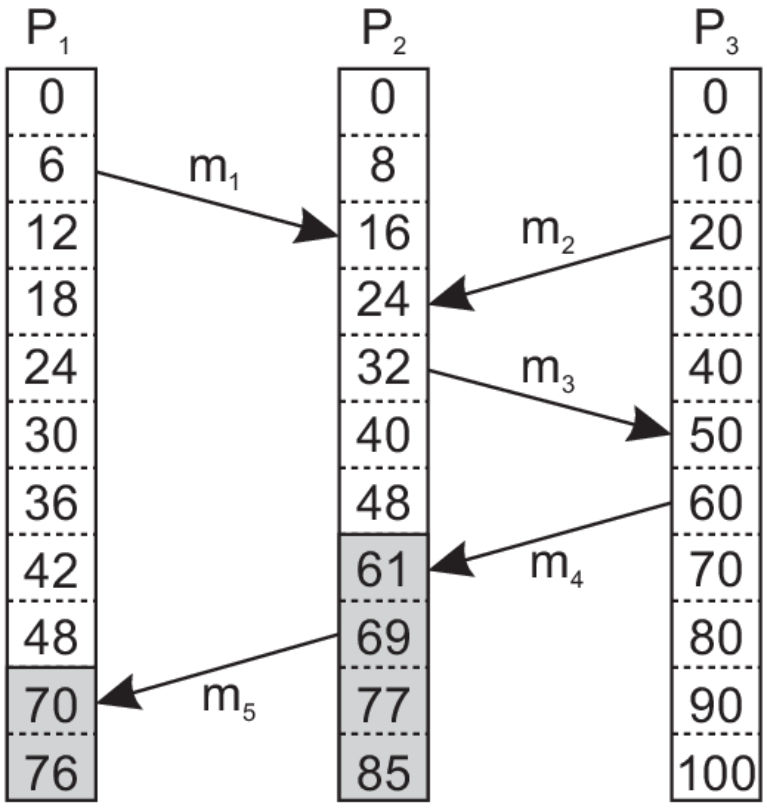
- Relógios físicos
- Relógios lógicos (de lamport)
- Relógios vetoriais

Relógios vetoriais

Ordem nas mensagens vindas de diferentes processos

Relógios vetoriais

O relógio de Lamport garante que se $a \rightarrow b$, então $C(a) < C(b)$. Porém **não** garante que se $C(a) < C(b)$ então a tenha ocorrido antes de b. Ou seja, não sabemos o relacionamento de a e b só comparando seus timestamps.



Exemplo:

Evento a: m_1 foi enviado em $T = 6$;
Evento b: m_2 foi enviado em $T = 20$.

Ou seja, $C(a) < C(b)$ porém não podemos concluir que $a \rightarrow b$.

Lembre que: "Se dois eventos x e y ocorrem em processos distintos e nunca interagem então nem $x \rightarrow y$ nem $y \rightarrow x$ são verdade"

Relógios vetoriais: capturando a dependência causal

Relógios vetoriais [Fidge, Mattern 1988] foram criados para resolver as limitações do relógio de Lamport, *i.e.*, o fato de que ele **não garante que se** $C(a) < C(b)$ **então** $a \rightarrow b$.

Solução: cada P_i mantém um vetor VC_i (Vector Clock)

$VC_i[i]$ é o relógio lógico local do processador P_i

$VC_i[j] = k$, então P_i sabe que k eventos ocorreram em P_j.

Relógios vetoriais: capturando a dependência causal

Relógios vetoriais [Fidge, Mattern 1988] foram criados para resolver as limitações do relógio de Lamport, *i.e.*, o fato de que ele **não garante que se** $C(a) < C(b)$ **então** $a \rightarrow b$.

Solução: cada P_i mantém um vetor VC_i (Vector Clock)

$VC_i[i]$ é o relógio lógico local do processador P_i

$VC_i[j] = k$, então P_i sabe que k eventos ocorreram em P_j .

Definição

Dizemos que ***b*** **pode depender causalmente** de ***a*** se ***ts*(*a*) < *ts*(*b*)** com:

para todo j, $ts(a)[j] \leq ts(b)[j]$
e existe pelo menos um índice j' para o qual $ts(a)[j'] < ts(b)[j']$

e.g., $ts(a) = [1,2,1]$. $ts(b) = [1,2,2]$. $ts(a) < ts(b)$?

Relógios vetoriais: regras

1. antes da execução de um evento, P_i executa $VC_i[i] \leftarrow VC_i[i] + 1$

2. antes do processo P_i enviar uma mensagem m para P_j, ele define o *timestamp* (no vetor) de m, $ts(m)$, como VC_i (após executar o passo 1, registrando assim o envio)

3. no recebimento de uma mensagem m, o processo P_j define

   $VC_j[k] \leftarrow \max\{VC_j[k], ts(m)[k]\}$ para todo k

Relógios vetoriais: regras

1. antes da execução de um evento, P_i executa $VC_i[i] \leftarrow VC_i[i] + 1$

2. antes do processo P_i enviar uma mensagem m para P_j, ele define o *timestamp* (no vetor) de m, $ts(m)$, como VC_i (após executar o passo 1, registrando assim o envio)

3. no recebimento de uma mensagem m, o processo P_j define

   $VC_j[k] \leftarrow \max\{VC_j[k], ts(m)[k]\}$ para todo k

Muito similar à regra do relógio lógico de Lamport, porém específica para cada processo

Relógios vetoriais: regras

1. antes da execução de um evento, P_i executa $VC_i[i] \leftarrow VC_i[i] + 1$

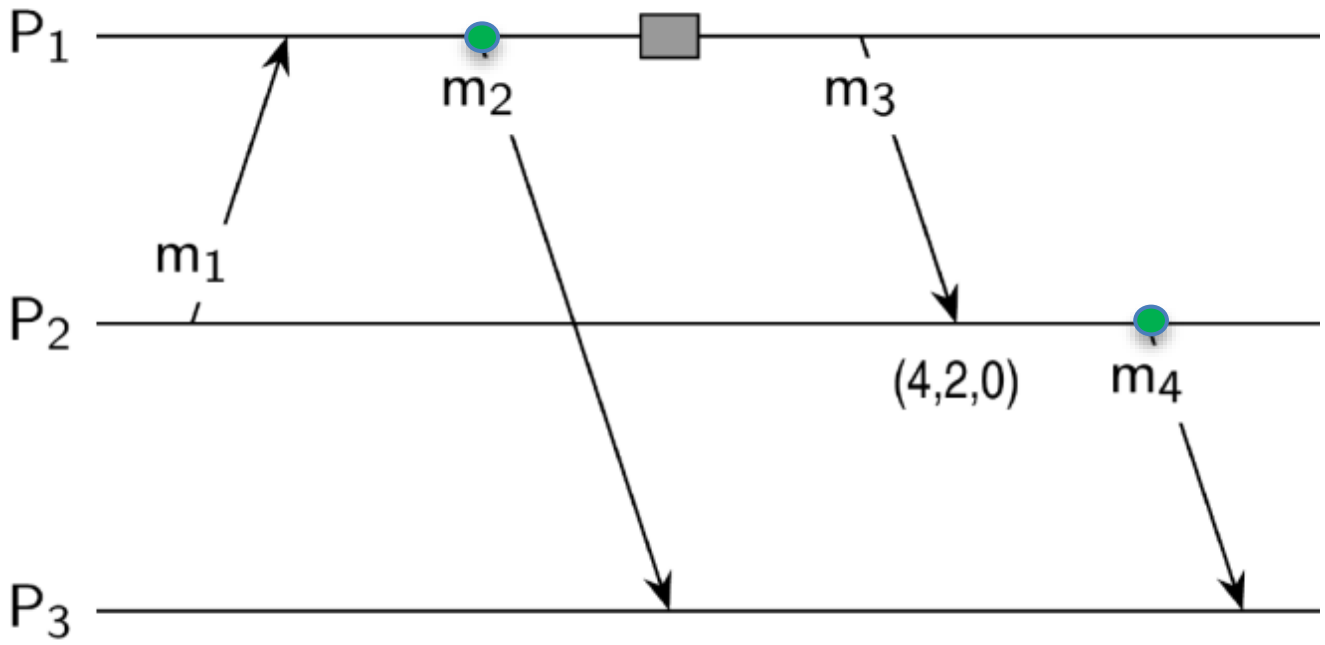
2. antes do processo P_i enviar uma mensagem m para P_j, ele define o *timestamp* (no vetor) de m, $ts(m)$, como VC_i (após executar o passo 1, registrando assim o envio)

3. no recebimento de uma mensagem m, o processo P_j define

   $VC_j[k] \leftarrow \max\{VC_j[k], ts(m)[k]\}$ para todo k
   e executa o passo 1 (registrando assim o recebimento)

Muito similar à regra do relógio lógico de Lamport, porém específica para cada processo

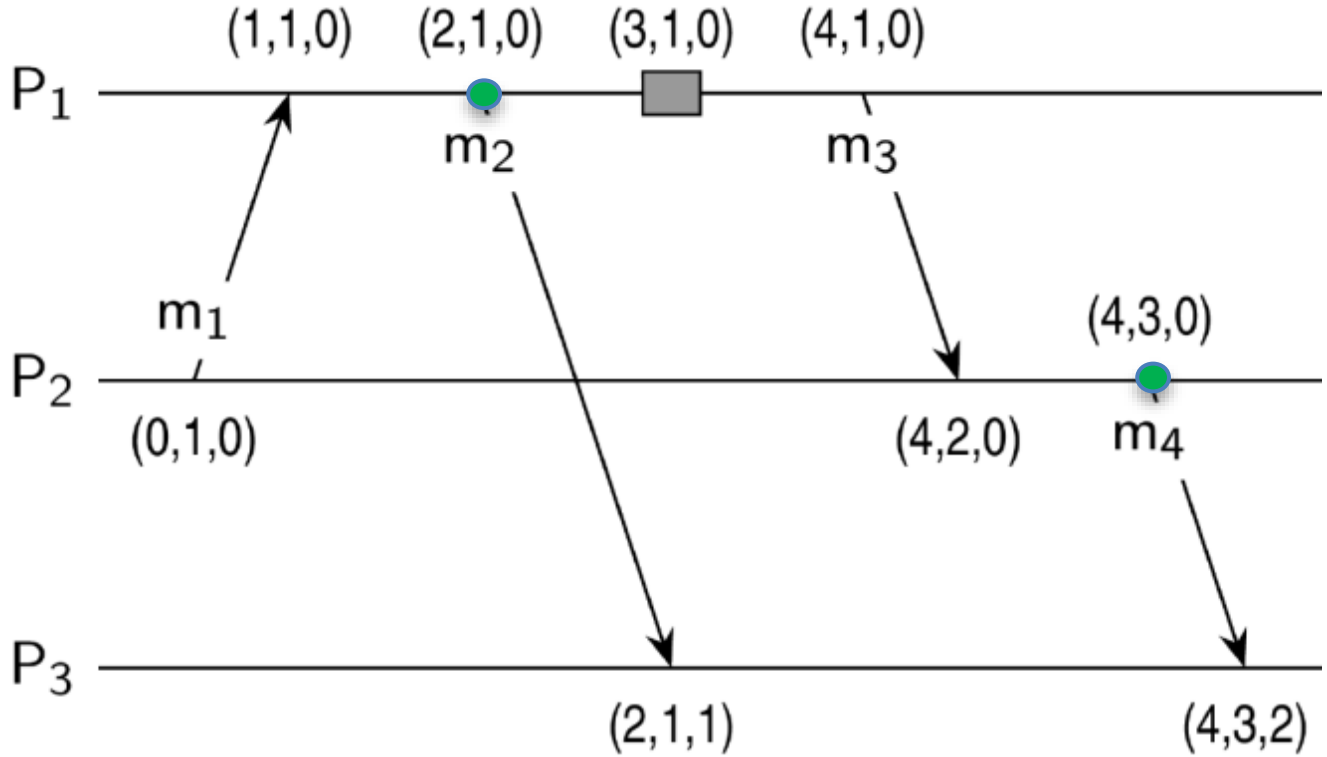
Relógios Vetoriais: Exemplo



Assuma que os relógios começam em zero

- Insira o relógios vetoriais para cada evento.
- Analise o VCs no envio (em verde) dos eventos *m*2 e *m*4
  i.e., se $ts(m2) < ts(m4)$ e a conclusão

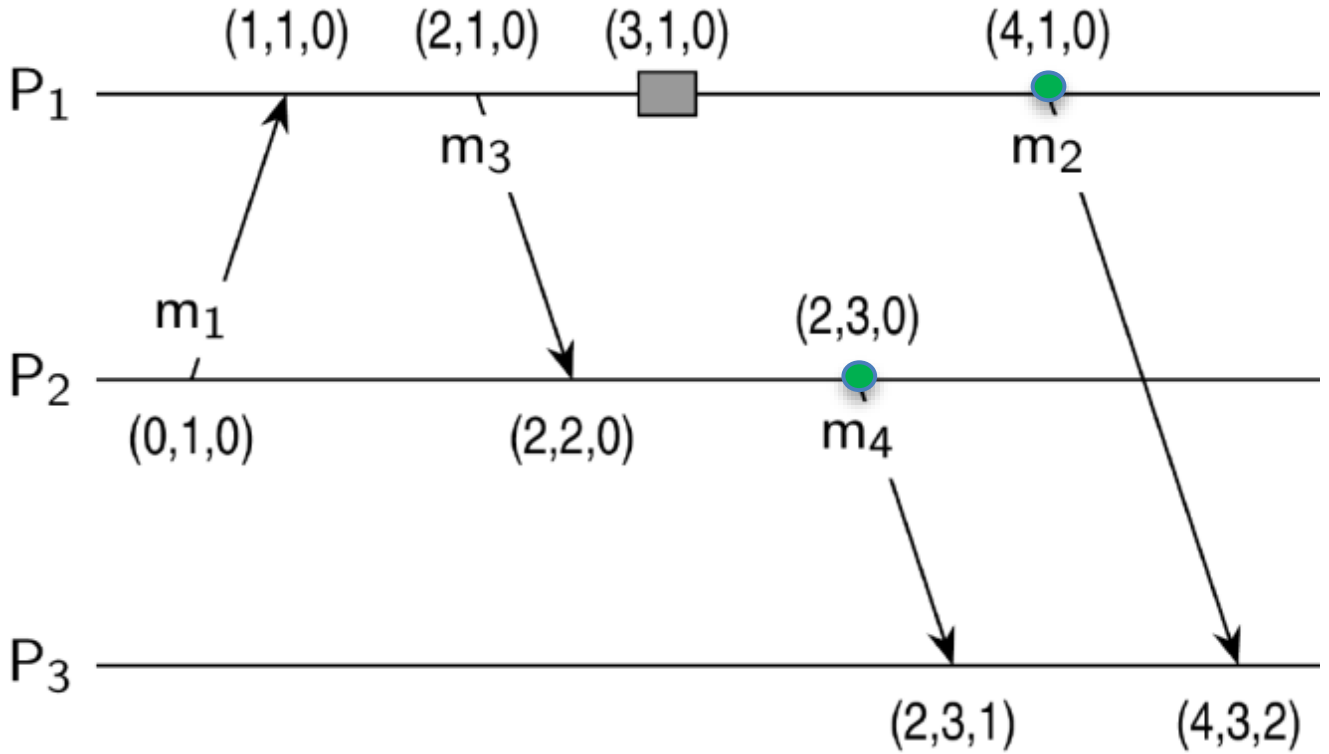
Relógios Vetoriais: Exemplo



ts(m2)	ts(m4)	ts(m2) < ts(m4)	ts(m2) > ts(m4)	Conclusão
(2,1,0)	(4,3,0)	Sim	Não	m2 pode preceder causalmente m4, m2→m4

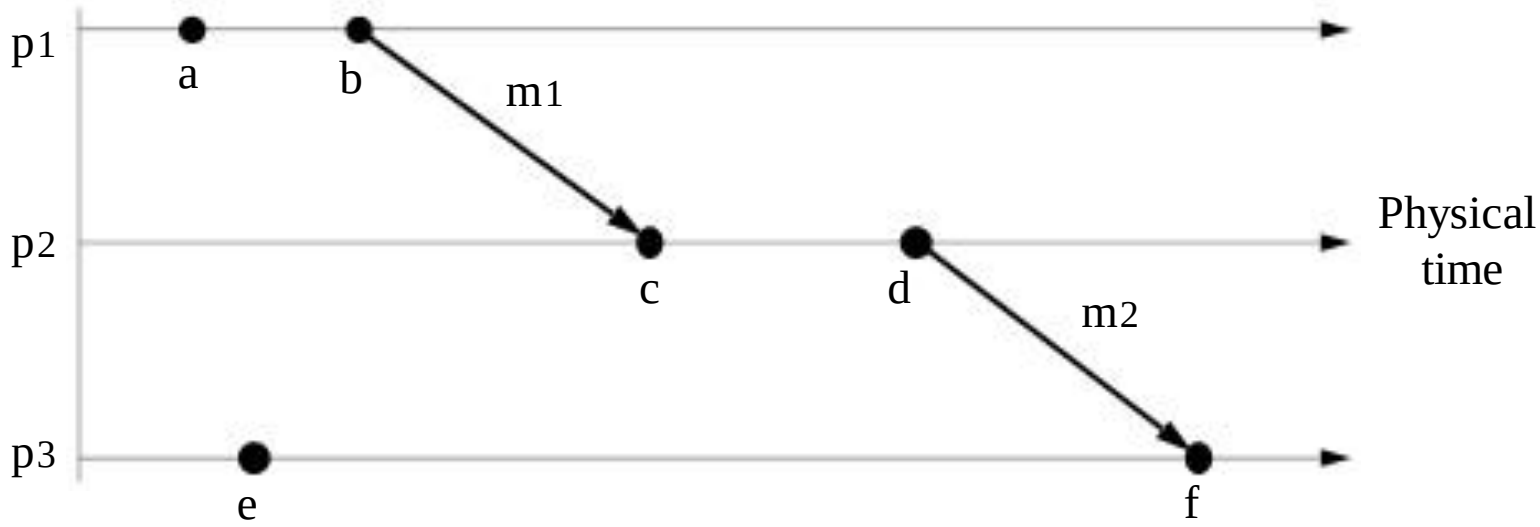
Relógios Vetoriais: Exemplo

Suponha agora um atraso no envio de $m2$:



ts(m2)	ts(m4)	ts(m2) < ts(m4)	ts(m2) > ts(m4)	Conclusão
(4,1,0)	(2,3,0)	Não	Não	m2 e m4 podem ser concorrentes (e conflitar)

Relógios Vetoriais: Exercício

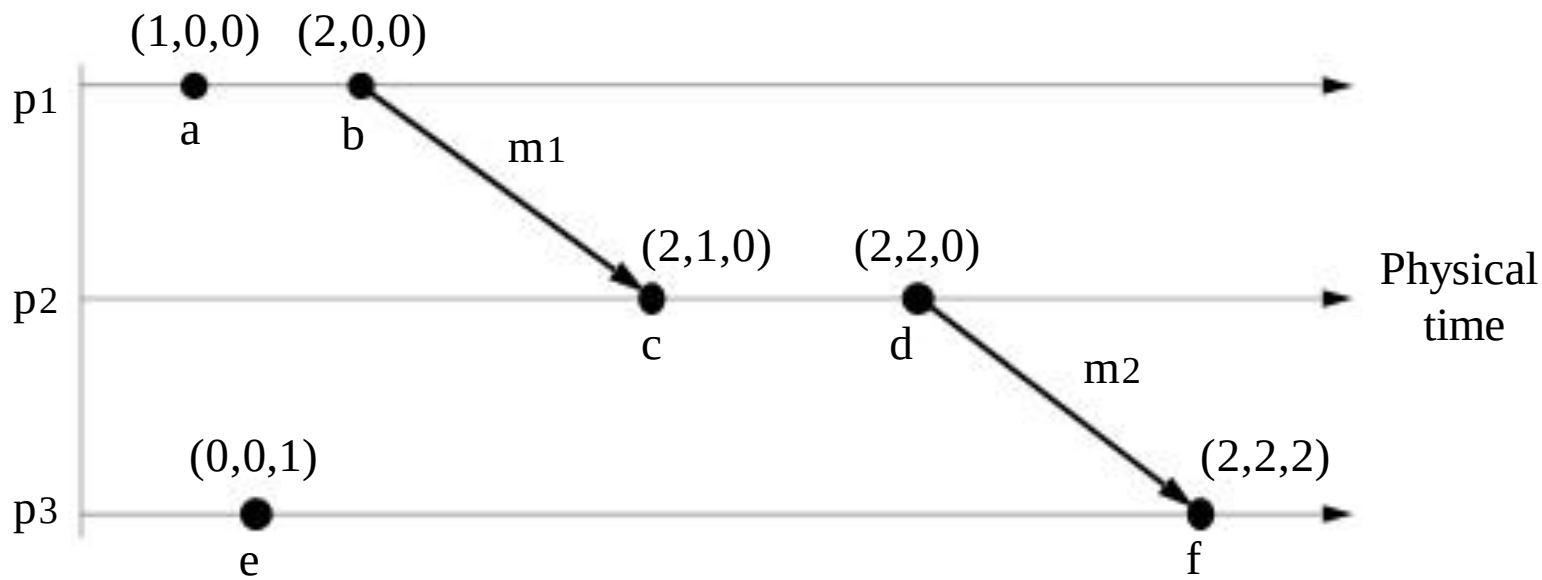


Fonte: CDKB

Exercício

1. Insira os relógios vetoriais para cada evento

Relógios Vetoriais: Exercício



Fonte: CDKB

Exercício

1. O que pode ser dito sobre **a** e **f**?
2. O que pode ser dito sobre **c** e **e**?

Relógios Vetoriais: multicast

Relógios Vetoriais: multicast

- Relógios vetoriais
  - → Multicast de Ordem Causal

Relógios Vetoriais: Multicast ordenado por causalidade

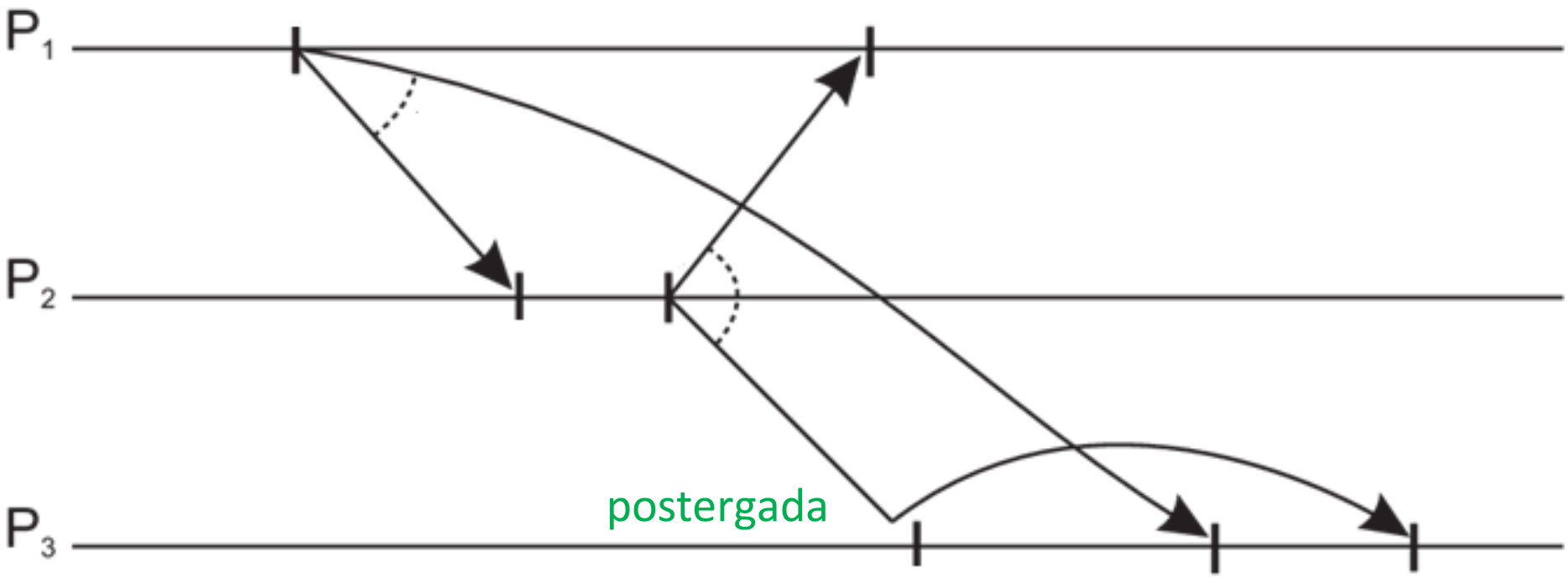
Ideia

Garantir que uma mensagem seja *repassada* (à aplicação) somente **se todas as mensagens que as precederem por causalidade tiverem sido repassadas**.

Multicasts ordenados por causalidade **são menos restritivos** do que multicasts *com ordem total*. Se duas mensagens não tem uma relação causal, então a ordem que elas serão repassadas (delivered) pode ser **diferente para cada um dos processos**.

Relógios Vetoriais: Multicast ordenado por causalidade

Ideia



O relógio será ajustado somente quando enviar ou *repassar* uma mensagem *m* (e não quando receber *m*)

Relógios Vetoriais: Multicast ordenado por causalidade

Para garantir que as mensagens serão *repassadas* seguindo a ordem causal:

Passos

1. P_i incrementa $VC_i[i]$ **somente quando enviar** uma mensagem;
2. P_j "ajusta" VC_j quando **repassar**[1] uma mensagem:

$$VC_i[k] = \max\{VC_j[k], ts(m)[k]\}, \forall k \; != j$$

Note que não incrementa o vetor quando não é envio

[1] **Atenção**: as mensagens **não** são ajustadas quando são *recebidas*, mas sim quando elas são *repassadas (delivered)* à aplicação

Relógios Vetoriais: Multicast ordenado por causalidade

Para garantir que as mensagens serão *repassadas* seguindo a ordem causal:

Passos

1. P_i incrementa $VC_i[i]$ **somente quando enviar** uma mensagem;
2. P_j "ajusta" VC_j quando **repassar**[1] uma mensagem:

$$VC_i[k] = \max\{VC_j[k], ts(m)[k]\}, \forall k \; != j$$

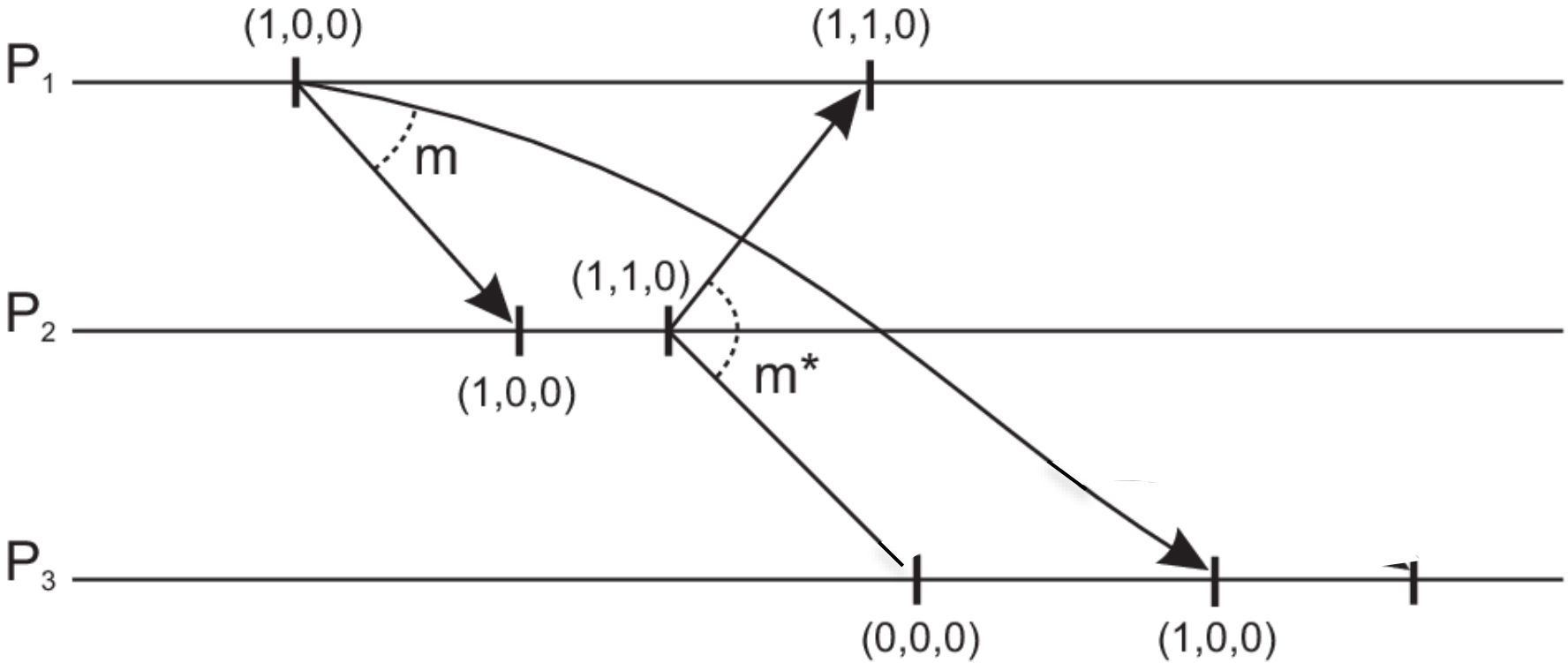
Além disto, P_j posterga o **repasse** de m (enviada por P_i) até que:

- $ts(m)[i] = VC_j[i] + 1$. (m é a próxima msg que P_j espera de P_i)
- $ts(m)[k] \leq VC_j[k]$ para k != i. (P_j já repassou todas as mensagens P_i que repassou)

[1] **Atenção**: as mensagens **não** são ajustadas quando são *recebidas*, mas sim quando elas são *repassadas (delivered)* à aplicação

Relógios Vetoriais: Multicast ordenado por causalidade

Exemplo



Relógios Vetoriais: Multicast ordenado por causalidade

Exercício

Tome $VC_3 = [0, 2, 2]$, $ts(m) = [1, 3, 0]$ vinda de P_1. Que informação P_3 tem antes de receber m e o que ele irá fazer quando receber m ?

Relógios Vetoriais: DynamoDB

Amazon DynamoDB [Amazon 2012]

- Serviço de banco de dados NoSQL rápido e para qualquer escala
- Alto desempenho para 99% das requisições: acesso a dados em < 300ms em 2007 (10 ms segundo o site em 2019)
- Alta disponibilidade, consistência eventual

Relógios Vetoriais: DynamoDB

Consistência

A consistência é obtida por versionamento (com relógios vetoriais).

Relógios Vetoriais: DynamoDB

Consistência

A consistência é obtida por versionamento (com relógios vetoriais).

O objetivo fundamental do design é: “writes are never rejected”

“For a number of Amazon services, rejecting customer updates could result in a poor customer experience. For instance, the shopping cart service must allow customers to add and remove items from their shopping cart even amidst network and server failures.”

Relógios Vetoriais: DynamoDB

Consistência

A consistência é obtida por versionamento (com relógios vetoriais).

O objetivo fundamental do design é: “writes are never rejected”

“For a number of Amazon services, rejecting customer updates could result in a poor customer experience. For instance, the shopping cart service must allow customers to add and remove items from their shopping cart even amidst network and server failures.”

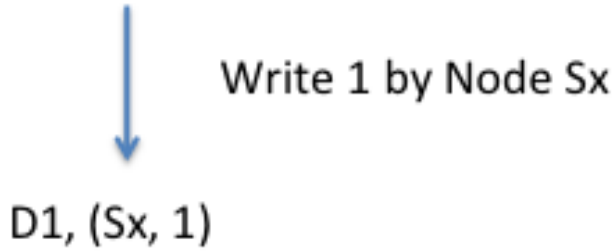
Nota: writes em diferentes servidores (ao mesmo tempo) podem gerar inconsistências.

Na versão [2022] usam Paxos (Capítulo 8)

“This requirement forces us to push the complexity of conflict resolution to the reads”

Relógios Vetoriais: DynamoDB

Consistência

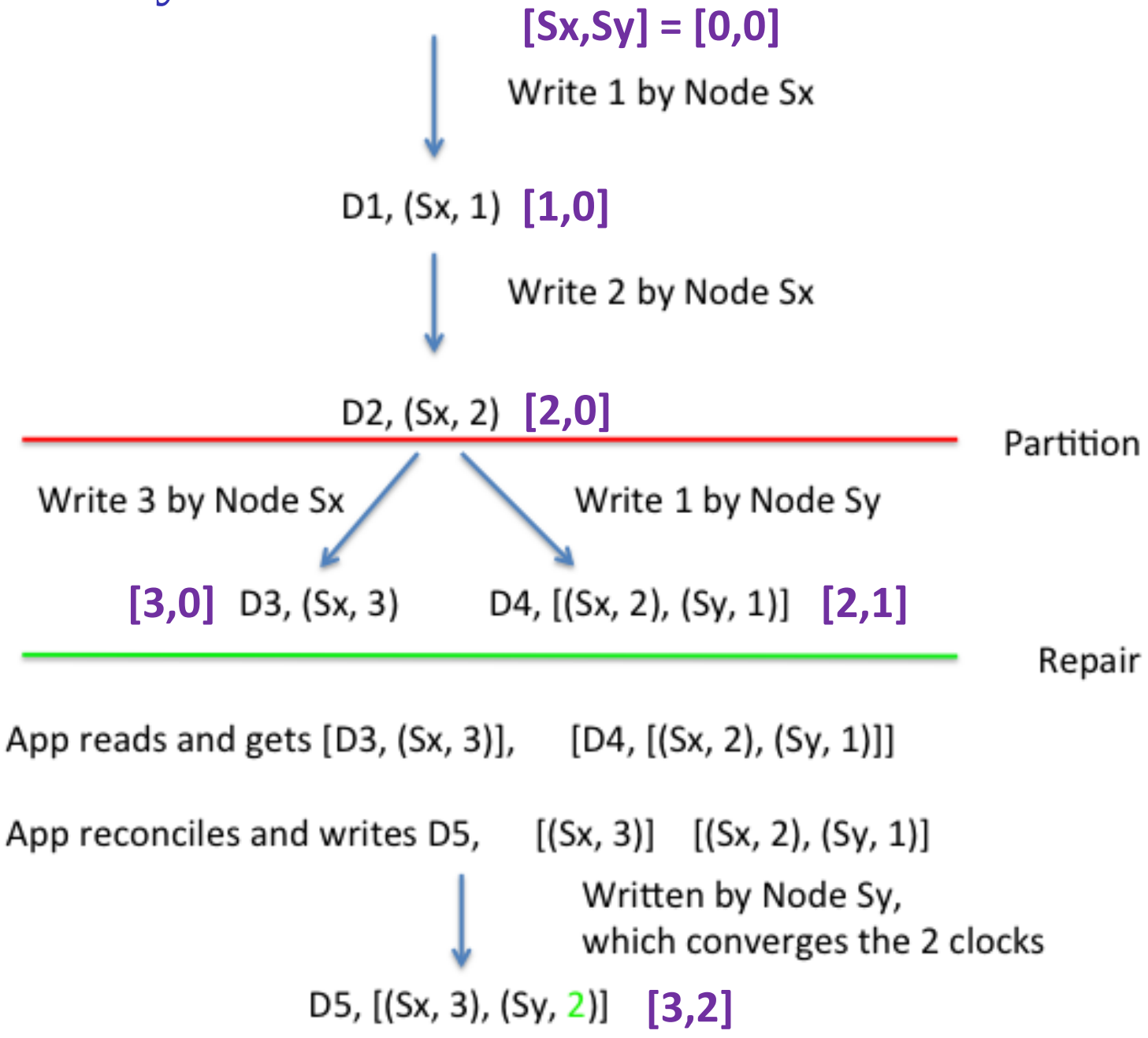


Até aqui sem reconciliação

Reconciliação na leitura

Relógios Vetoriais: DynamoDB

Consistência



Relógios Vetoriais: DynamoDB

Consistência

In our next experiment (2007), the number of versions returned to the shopping cart service was profiled for a period of 24 hours.

- 99.94% of requests saw exactly one version;
- 0.00057% of requests saw 2 versions;
- 0.00047% of requests saw 3 versions and
- 0.00009% of requests saw 4 versions.

This shows that divergent versions are created rarely.

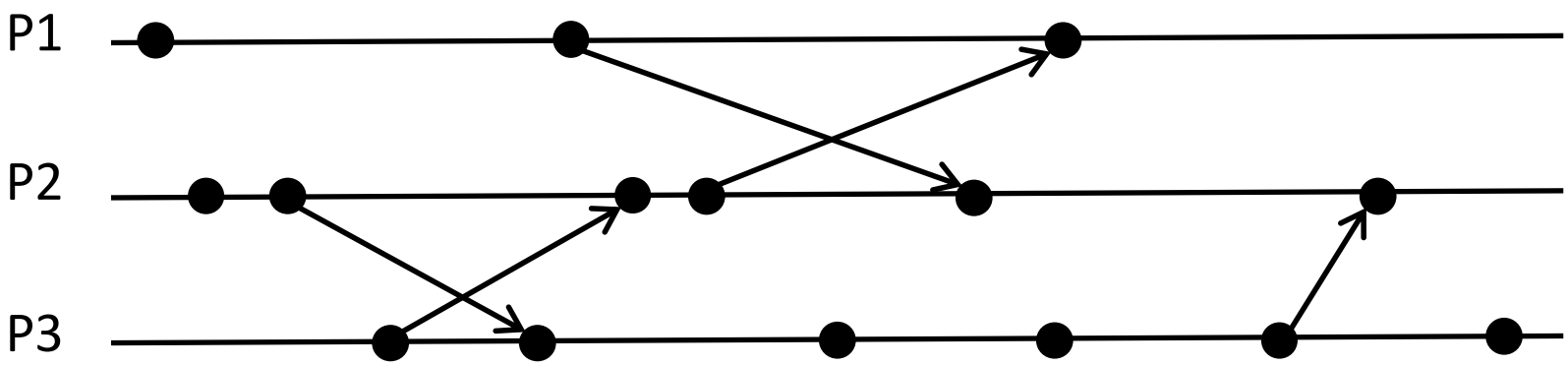
Porém “DynamoDB can handle more than 10 trillion requests per day and can support peaks of more than 20 million requests per second”. Em [2022], foram 89 milhões de requisições por segundo.

Note que, para 20 milhões de requisições, 12.000 terão +1 versão

Conceitos adquiridos

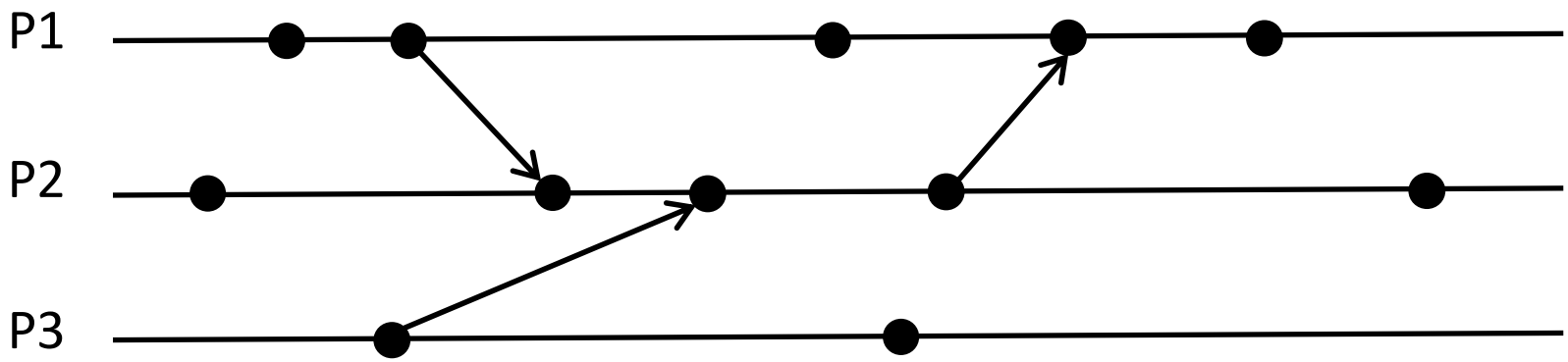
- Relógios físicos: sincronização interna e externa.
- Algoritmo de Cristian e de Berkeley.
- Enviar/Receber/Entregar uma mensagem.
- Relação happened-before (aconteceu-antes)
- Relógios lógicos (Lamport) e vetoriais
- Multicast de ordem total
- Multicast de ordem causal
- DynamoDB.

Relógio lógico: Exercício



Insira o relógio lógico (de Lamport) para cada evento.
Assuma que os relógios começam em zero

Relógio Vetorial: Exercício



Insira o relógio vetorial para cada evento.
Assuma que os relógios começam em zero