

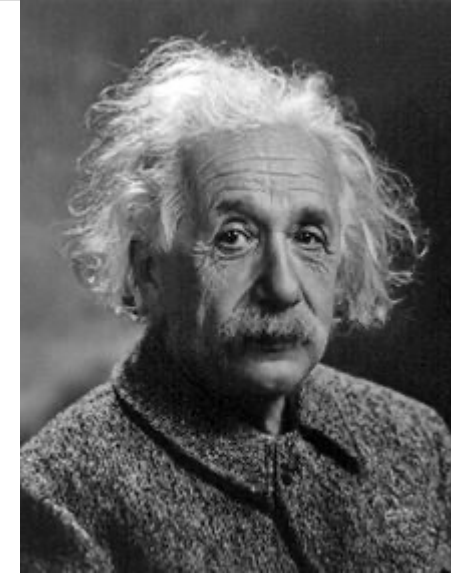
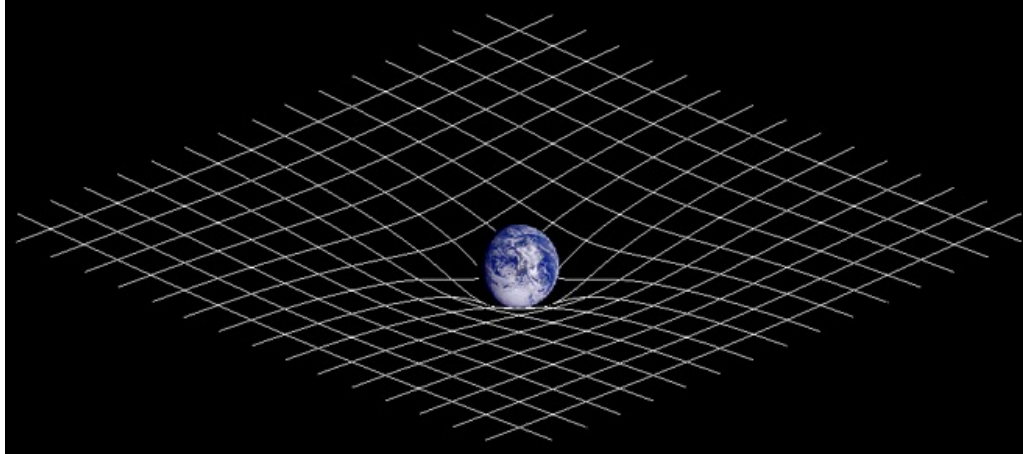


Fundamentos de sistemas distribuídos

Tempo



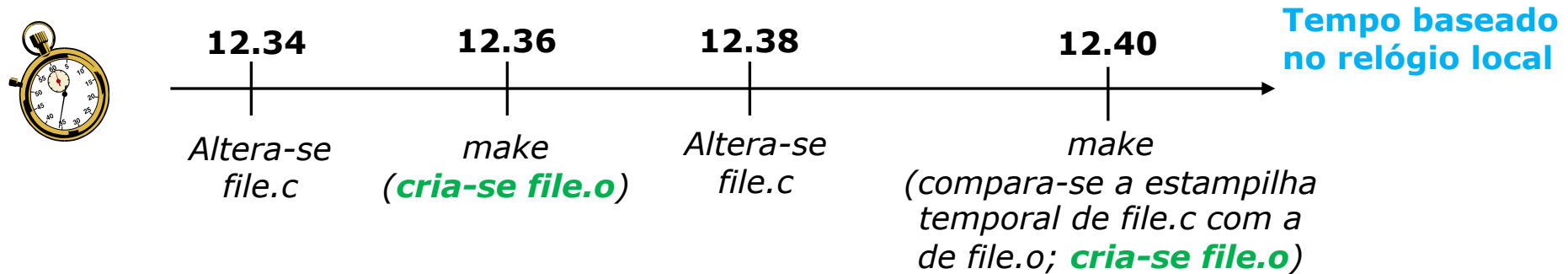
Medir o tempo não é tarefa simples



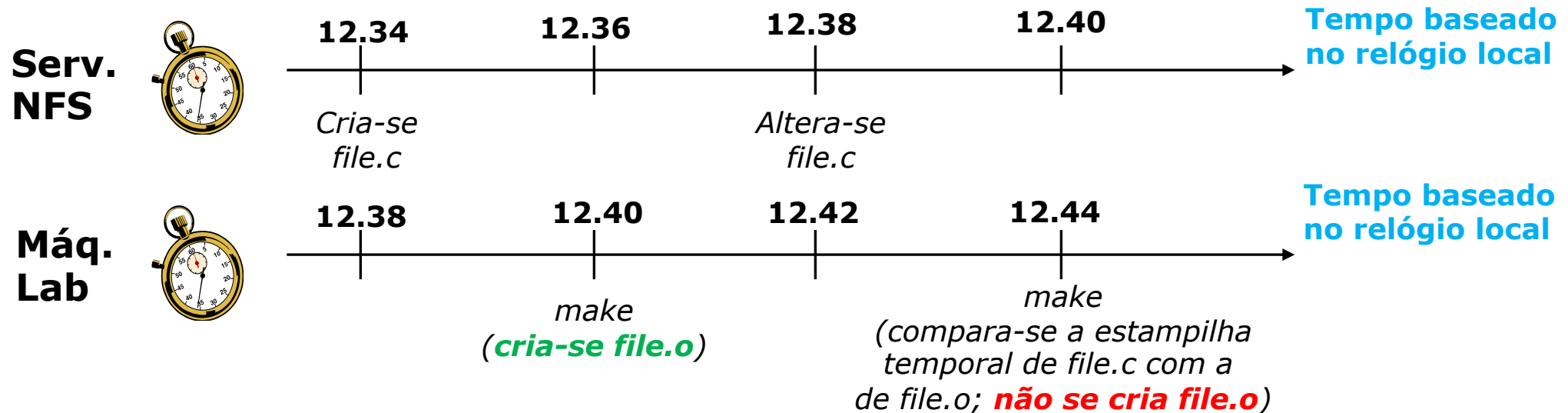
- É que o **tempo é relativo**
 - Já que a velocidade da luz é constante
- Em sistemas distribuídos a noção de tempo físico também é complicada
 - Não por causa da teoria da relatividade
 - os seus efeitos são negligenciáveis na maior parte dos contextos
 - O problema é outro: **não há um relógio global**

Motivação

- Exemplo : comando *make* num único computador



- Exemplo : comando *make* num sistema distribuído





Problema

- Dada a importância de termos relógios sincronizados, será possível **sincronizar** todos os relógios num sistema distribuído?

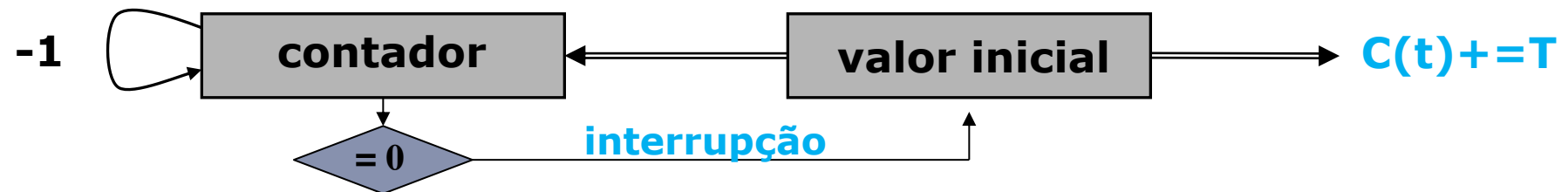


Relógios físicos



Funcionamento de um relógio físico

- Os relógios de um computador são normalmente baseados num **cristal de quartzo**, que quando submetido a uma dada tensão **oscila a uma frequência bem definida**



1. Contador é decrementado em cada oscilação do cristal
2. Quando o contador chega a 0, é gerada uma interrupção e o valor inicial é colocado no contador
3. Após cada interrupção, o valor do relógio local (denotado por **C(t)**), é incrementado num valor predefinido (**T**) de acordo com a frequência do cristal



Erro entre relógios (*clock skew*)

- Há vários factores que **afetam a frequência de oscilação**
 - tipo de cristal, corte do cristal, valor da tensão, temperatura
- Num sistema distribuído os cristais podem ter frequências ligeiramente diferentes e assim os relógios deixam de estar sincronizados
- Ao erro de sincronização entre computadores chama-se ***clock skew***



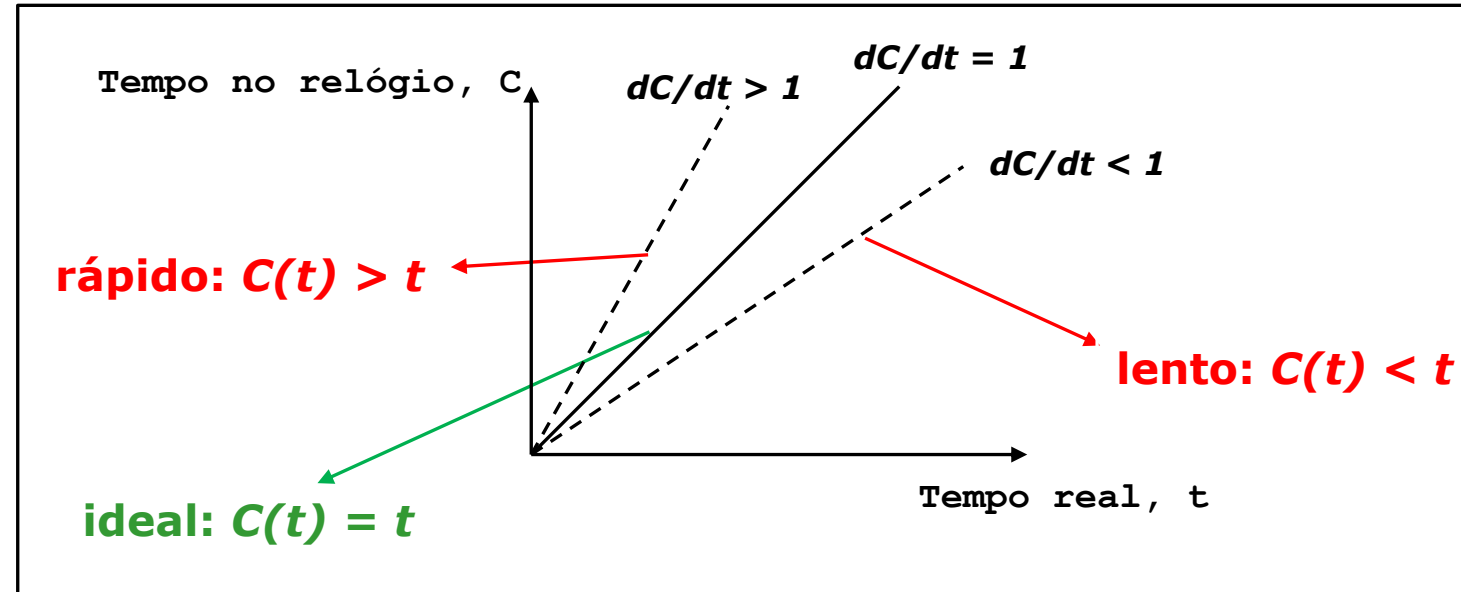
Taxa de deriva (*drift rate*)

- Pretendemos que os processos cheguem a um **acordo** em relação ao tempo nos seus relógios, e eventualmente também que esse tempo seja **próximo** do tempo real (tempo medido num relógio de referência “perfeito”)

Especificação do fabricante:

$$1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho$$

drift máximo



- Problema: os relógios físicos têm taxas de deriva (*drift rates*) diferentes
 - Para garantir um erro entre relógios (*clock skew*) menor que δ , é preciso resincronizar os relógios periodicamente com período **$P < \delta / 2\rho$**

- Considere o comportamento de dois computadores num sistema distribuído. Ambos têm relógios com cadência especificada de 1000 vezes por milisegundo, mas na realidade um deles tem uma cadência inferior, de apenas 990 vezes por milisegundo.
 - Se as actualizações do relógio ocorrerem uma vez por minuto, qual é o erro entre relógios (*clock skew*) que vai ocorrer?

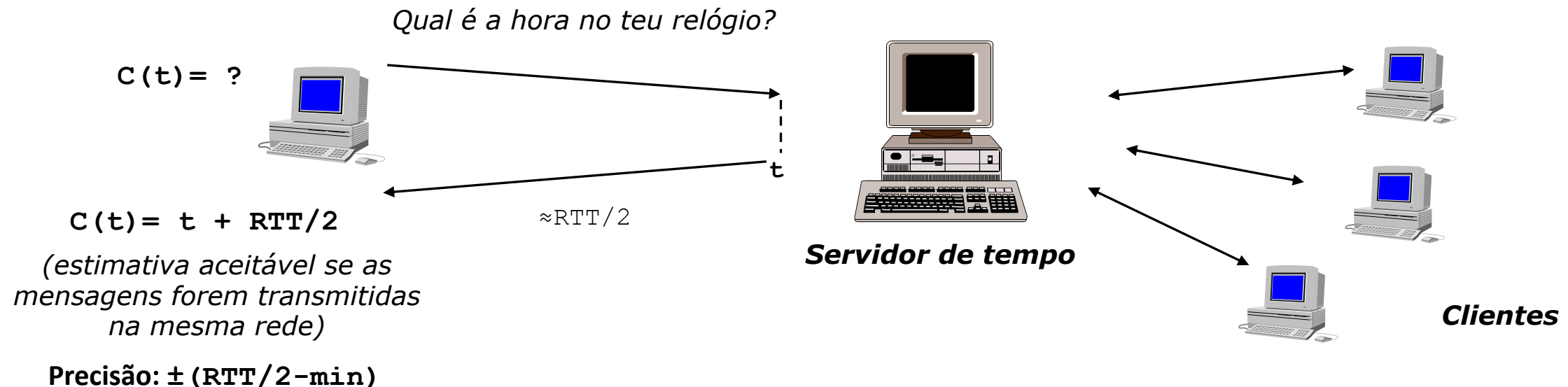


Algoritmos de sincronização de relógios

- Sincronização
 - **Externa:** relógios dos processos são sincronizados através de uma **referência externa**
 - **Interna:** relógios dos processos de um sistema **sincronizam-se entre si**
- Algoritmos
 1. Algoritmo de Cristian
 2. Algoritmo de Berkeley
 3. Network Time Protocol (NTP)

Algoritmo de Cristian

- Relógios dos clientes sincronizados pelo relógio de um **servidor de tempo** (**sincronização externa**) [Cristian1989]
 - Normalmente com acesso a um relógio muito preciso (e.g., usando GPS)
- O atraso do envio da mensagem pela rede vai influenciar o valor reportado
 - É necessário ter uma boa estimativa desse atraso.
- Algoritmo probabilístico: só existe sincronização se o RTT entre cliente e servidor for pequeno relativamente à precisão necessária



refletir.com

- Um cliente tenta sincronizar-se com um servidor. Para tal, guarda os RTT e os tempos enviados pelo servidor, de acordo com a tabela abaixo.

<i>Round-trip (ms)</i>	<i>Time (hr:min:sec)</i>
22	10:54:23.674
25	10:54:25.450
20	10:54:28.342

- Com qual destes valores é que o servidor deve acertar o seu relógio de modo a conseguir a melhor precisão?
- Qual o valor com que o deve acertar?
- Qual a precisão desse acerto?
- E se soubermos que o tempo de envio de uma mensagem é no mínimo de 8ms, essa precisão é alterada? Se sim, qual o novo valor?

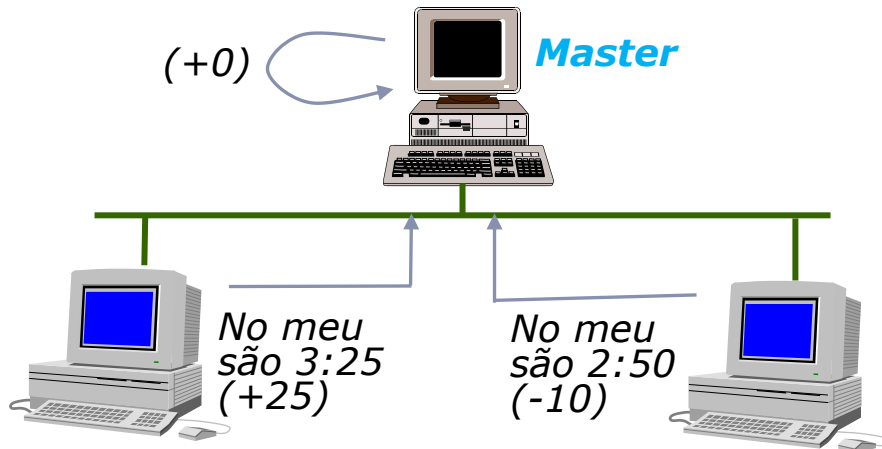
Algoritmo de Berkeley [Gusella1989]

- Sincronização dos relógios é feita de forma **distribuída** entre as várias máquinas
 - evitando a necessidade de um relógio de grande precisão disponível
- Periodicamente o *master* pergunta aos outros computadores qual o valor dos seus relógios (*polling*)

No meu relógio são 3:00.
Qual o valor do teu relógio?

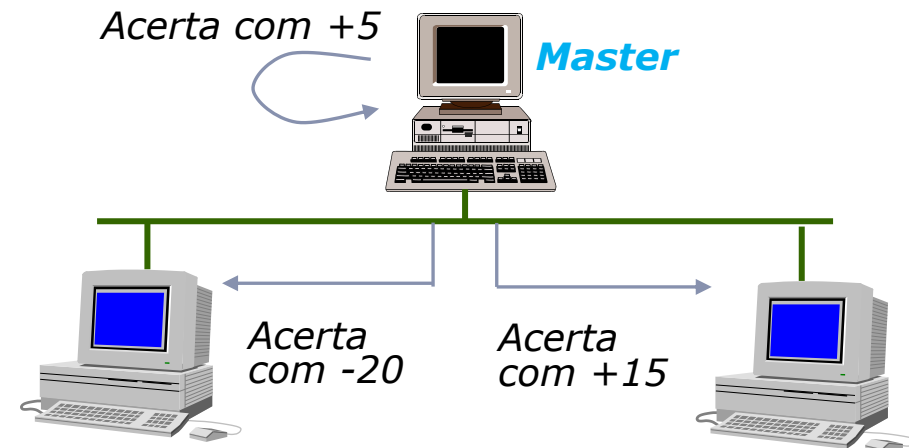


(+0) **Master**



Média = 3:05
(+5)

Acerta com +5 **Master**



- Considere uma rede em que três máquinas, A, B e C, sincronizam os seus relógios periodicamente utilizando o algoritmo de Berkeley. Se o servidor de tempo de A perguntar a todas as máquinas o valor de seus relógios locais (enviando a sua hora, 13:15:15) e receber como respostas 13:15:05 e 13:16:07, de B e C, respectivamente, qual o acerto dos relógios que o *master* enviará a cada uma das máquinas?

A = 13:15:15

B = 13:15:05

C = 13:16:07



3. NTP: *Network Time Protocol* [Mills1995]

- Algoritmo usado para sincronização de relógios **na Internet**
 - Para mitigar os atrasos existentes nesta rede, que podem ser grandes e variáveis, usa técnicas estatísticas para **filtrar** os dados e **discrimina** servidores.
 - Consegue uma precisão numa gama entre 1 e 50 milissegundos
- Três modos de funcionamento
 - **Multicast**, para usar em LANs de alta velocidade
 - Servidores enviam, por multicast, o tempo aos outros computadores que assim acertam os seus relógios (assumindo pequeno atraso na rede)
 - **Chamada a procedimento**
 - Operação **similar** ao algoritmo de **Cristian** (ver próximo slide para detalhes)
 - Maior precisão do que modo *multicast*
 - **Simétrico**
 - Usado para os servidores que necessitam ainda de **maior precisão**
 - Mantém-se **associação** entre servidores que é mantida para a precisão ir melhorando com o passar do tempo



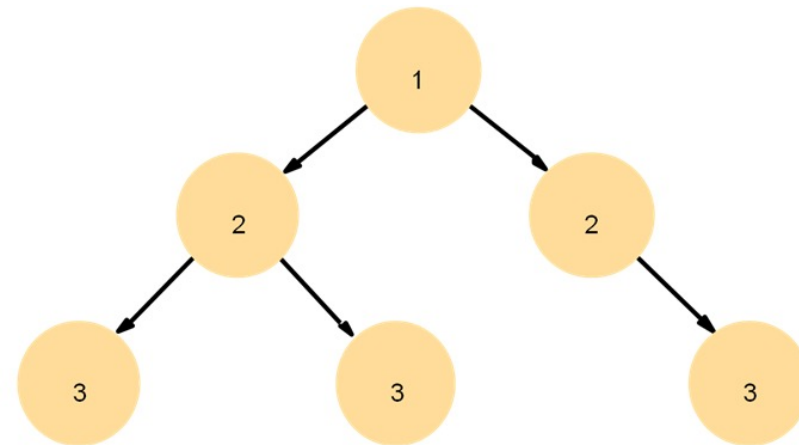
NTP: modo de chamada a procedimento

- Cada par de máquinas A e B aplica o algoritmo de **Cristian** simetricamente
- Cada máquina **armazena os últimos 8 pares** ($\vartheta = \text{offset}$, $\delta = \text{atraso}$) calculados
- Cada máquina **escolhe o par** (ϑ, δ) tal que **δ seja o menor atraso** de entre os 8 pares armazenados, e acerta o seu relógio com $C += \vartheta$
- **Problema:** se aplicarmos esta ideia “indiscriminadamente” pela rede pode ocorrer uma máquina com um relógio mais preciso se acertar com uma máquina de relógio menos preciso (o que não seria bom)



NTP: divisão em níveis

- Para evitar este problema, as máquinas são **divididas em níveis** (*stratum*):
- Máquina com relógio atómico = *stratum* 1 (o melhor)
- Máquina A só sincroniza o seu relógio com máquina B se estiver num *stratum* de número superior (i.e., o relógio de A é menos preciso)
- Após a sincronização:
 - **$\text{stratum}(A) = \text{stratum}(B) + 1$**





Relógios lógicos



Relógios lógicos

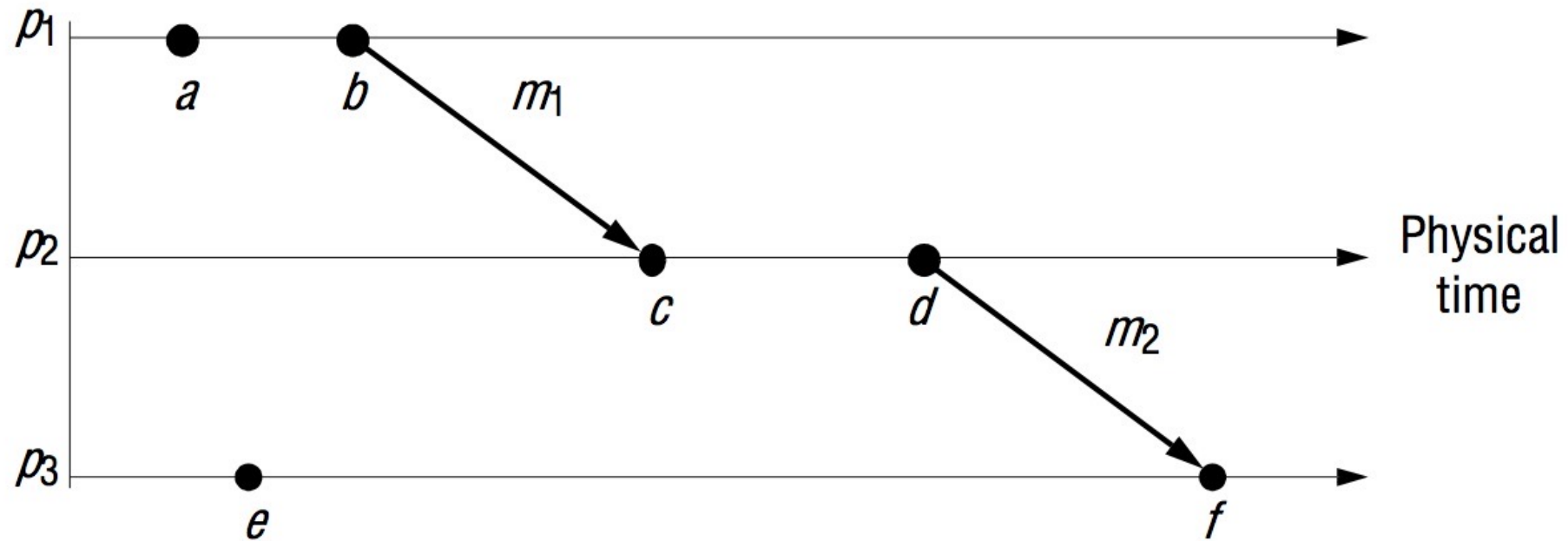
- Para muitas aplicações é suficiente que os processos cheguem a um acordo em relação à **ordem** com que ocorrem certos eventos
 - Não é necessário os processos chegarem a acordo sobre o tempo “real”
 - Mas podemos usar um esquema de **causalidade** semelhante
 - E isso permite-nos construir um **relógio lógico**.
- Relação ***happens-before/aconteceu-antes*** (\rightarrow) [Lamport1978]
 1. se ***a*** e ***b*** são eventos do mesmo processo, e ***a*** ocorre antes de ***b***, então ***a*** \rightarrow ***b***
 2. se ***a*** indica um evento *envio de mensagem*, e ***b*** o evento da *recepção dessa mesma mensagem*, então ***a*** \rightarrow ***b***

Propriedades:

Transitividade : se ***a*** \rightarrow ***b*** e ***b*** \rightarrow ***c*** então ***a*** \rightarrow ***c***

Eventos concorrentes: sem nem ***a*** \rightarrow ***b***, nem ***b*** \rightarrow ***a***, então ***a*** \parallel ***b***

Exemplos de causalidade

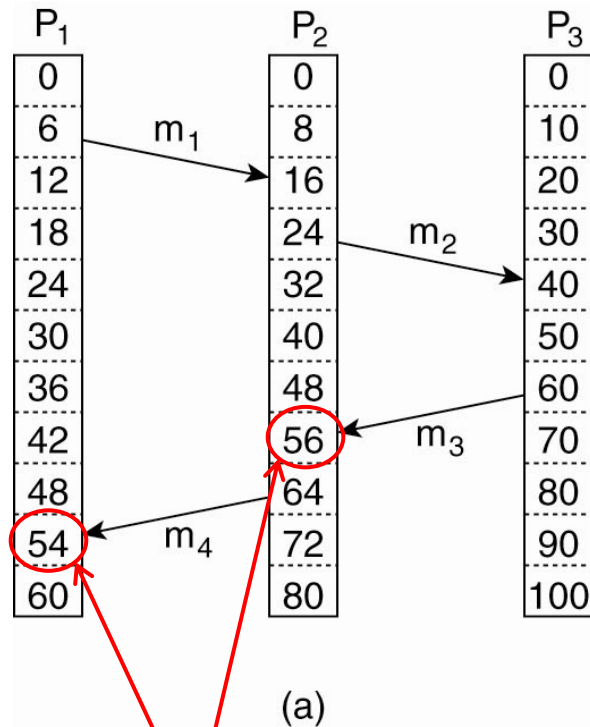

 $a \rightarrow b$
 $c \rightarrow d$
 $b \rightarrow c$
 $a \rightarrow f$
 $a \parallel e$



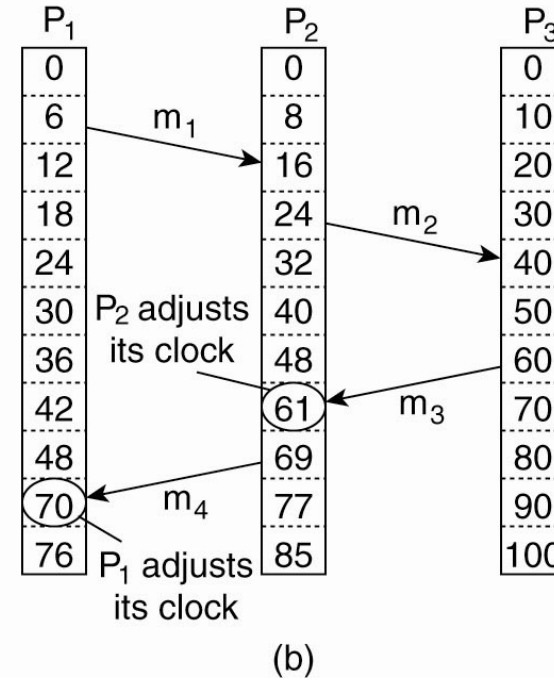
Relógios lógicos de Lamport [Lamport1978]

- Mecanismo para expressar relação *aconteceu-antes numericamente*
 1. se $a \rightarrow b$ então $C(a) < C(b)$
 - se os eventos ocorrerem *no mesmo processo*, e a ocorre *antes* de b , então $C(a) < C(b)$
 - se a for o evento *envio* de mensagem e b a sua *recepção*, então $C(a) < C(b)$
 2. o valor de $C(e)$ *nunca decresce*
 - As correções ao relógio devem ser feitas sempre por incrementos

Relógio de *Lamport*: ideia-base



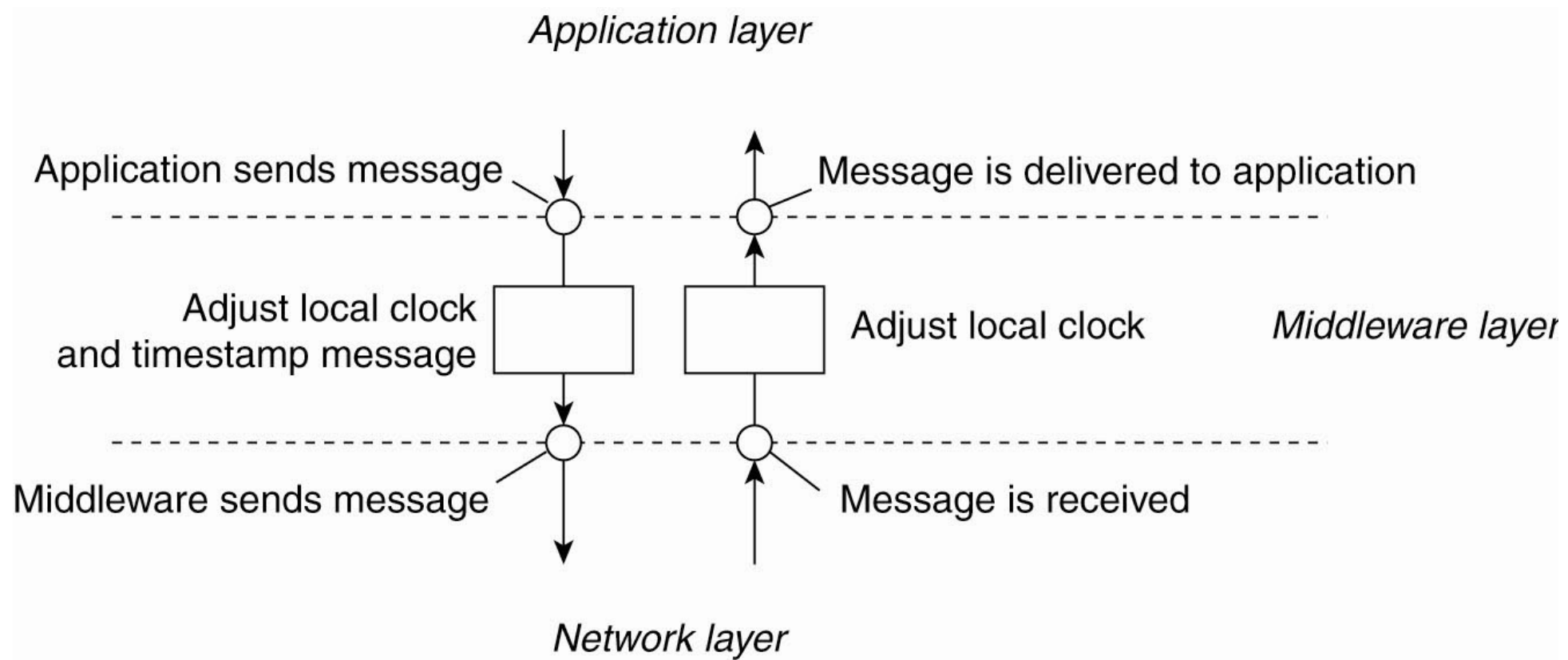
Impossível!



"Correção" de Lamport

Posicionamento do relógio lógico de *Lamport*

- Os relógios lógicos são normalmente concretizados na camada de **middleware**





Implementação de relógio de *Lamport*

1. Manter um **contador** C_i em cada processo P_i que é inicializado a 0
2. Sempre que ocorre um evento interno no processo (por exemplo, quando envia uma mensagem), **incrementar o contador**

$$C_i \leftarrow C_i + 1$$

3. Quando P_i transmite mensagem, **envia-se** junto com os dados o valor actual do contador C_i
4. Quando P_j **recebe** uma mensagem, incrementa o contador segundo a fórmula:

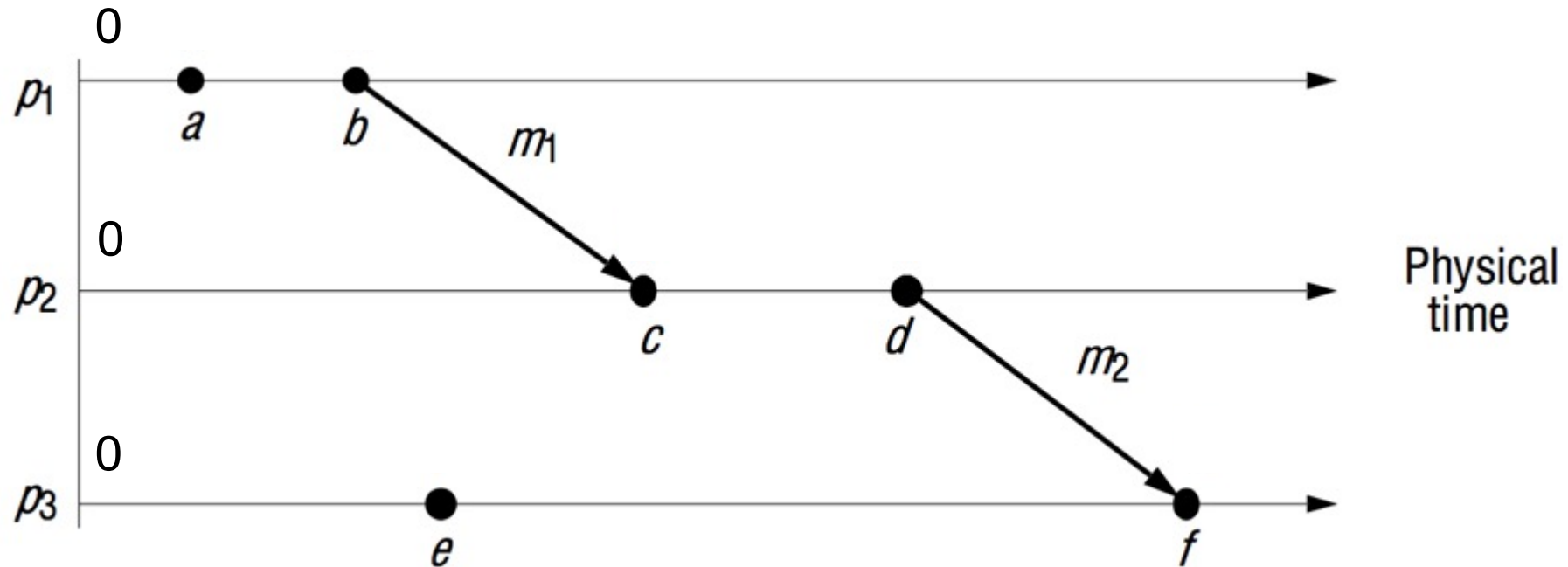
$$C_j \leftarrow \max\{C_j, C_i\} + 1$$

Recordar o que se pretendia do relógio C:

1. se $a \rightarrow b$ então $C(a) < C(b)$
2. o valor de $C(e)$ nunca decresce



Relógio lógico de Lamport: exemplo



$x \rightarrow y$ então $C(x) < C(y)$

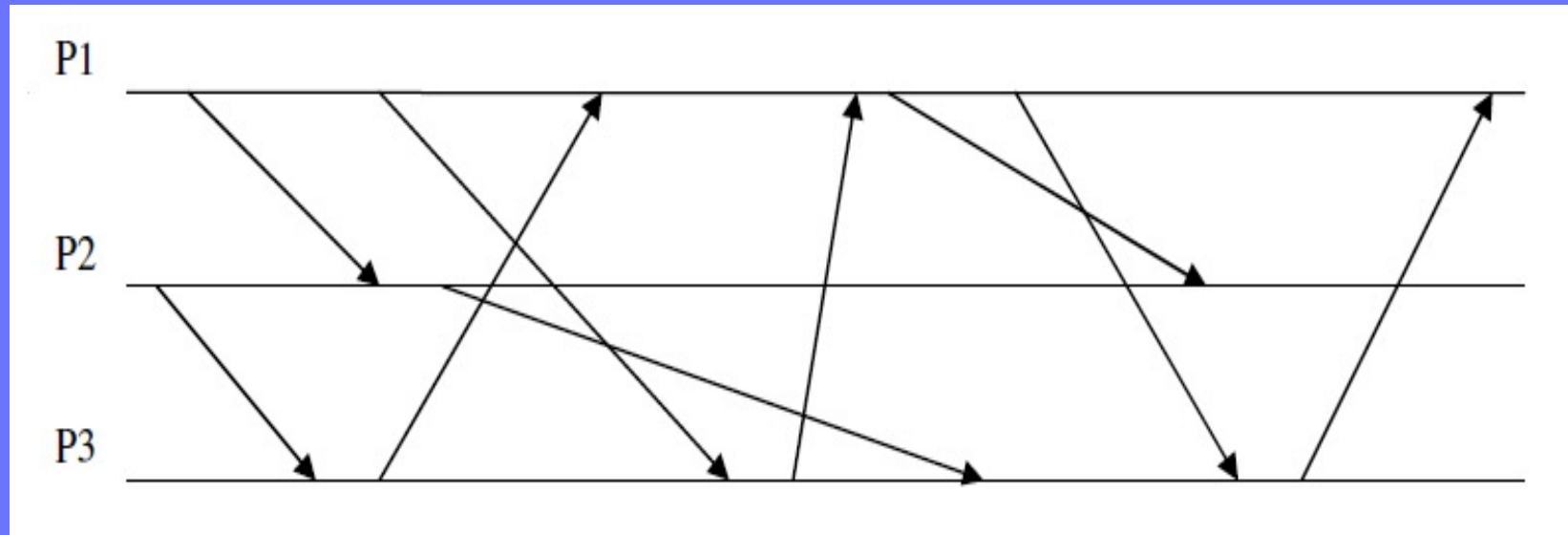


$C(x) < C(y)$ então $x \rightarrow y$



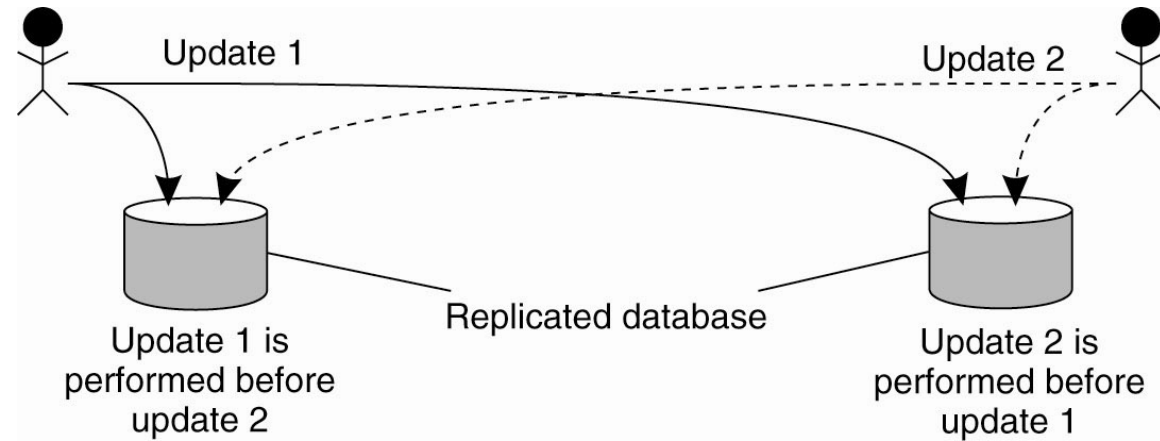
(Dois exemplos: $e \parallel b$, $e \parallel c$)

- Observe o seguinte diagrama temporal onde está representada a execução de três processos P1, P2 e P3. Pretende-se ordenar os eventos de envio e recepção de mensagens usando relógios lógicos de Lamport. Assuma que os relógios locais são inicializados a zero. Represente na figura o valor do relógio associado a cada um dos eventos de envio e recepção.



Exemplo de aplicação de relógios de Lamport

Problema: **incoerência** entre réplicas



- Exemplo:
 - Base da dados contém conta bancária com €1000
 - Update 1 = “Deposit €100” e Update 2 = “Increment 1%”
 - Estado da base de dados final incoerente: €1111 numa, €1110 noutra
- Como resolver o problema da coerência?
 - Fazer com que todas as réplicas processem a **mesma sequência** de operações, na **mesma ordem** (**difusão totalmente ordenada** – *totally-ordered multicast*)



Difusão totalmente ordenada com relógios de Lamport

Pressupostos

- Cada mensagem é estampilhada com o **tempo lógico** atual do emissor
- A transmissão é **fiável** (nenhuma mensagem é perdida) e as mensagens enviadas por um processo são recebidas **na ordem** em que foram enviadas

Algoritmo

1. Todas as mensagens são **enviadas a todos** os processos do sistema (incluindo, conceptualmente, o próprio)
2. Cada processo do sistema mantém uma **fila de mensagens ordenada pela estampa**. Quando um processo recebe uma mensagem, a mensagem é colocada nesta fila e uma **confirmação é enviada a todos** os processos do sistema.
 - Nota: no caso de os tempos lógicos serem iguais, pode usar-se ID do processo para resolver “conflito” e decidir ordem
 - Como resultado **todos os processos vão ter a mesma cópia da fila local**.
3. Uma mensagem m da fila só é processada quando a) **m está na cabeça** da fila e b) **m foi confirmada** por todos os processos do sistema


Resultado

Todos os processos acabam por remover a mesma sequência de mensagens da fila, i.e., processam as mensagens **ordenadamente**



Vector clocks

- Foram propostos [Mattern1989][Fidge1991] para resolver uma **limitação** dos relógios de Lamport:

$$C(x) < C(y), x \rightarrow y$$


- Um **vector clock** V é um vetor com N inteiros (para os N processos).
 - Cada processo mantém o seu vetor V_i e **envia-o** em todas as mensagens.



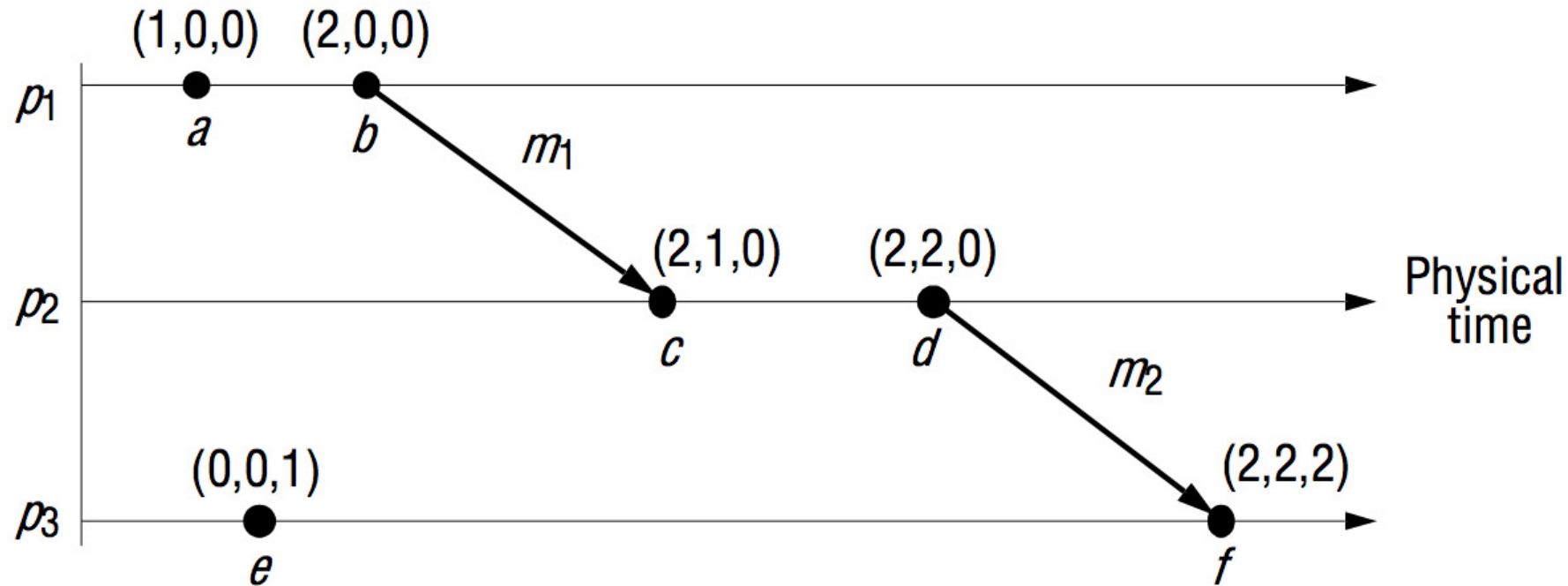
Vector clocks: implementação

1. Processo i coloca todos os elementos do seu vetor V_i a zero.
2. Sempre que ocorre um evento: $V_i[i] = V_i[i] + 1$
3. O vetor V_i é incluído ($t[j]$) em **todas** as mensagens enviadas
4. Quando um processo recebe uma mensagem, além de incrementar o seu contador, **atualiza** o seu vetor:

$$V_i[j] = \max(V_i[j], t[j]) \text{ para } j=1, 2, \dots, N$$

$V_i < V_j$ se pelo menos **um** elemento de V_i for **menor** e **nenhum** for **maior** que V_j

Vector clocks: exemplo



$x \rightarrow y$ então $V(x) < V(y)$



$V(x) < V(y)$ então $x \rightarrow y$



(Rever exemplo e || b)



Bibliografia recomendada

- [Coulouris et al]
 - Secções 14.1 a 14.4
- [van Steen, A.S. Tanenbaum]
 - Secções 6.1 e 6.2

