# ACH 2147 — DESENVOLVIMENTO DE SISTEMAS DE INFORMAÇÃO DISTRIBUÍDOS

## COORDENAÇÃO

Daniel Cordeiro

13 e 15 de junho de 2018

Escola de Artes, Ciências e Humanidades | EACH | USP

# NOVA DATA DA P2 PARA T.94

**t.94** ~~27 de junho~~ quarta, 4 de julho de 2018

**t.04** sexta, 29 de junho de 2018

# SINCRONIZAÇÃO DE RELÓGIOS

- relógios físicos
- relógios lógicos
- relógios vetoriais

**Problema**
Algumas vezes precisamos saber a hora exata e não apenas uma ordenação de eventos.

**Coordinated Universal Time (UTC):**

- baseado no número de transições por segundo do átomo de césio 133 (bastante preciso)
- atualmente, o tempo é medido como a média de cerca de 50 relógios de césio espalhados pelo mundo
- introduz um *segundo bissexto* de tempos em tempos para compensar o fato de que os dias estão se tornando maiores

**Nota:**
O valor do UTC é enviado via *broadcast* por satélite e por ondas curtas de rádio. Satélites tem um acurácia de $\pm 0.5$ ms.

## Precisão

O objetivo é tentar fazer com que o desvio entre dois relógios em quaisquer duas máquinas fique dentro de um limite especificado, conhecido como a precisão $\pi$:

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

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

## Acurácia

No caso da acurácia, queremos manter o relógio limitado a um valor $\alpha$:

$$\forall t, \forall p : |C_p(t) - t| \leq \alpha$$

## Sincronização

**Sincronização interna:** manter a precisão dos relógios

**Sincronização externa:** manter a acurácia dos relógios

## Especificação dos relógios

- Todo relógio tem especificado sua taxa máxima de desvio do relógio $\rho$.
- $F(t)$: frequência do oscilador do relógio do hardware no tempo $t$
- $F$: frequência (constante) do relógio ideal:

$$\forall t : (1 - \rho) \leq \frac{F(t)}{F} \leq (1 + \rho)$$

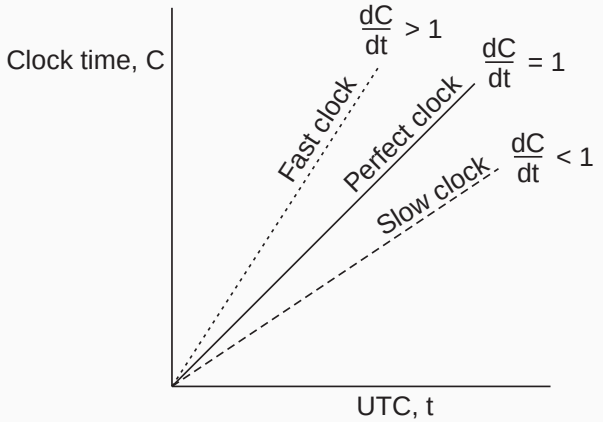
## Observação

Interrupções de hardware acoplam um relógio de software a um relógio de hardware, que também tem sua taxa de desvio:

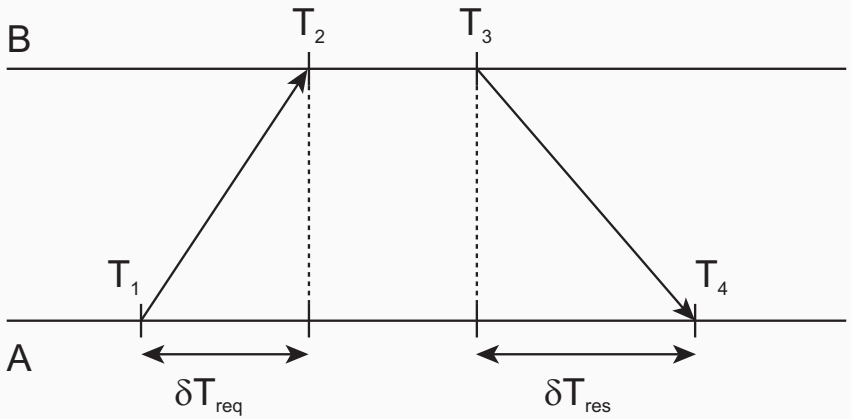
$$C_p(t) = \frac{1}{F} \int_0^t F(t)dt \Rightarrow \frac{dC_p(t)}{dt} = \frac{F(t)}{F}$$

$$\Rightarrow \forall t : 1 - \rho \leq \frac{dC_p(t)}{dt} \leq 1 + \rho$$

## Relógios rápidos, perfeitos e lentos



## Recuperação do horário atual de um servidor




## Cálculo da diferença relativa $\theta$ e o atraso $\delta$

Assumindo que: $\delta T_{req} = T_2 - T_1 \approx T_4 - T_3 = \delta T_{res}$

$$\theta = T_3 + \big((T_2 - T_1) + (T_4 - T_3)\big)/2 - T_4 = \big((T_2 - T_1) + (T_3 - T_4)\big)/2$$

$$\delta = \big((T_4 - T_1) - (T_3 - T_2)\big)/2$$

## Network Time Protocol

Colete oito pares $(\theta, \delta)$ e escolha os $\theta$ cujos atrasos $\delta$ sejam minimais.

**Sincronização externa**
Cada máquina pede a um *servidor de hora* a hora certa pelo menos uma vez a cada $\delta/(2\rho)$ (Network Time Protocol)

**OK, mas...**
você ainda precisa de uma maneira precisa de medir o *round trip delay*, incluindo o tratamento da interrupção e o processamento das mensagens.

**Sincronização interna**
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 provavelmente terá todas as máquinas em sincronia. Você nem precisa propagar o horário UTC.

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

# Relógios lógicos

O que importa na maior parte dos sistemas distribuídos não é fazer com que todos os processos concordem exatamente com o horário, 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.

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

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

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

- 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 $a \rightarrow b$

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

- 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 $a \rightarrow c$

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

- 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 $a \rightarrow c$

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

- 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 $a \rightarrow c$

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

**Problema**

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

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

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

**Outro problema**
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

## Solução

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

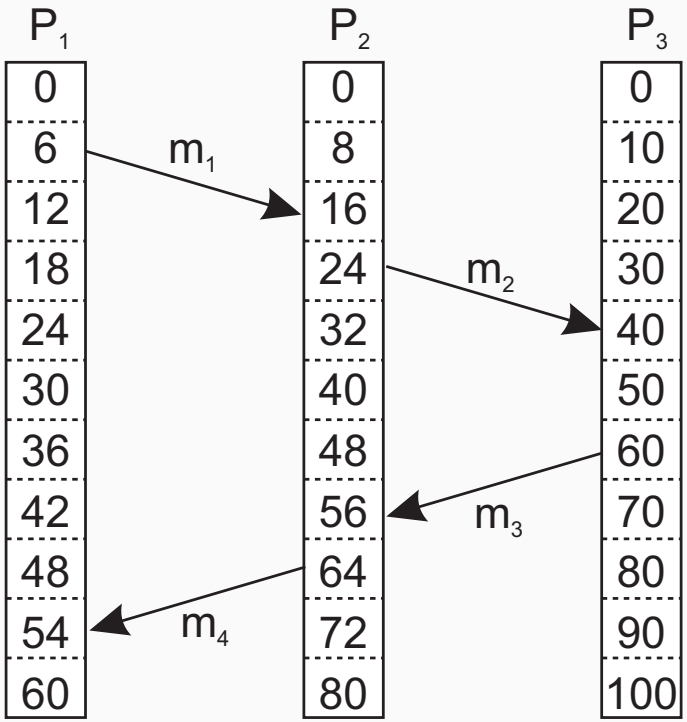
1. para quaisquer dois **eventos sucessivos** que ocorrer em $P_i$, $C_i$ é incrementado em 1
2. toda vez que uma mensagem $m$ for enviada por um processo $P_i$, a mensagem deve receber um *timestamp* $ts(m) = C_i$
3. sempre que uma mensagem $m$ for recebida por um processo $P_j$, $P_j$ ajustará seu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

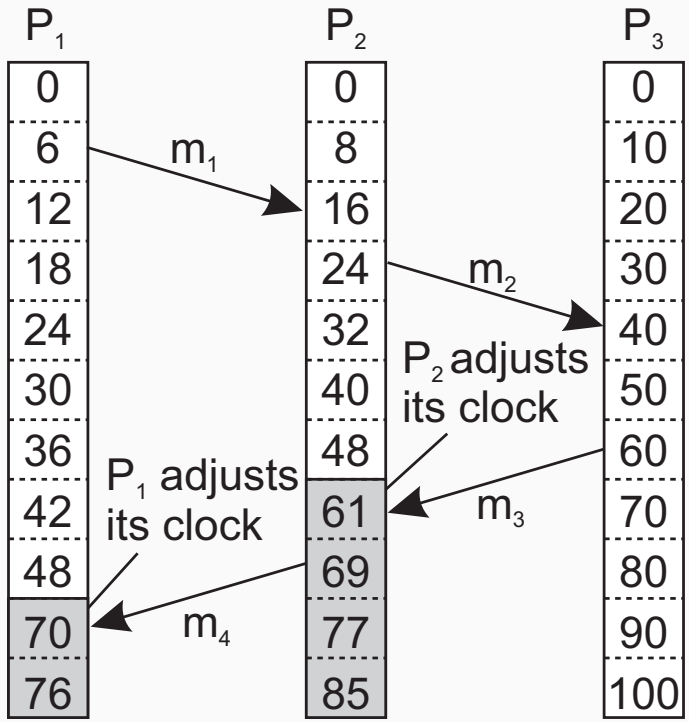
## Observações:

- a propriedade **P1** é satisfeita por (1); propriedade **P2** por (2) e (3)
- ainda assim pode acontecer de dois eventos ocorrerem ao mesmo tempo. Desempate usando os IDs dos processos.

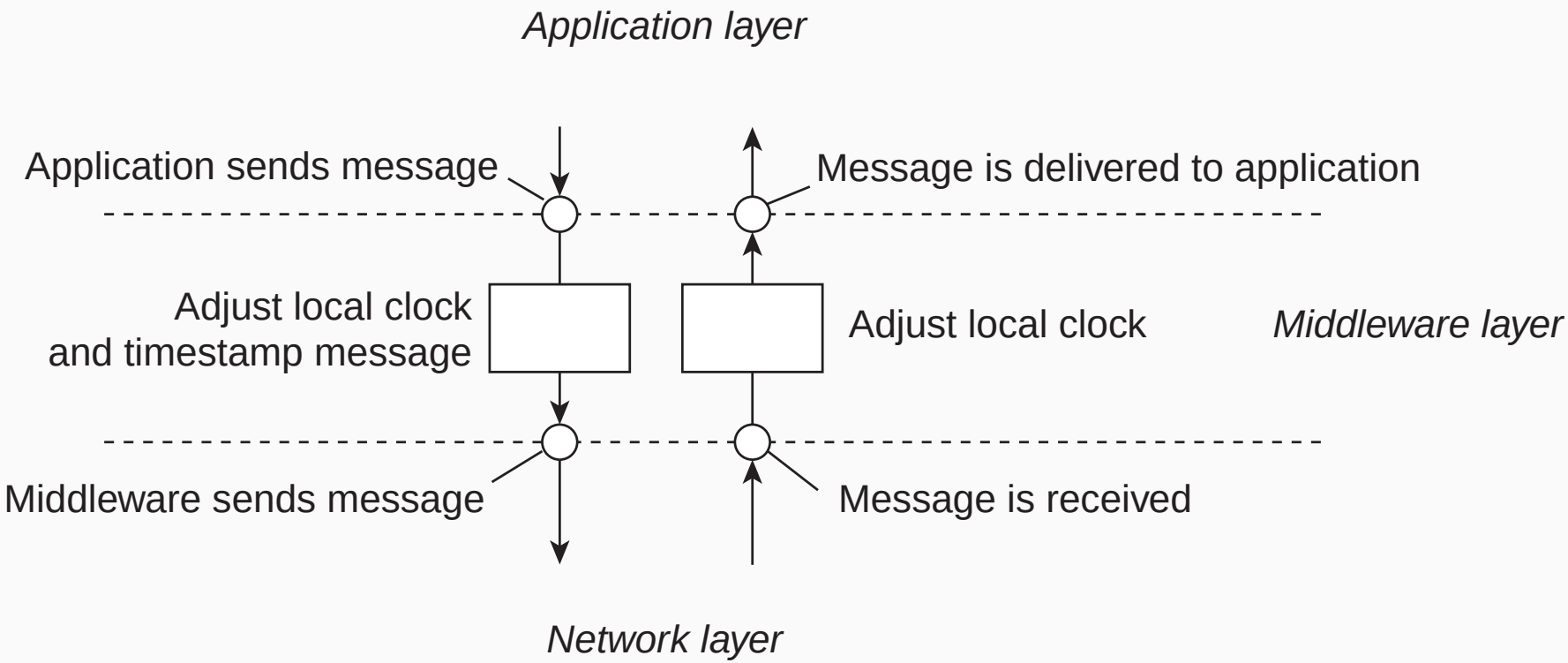
# RELÓGIO LÓGICO DE LAMPORT – EXEMPLO

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



**Nota**
Os ajustes ocorrem na camada do *middleware*

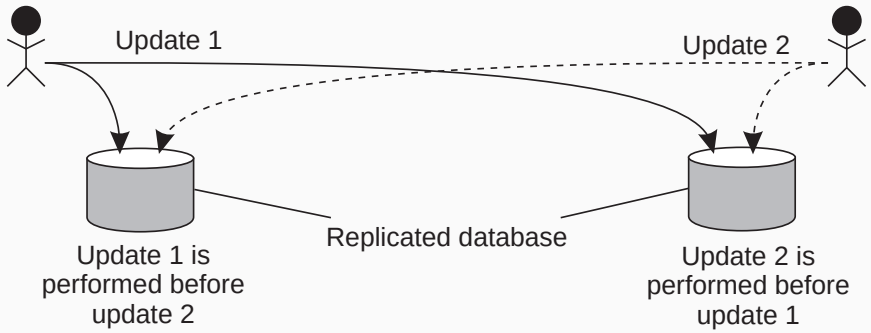


# EXEMPLO: MULTICAST COM ORDEM TOTAL

## Problema

Alguma vezes precisamos garantir que atualizações concorrentes em um banco de dados replicado sejam vistos 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%
- Há duas réplicas




## Resultado

Na ausência de sincronização correta,
réplica #1 ← R$ 1111, enquanto que na réplica #2 ← R$ 1110.

## Solução

- processo $P_i$ 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*) por todos os outros processos

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

(1) $m_i$ estiver na cabeça da fila $queue_j$

(2) para todo processo $P_k$, existe uma mensagem $m_k$ na $queue_j$ com um *timestamp* maior.

**Nota**

Assumimos que a comunicação é confiável e que a ordem FIFO é respeitada.

# O ALGORITMO DE MULTICAST FUNCIONA?

Observe que:

- 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 recebido)
- 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*
- se *n* 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 de que antes de tirar um deles da fila, ele terá que receber os ACKs de todos os outros servidores, o que permitirá comparar os valores dos relógios e entregar para as mensagens na ordem total dos relógios

# RELÓGIO DE LAMPORT PARA EXCLUSÃO MÚTUA

```
class Process:
  def __init__(self, chan):
    self.queue      = []                    # The request queue
    self.clock      = 0                     # The current logical clock

  def requestToEnter(self):
    self.clock = self.clock + 1                          # Increment clock value
    self.queue.append((self.clock, self.procID, ENTER)) # Append request to q
    self.cleanupQ()                                      # Sort the queue
    self.chan.sendTo(self.otherProcs, (self.clock,self.procID,ENTER)) # Send request

  def allowToEnter(self, requester):
    self.clock = self.clock + 1                          # Increment clock value
    self.chan.sendTo([requester], (self.clock,self.procID,ALLOW)) # Permit other

  def release(self):
    tmp = [r for r in self.queue[1:] if r[2] == ENTER]  # Remove all ALLOWs
    self.queue = tmp                                     # and copy to new queue
    self.clock = self.clock + 1                          # Increment clock value
    self.chan.sendTo(self.otherProcs, (self.clock,self.procID,RELEASE)) # Release

  def allowedToEnter(self):
    commProcs = set([req[1] for req in self.queue[1:]]) # See who has sent a message
    return (self.queue[0][1]==self.procID and len(self.otherProcs)==len(commProcs))
```

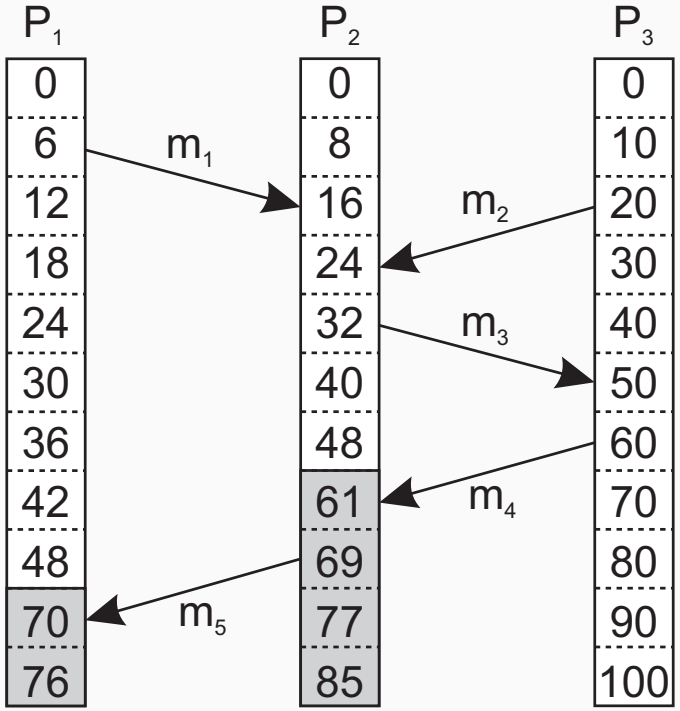
# RELÓGIO DE LAMPORT PARA EXCLUSÃO MÚTUA

```
def receive(self):
  msg = self.chan.recvFrom(self.otherProcs)[1]  # Pick up any message
  self.clock = max(self.clock, msg[0])          # Adjust clock value...
  self.clock = self.clock + 1                   # ...and increment
  if msg[2] == ENTER:
    self.queue.append(msg)                      # Append an ENTER request
    self.allowToEnter(msg[1])                   # and unconditionally allow
  elif msg[2] == ALLOW:
    self.queue.append(msg)                      # Append an ALLOW
  elif msg[2] == RELEASE:
    del(self.queue[0])                          # Just remove first message
  self.cleanupQ()                               # And sort and cleanup
```

## Analogia com multicast de ordem total

- No multicast de ordem total, todos os processos construíam fila idênticas, entregando as mensagens na mesma ordem
- Exclusão mútua implica em concordar sobre a ordem em que os processos devem ter sua entrada permitida na seção crítica

**Observação:**
Relógios de Lamport **não** garantem que $C(a) < C(b)$ implica que $a$ tenha realmente ocorrido antes de $b$:




**Observação**
Evento $a$: $m_1$ foi recebido em $T = 16$;
Evento $b$: $m_2$ foi enviado em $T = 20$.

**Nota**
Nós não podemos concluir que $a$ precede temporalmente (precedência causal) $b$.

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

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

**Precedência vs. dependência**

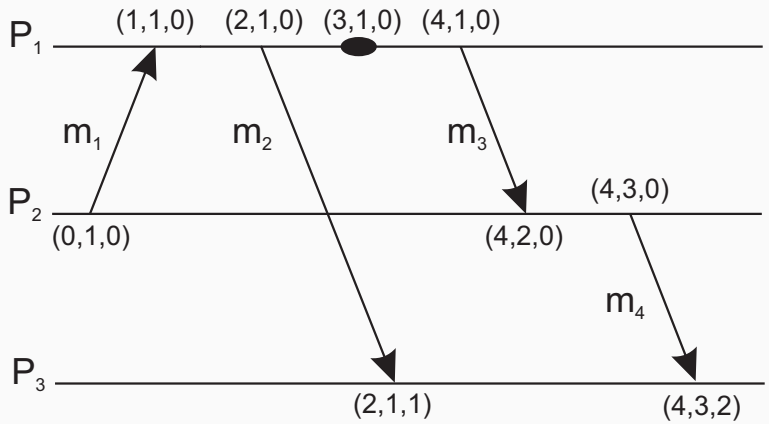
- Dizemos que $a$ precede causalmente $b$
- $b$ pode depender causalmente de $a$, já que há informação de $a$ que pode ter sido propagada para $b$

**Solução: cada $P_i$ mantém um vetor $VC_i$**

- $VC_i[i]$ é o relógio lógico local do processador $P_i$
- se $VC_i[j] = k$, então $P_i$ sabe que $k$ eventos ocorreram em $P_j$.

**Mantendo os relógios vetoriais**

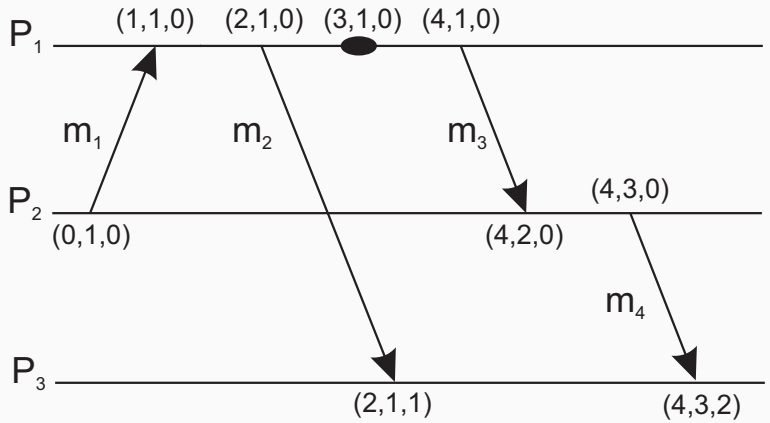
1. antes da execução de um evento, $P_i$ executa $VC_i[i] \leftarrow VC_i[i] + 1$
2. quando o processo $P_i$ enviar uma mensagem $m$ para $P_j$, ele define o *timestamp* (vetorial) de m $ts(m)$ como sendo $VC_i$ (após executar o passo 1)
3. no recebimento de uma mensagem $m$, o processo $P_j$ define $VC_j[k] \leftarrow \max\{VC_j[k], ts(m)[k]\}$

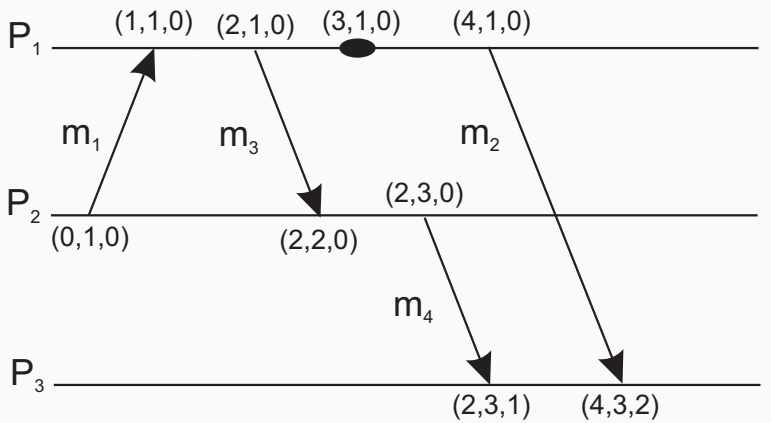



## Análise

| Situação | $ts(m_2)$ | $ts(m_4)$ | $ts(m_2) < ts(m_4)$ | $ts(m_2) > ts(m_4)$ | Conclusão |
|---|---|---|---|---|---|
| (a) | (2,1,0) | (4,3,0) | Sim | Não | $m_2$ pode preceder causalmente $m_4$ |

Suponha agora um atraso no envio de $m_2$:




## Análise

| Situação | $ts(m_2)$ | $ts(m_4)$ | $ts(m_2) < ts(m_4)$ | $ts(m_2) > ts(m_4)$ | Conclusão |
|---|---|---|---|---|---|
| (a) | (2,1,0) | (4,3,0) | Sim | Não | $m_2$ pode preceder causalmente $m_4$ |
| (b) | (4,1,0) | (2,3,0) | Não | Não | $m_2$ e $m_4$ podem conflitar |

**Observação**
Agora é possível garantir que uma mensagem seja entregue somente se todas as mensagens que as procederem por causalidade tiverem sido entregues.

**Ajuste**
$P_i$ incrementa $VC_i[i]$ somente quando enviar uma mensagem e $P_j$ "ajusta" $VC_j$ quando receber uma mensagem (mas não muda $VC_j[j]$)

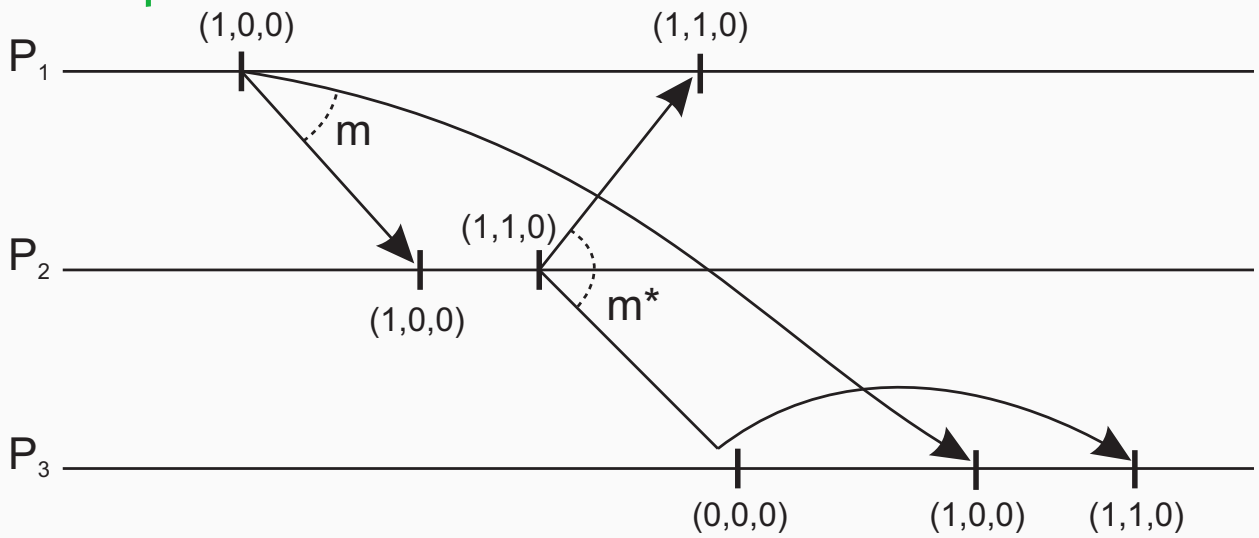
**Observação**
Agora é possível garantir que uma mensagem seja entregue somente se todas as mensagens que as procederem por causalidade tiverem sido entregues.

**Ajuste**
$P_i$ incrementa $VC_i[i]$ somente quando enviar uma mensagem e $P_j$ "ajusta" $VC_j$ quando receber uma mensagem (mas não muda $VC_j[j]$)

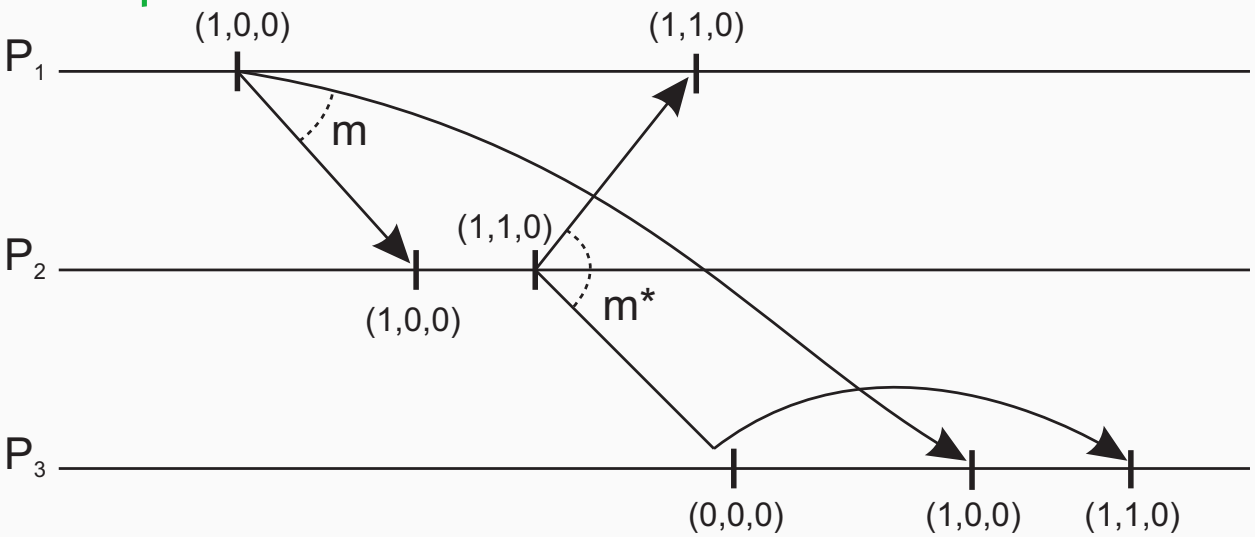
$P_j$ posterga a entrega de $m$ até que:

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

## Exemplo



## Exemplo




## Exemplo

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

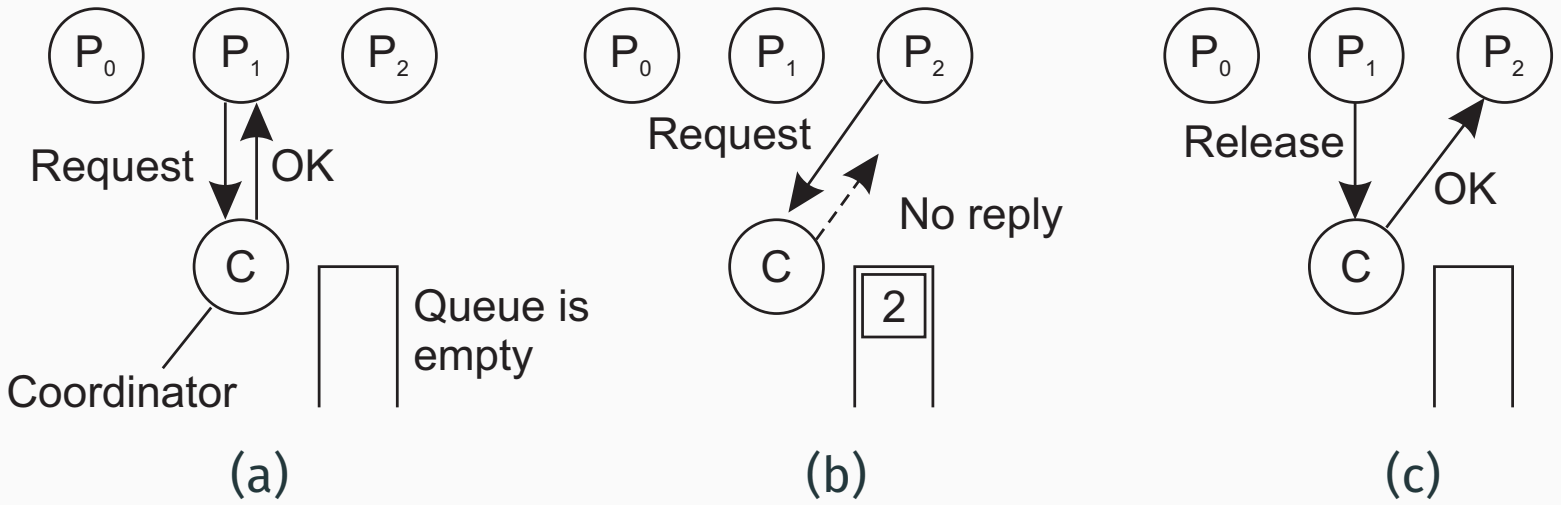
**Problema**
Alguns processos em um sistema distribuído querem acesso exclusivo a algum recurso.

**Soluções:**

**Baseado em permissão:** um processo que quiser entrar na seção crítica (ou acessar um recurso) precisa da permissão de outros processos

**Baseado em tokens:** um *token* é passado entre processos. Aquele que tiver o *token* pode entrar na seção crítica ou passá-lo para frente quando não estiver interessado.

Use um coordenador




(a) Processo $P_1$ pede permissão ao coordenador para acessar o recurso compartilhado. Permissão concedida.

(b) Processo $P_2$ então pede permissão para acessar o mesmo recurso. O coordenador não responde.

(c) Quando $P_1$ libera o recurso, avisa o coordenador, que então responde para $P_2$.

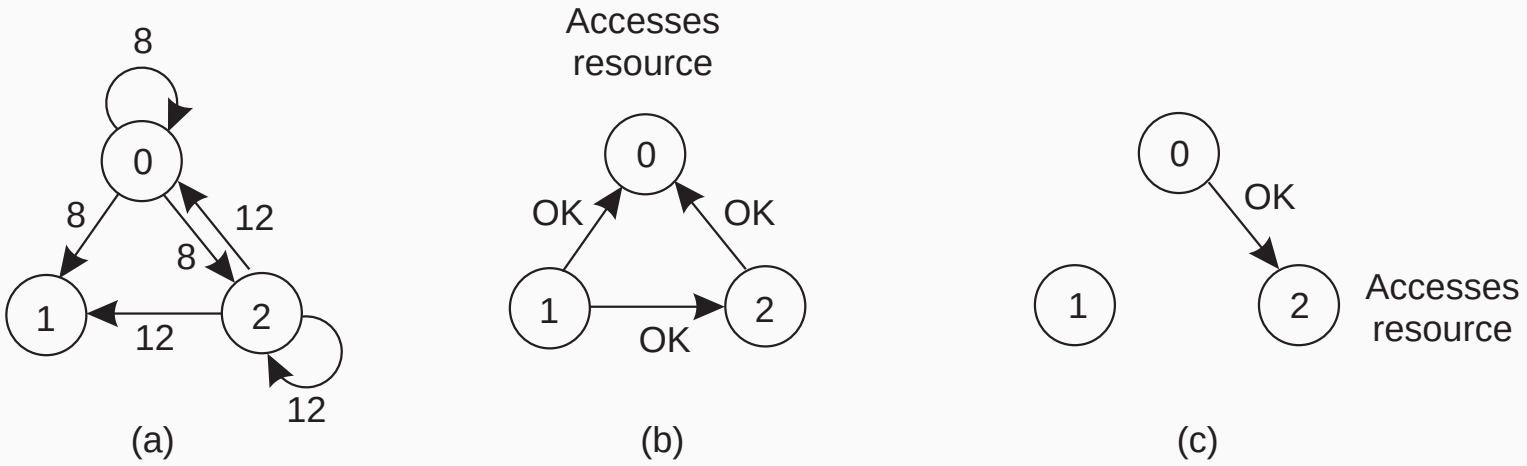
**Princípio**

Mesmo do Lamport, exceto que acks não são enviados. Ao invés disso, respostas (permissões) são enviadas quando:

- o processo receptor não tem interesse no recurso compartilhado; ou
- o processo receptor está esperando por um recurso, mas tem menos prioridade (a prioridade é determinada via comparação de timestamps)

Em todos os outros casos, o envio da resposta é adiado, implicando a necessidade de alguma administração local.

Exemplo com três processos:



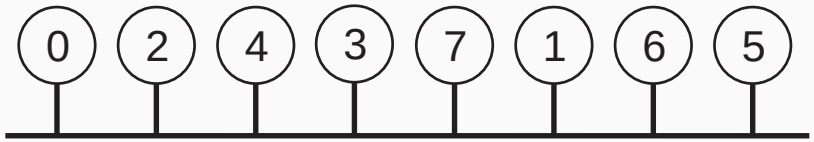

(a) dois processos querem acessar um recurso compartilhado ao mesmo tempo

(b) $P_0$ tem o menor *timestamp*; ele ganha

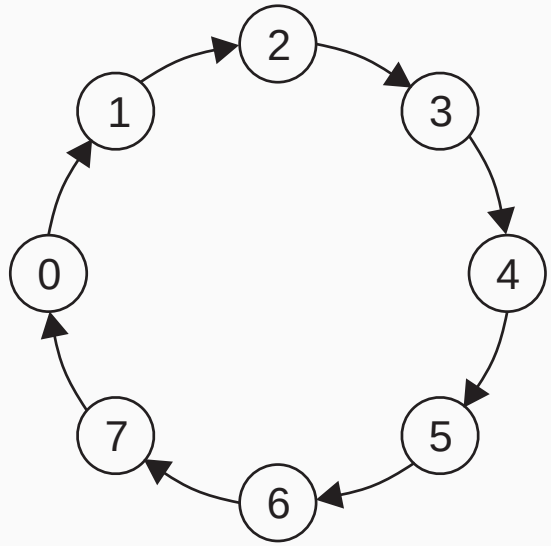
(c) quando $P_0$ terminar, também manda um OK; assim $P_2$ agora pode continuar

**Ideia**

Organizar os processos em anel lógico e passar um *token* entre eles. Aquele que estiver com o *token* pode entrar na seção crítica (se ele quiser).




(a)




(b)

**Princípio**
Assuma que todo recurso é replicado $N$ vezes, com cada réplica associada a seu próprio coordenador $\Rightarrow$ acesso requer a maioria dos votos de $m > N/2$ coordenadores. Um coordenador sempre responde imediatamente a uma requisição.

**Hipótese**
Quando um coordenador morrer, ele se recuperará rapidamente, mas terá esquecido tudo sobre as permissões que ele deu.

## Quão robusto é esse sistema?

- Seja $p = \Delta t/T$ a probabilidade de que um coordenador morra e se recupere em um período $\Delta t$ e que tenha uma esperança de vida $T$.
- A probabilidade $\mathbb{P}[k]$ de que $k$ dos $m$ coordenadores sejam resetados durante o mesmo intervalo é:

$$\mathbb{P}[k] = \binom{m}{k} p^k (1-p)^{m-k}$$

- $f$ coordenadores resetam $\Rightarrow$ corretude é violada quando os coordenadores que não falharam são minoria: quando $m - f \leq N/2$ ou $f \geq m - N/2$
- A probabilidade de violação é $\sum_{m-N/2}^{N} \mathbb{P}[k]$.

## Probabilidade de violação em função dos parâmetros

| N | m | p | Violação |
|---|---|---|---|
| 8 | 5 | 3 seg/hora | $< 10^{-15}$ |
| 8 | 6 | 3 seg/hora | $< 10^{-18}$ |
| 16 | 9 | 3 seg/hora | $< 10^{-27}$ |
| 16 | 12 | 3 seg/hora | $< 10^{-36}$ |
| 32 | 17 | 3 seg/hora | $< 10^{-52}$ |
| 32 | 24 | 3 seg/hora | $< 10^{-73}$ |

| N | m | p | Violação |
|---|---|---|---|
| 8 | 5 | 30 seg/hora | $< 10^{-10}$ |
| 8 | 6 | 30 seg/hora | $< 10^{-11}$ |
| 16 | 9 | 30 seg/hora | $< 10^{-18}$ |
| 16 | 12 | 30 seg/hora | $< 10^{-24}$ |
| 32 | 17 | 30 seg/hora | $< 10^{-35}$ |
| 32 | 24 | 30 seg/hora | $< 10^{-49}$ |

# EXCLUSÃO MÚTUA: COMPARAÇÃO

| Algorítimo | # msgs por entrada/saída | Atraso para entrar (em qde msgs) | Problemas |
|---|---|---|---|
| Centralizado | 3 | 2 | Morte do coordenador |
| Decentralizado | 2mk + m, k = 1,2,... | 2mk | *Starvation*, ineficiente. |
| Distribuído | 2 (n – 1) | 2 (n – 1) | Morte de qualquer |
| Token ring | 1 à $\infty$ | 0 à n – 1 | Perder token, proc. morrer |