Ordenação

Relógios lógicos

Índice

- Ordenação
  - FIFO
  - Ordenação Causal
  - Ordenação Total
  - Algoritmos
- Tempo Lógico
  - Relógios Lógicos
  - Relógios Vectoriais

Introdução

- Ordenação
  - Objetivo 1 – Determinar à posteriori a ordem pela qual um conjunto de eventos ocorreu.

  ➔ Permite indicar quais os eventos que ocorreram primeiro e assumir, ou excluir, relações de causa efeito entre eles.

  ➔ Exemplo de aplicação: se fizermos o registo da ordem pela qual um conjunto de eventos ocorre, poderemos repetir uma computação não determinística

Introdução (cont.)

- Ordenação
  - Objetivo 2 – Garantir que um conjunto de eventos ocorra segundo uma ordem pré-determinada.

➔ Pode ser conseguido por protocolos de entrega ordenada de mensagens.

Noção de Ordem

- A noção mais intuitiva é a ordem física, em que os eventos ocorrem numa linha de tempo real.

- Esta ordem pode ser capturada se for atribuído a todos os eventos um "timestamp" com o valor de um relógio global.

Noção de Precedência ("Happened Before" – Lamport 1978)

- Se *a* e *b* são eventos no mesmo processo e *a* ocorre antes de b, dizemos que $a \rightarrow b$ *(a precede b).*
- Se *a* é o envio da mensagem *m* e *b* é a recepção da mensagem *m*, então $a \rightarrow b$.
- Se $a \rightarrow b$ e $b \rightarrow c$, então $a \rightarrow c$.

Ordem FIFO (First-In-First-Out)

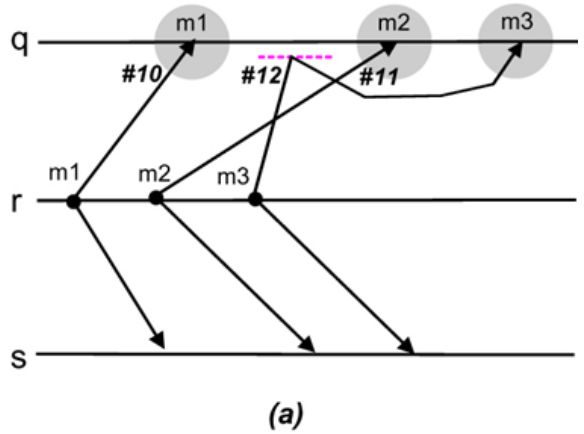
- Quaisquer duas mensagens enviadas pelo mesmo processo são entregues pela ordem de envio a qualquer outro processo.

- A ordem FIFO é assegurada pela atribuição de um número de sequência local.

- O receptor entrega as mensagens à aplicação pela ordem dos seus números de sequência.

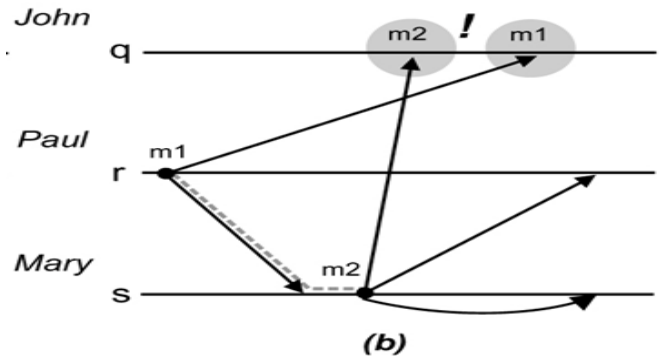
FIFO (continuação)

- Para isso o receptor deve ter um *buffer* temporário para as mensagens recebidas fora de ordem e caso exista alguma em falta pedir a retransmissão da mesma.



FIFO (continuação)

- A ordenação FIFO pode ser insuficiente.
- John está à espera de receber as mensagens por ordem de execução das tarefas.
- m1, m2 e m3 representam tarefas que deverão ser executadas em sequência
- !!!



Ordenação causal

- Consiste na garantia de que as mensagens enviadas por processos diferentes são entregues pela "ordem correcta" no receptor.

  - *Entrega Causal (Causal delivery)*

    Se

    $$envio_p(m) \rightarrow envio_q(n)$$

    Então

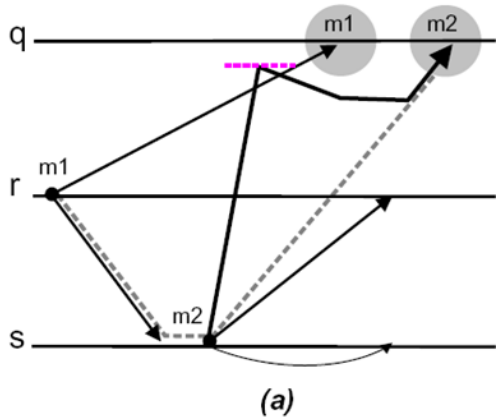
    $$entrega_r(m) \rightarrow entrega_r(n).$$

- A informação sobre a sequência lógica dos eventos é incluída nas mensagens.

Ordenação causal (continuação)

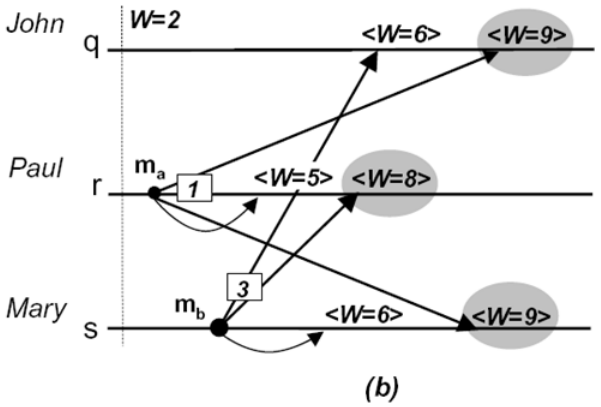
- Limitações:
  - Não garante que as mensagens dos processos concorrentes sejam ordenadas.

Ordem causal



Problema Causal

w <- max (w, v) +3



Ordenação Total

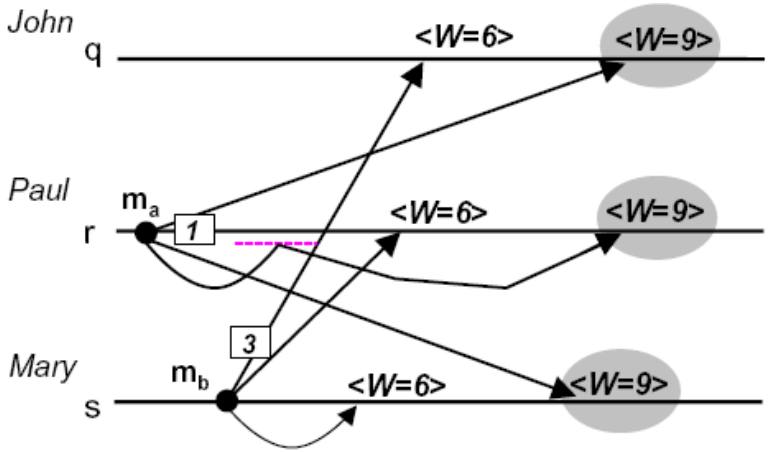
- Garante que quaisquer duas mensagens entregues a qualquer par de recetores são entregues na mesma ordem para ambos.

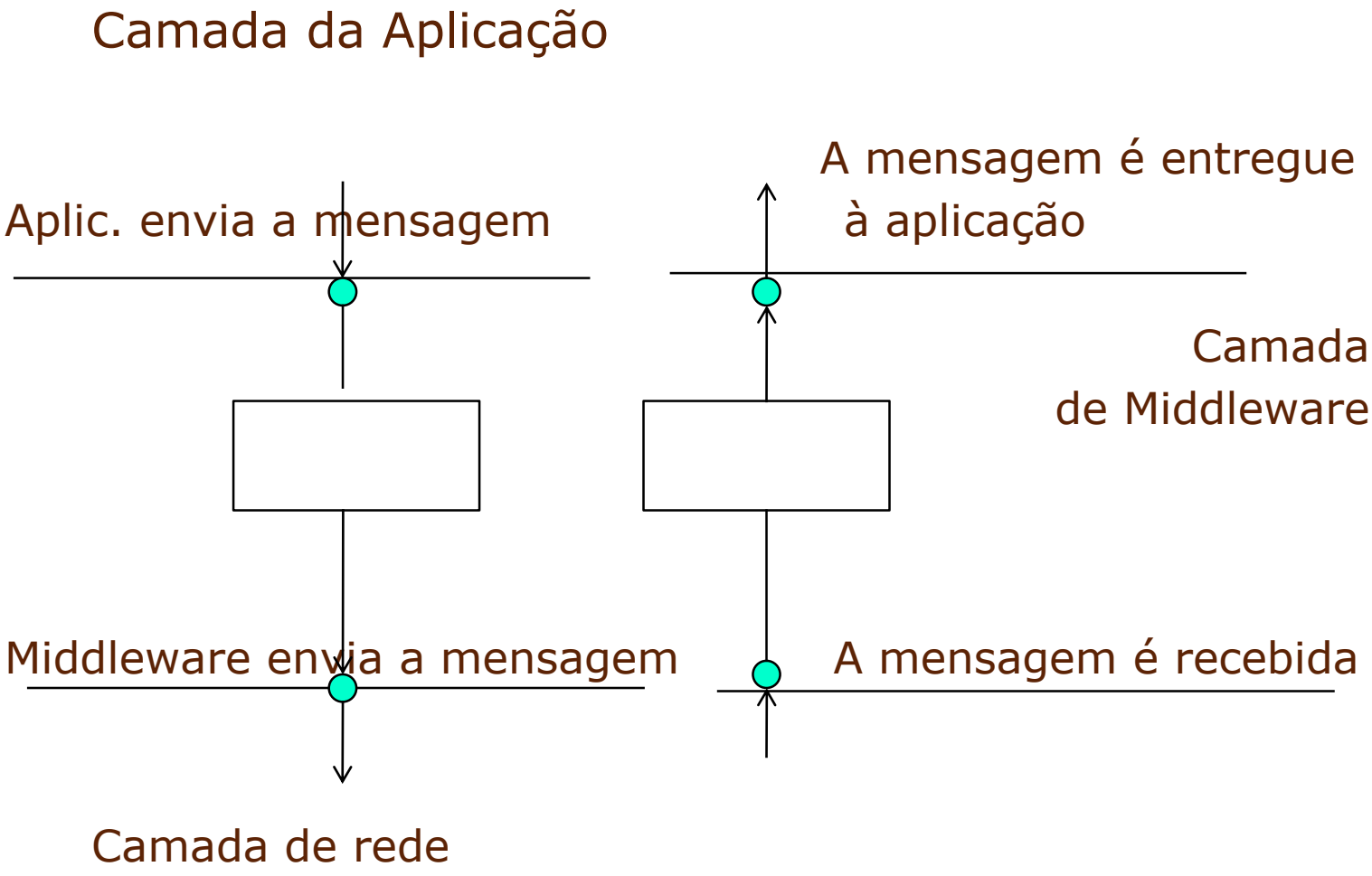
- Consiste na garantia que a entrega das mensagens se faz em todos os processos pela mesma ordem.

  - Se um processo receber m1 antes de receber m2, então qualquer outro processo que receber m2 irá receber antes m1.

Ordenação Total (continuação)

- Não implica nem a ordenação FIFO nem a Causal, a ordem de entrega pode ser arbitrária, desde que seja a mesma em todos os processos.




Camada da Aplicação
Aplic. envia a mensagem
A mensagem é entregue à aplicação
Camada de Middleware
Middleware envia a mensagem
A mensagem é recebida
Camada de rede

Algoritmos de ordenação Causal

- Objetivo:
  - Assegurar que as mensagens são entregues à aplicação de forma a respeitar a ordem causal, se m1 e m2 devem ser entregues ao mesmo processo e m1 precede m2 então m2 é entregue depois de m1.

Algoritmos de ordenação Causal (continuação)

- Uma das maneiras mais intuitivas é fazer com que cada mensagem carregue o sua própria lista “passado”.
- Para isto é necessário que cada processo mantenha uma lista de mensagens enviadas.

Algoritmos de ordenação Causal (continuação)

- Protocolo:
  - Quando uma mensagem é enviada, leva a lista "passado" do seu processo emissor no campo de controlo.
  - Depois de enviar a mensagem, o emissor adiciona essa mensagem à sua lista.
  - Quando uma mensagem é recebida é verificado o seu campo de controlo. As mensagens que se encontram nessa lista que ainda não foram entregues podem ser imediatamente entregues mesmo que não tenham sido recebidas ainda.

Algoritmos de ordenação Causal (continuação)

- Depois de essas mensagens terem sido entregues à aplicação a mensagem recebida é adicionada à lista "passado" de mensagens recebidas do receptor.

- Isto **garante** que as mensagens enviadas serão todas entregues mesmo que as anteriores se percam, pois a última carrega todas as outras como garantia.

- Um ponto negativo deste protocolo é o **tamanho** exagerado do campo de controlo, pois este pode crescer indefinidamente podendo conter muitas mensagens. O que torna este protocolo impraticável.

Algoritmos de ordenação Causal (continuação)

- Deve ser completado com um mecanismo para eliminar informação obsoleta.

- Por exemplo uma mensagem que já foi entregue a todos os recetores pode ser descartada.

Algoritmos de ordenação Causal (continuação)

- Uma forma de resolver este problema é reduzir o tamanho do campo de controlo guardando na lista "passado" apenas os identificadores das mensagens, em vez das mensagens completas.
- Nesta ideia assume-se que um 3º componente é responsável pela garantia de entrega.
- Ou seja, se uma mensagem for perdida, é de alguma forma retransmitida até ser recebida por todos os recetores pretendidos.

Algoritmos de ordenação Causal (continuação)

- Nesta forma a diferença é que quando uma mensagem é recebida e o campo de controlo verificado, se alguma mensagem ainda não tiver sido entregue a última é posta em espera até todas as outras chegarem e serem entregues.

- Só depois, a última é entregue e o seu identificador adicionado à lista “passado” do receptor.

Algoritmos de ordenação Total

- O objetivo da ordenação total é assegurar que todas as mensagens são entregues a todos os recetores pela mesma ordem.

Algoritmos de ordenação Total

- Sequenciador
  - Consiste em seleccionar um processo especial e atribuir-lhe a tarefa de ordenar todas as mensagens.
  - Todos os emissores enviam as suas mensagens para o sequenciador.
  - Que por sua vez atribui um número de sequência único a todas mensagens e posteriormente irá retransmiti-las a todos os receptores pretendidos.

  => Problema: falha do sequenciador

Tempo Lógico

- Em muitas aplicações não interessa conhecer exactamente o tempo real, mas apenas que os vários processos concordem num determinado tempo **t.**

- Apesar de ser possível sincronizar os relógios de um sistema distribuído, essa sincronização não tem de ser absoluta. Se dois processos não interagem, não é necessários que os seus relógios estejam sincronizados.

- O que muitas vezes é importante, não é qual o valor do tempo, mas que os processos concordem sobre a ordem pela qual os eventos ocorrem.

Relógios Lógicos

- Para sincronizar relógios, Lamport introduziu a noção de precedência que vimos atrás (pág. 6). Com a introdução da ideia de relógio lógico descreveu numericamente a noção de precedência

- Relógio Lógico = contador em software que implementa uma função monótona crescente.

- Cada processo, **p**, terá o seu relógio lógico, **C**, que será usado para atribuir o tempo (timestamp) em que ocorre um evento.

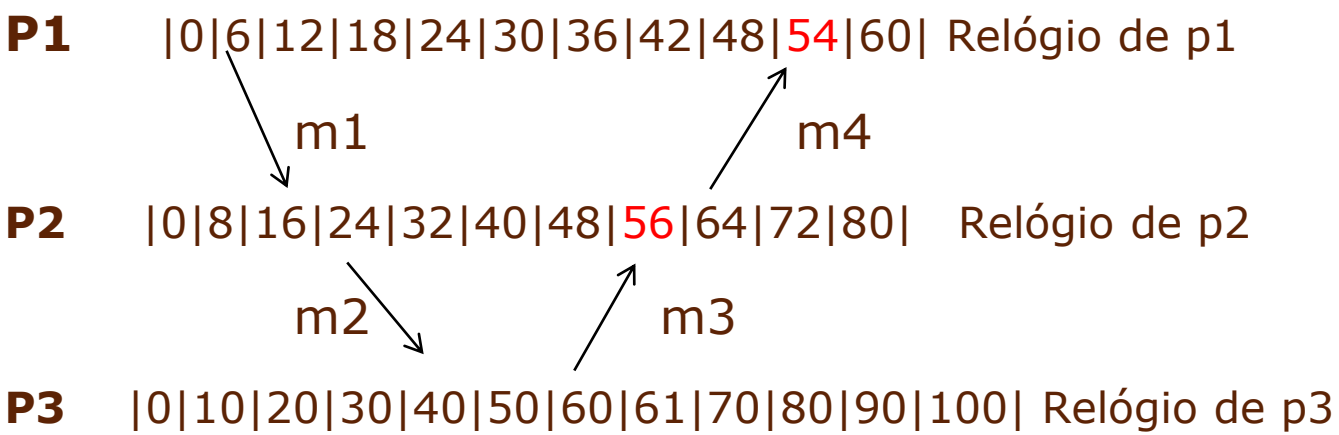
Relógios Lógicos (Lamport 1978)

- Precisamos de uma forma de medir o tempo tal que a cada evento ***a*** possa ser atribuído um tempo, ***C(a)***, com que todos os processos concordem.

- Os valores do tempo têm de ser tais que:

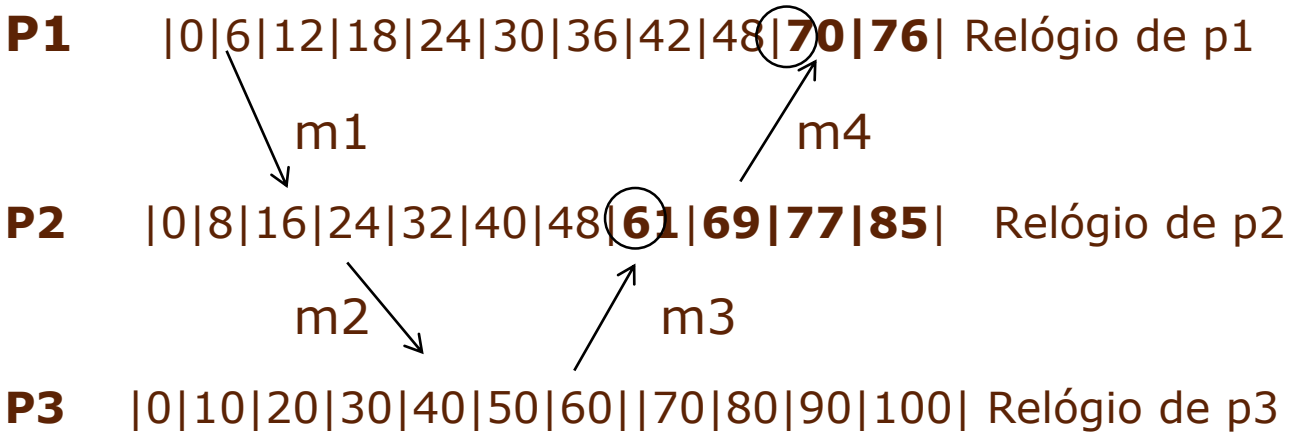
  Se a → b então C(a) < C(b)

Relógios Lógicos (Lamport 1978)



Os processos executam em máquinas diferentes, cada uma com o seu relógio, cada relógio com a sua velocidade.

Relógios Lógicos (Lamport 1978)



O algoritmo de Lamport vai corrigir os relógios, de forma a verificar-se a ordem causal. Cada mensagem transporta consigo o tempo do processo emissor

Relógios Lógicos

Algoritmo de Lamport:

Seja $\mathbf{L_i}$ o relógio lógico do processo $\mathbf{p_i}$, e $\mathbf{L_i(e)}$ o *timestamp* do evento **e** no processo $\mathbf{p_i}$.

Regra 1: $\mathbf{L_i}$ é incrementado antes da atribuição de um timestamp a qualquer evento em $\mathbf{p_i}$: $\mathbf{L_{i} := L_i + 1}$

Regra 2: a) Quando o processo $\mathbf{p_i}$ envia a mensagem **m**, anexa (*piggybacks*) a **m** o valor t = $\mathbf{L_i}$ (após o ter incrementado).

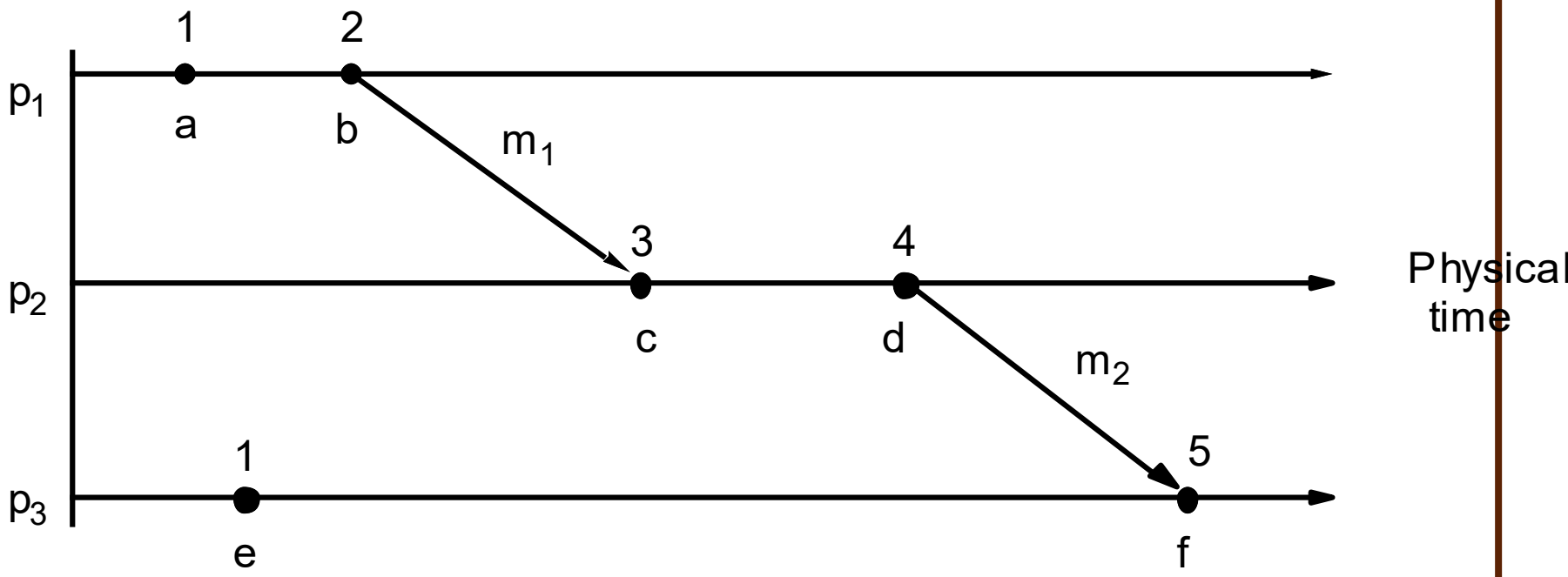
b) Ao receber uma mensagem (m,t), um processo $\mathbf{p_j}$ calcula $\mathbf{L_j}$:= max ($\mathbf{L_j}$, t), aplica a regra 1 e finalmente atribui um *timestamp* ao evento de receção da mensagem.

Relógios Lógicos

Nota: a → b implica L(a) < L(b)

se L(x) < L(Y) podemos concluir que x → Y ?

Exemplo de "Lamport timestamps":



Relógios Lógicos Totalmente Ordenados

- Na figura anterior L(a) = L(e), mas **a** e **e** são eventos distintos, gerados em processos distintos.

- Podemos criar uma ordem total para os eventos, isto é, uma ordem em que todos os pares de eventos distintos sejam ordenados, tendo em conta o identificador do processo onde ocorre o evento.

- Seja **e** um evento que ocorre em **pi**, com timestamp **Ti** e **e'** um evento que ocorre em **pj** com timestamp **Tj**, então definem-se os seus tempos lógicos globais como sendo: (Ti,i) e (Tj,j)

Relógios Lógicos Totalmente Ordenados

(Ti,i) < (Tj,j) sse

Ti < Tj

ou

Ti = Tj e i < j

Nota: Usado, por exemplo, para ordenar os processos que acedem a uma secção crítica

Relógios Vectoriais

Nos relógios lógicos totalmente ordenados, a ordem dos eventos sem relação causal é arbitrária, depende do identificador do processo.

(T(a) , pid) < (T(b), qid) não implica que a → b

Um relógio vectorial para um sistema com N processos é um array com N inteiros.

Relógios Vectoriais

Cada processo tem o seu próprio relógio vectorial Vi, que é usado para fazer o *timestamp* dos eventos locais.

Como no caso do algoritmo de Lamport, cada processo anexa às mensagens que envia, o valor do seu relógio vectorial.

Relógios Vectoriais

As regras para actualizar os relógios são:

R1: Inicialmente, Vi [j] = 0 , para j=1, 2, …, N.

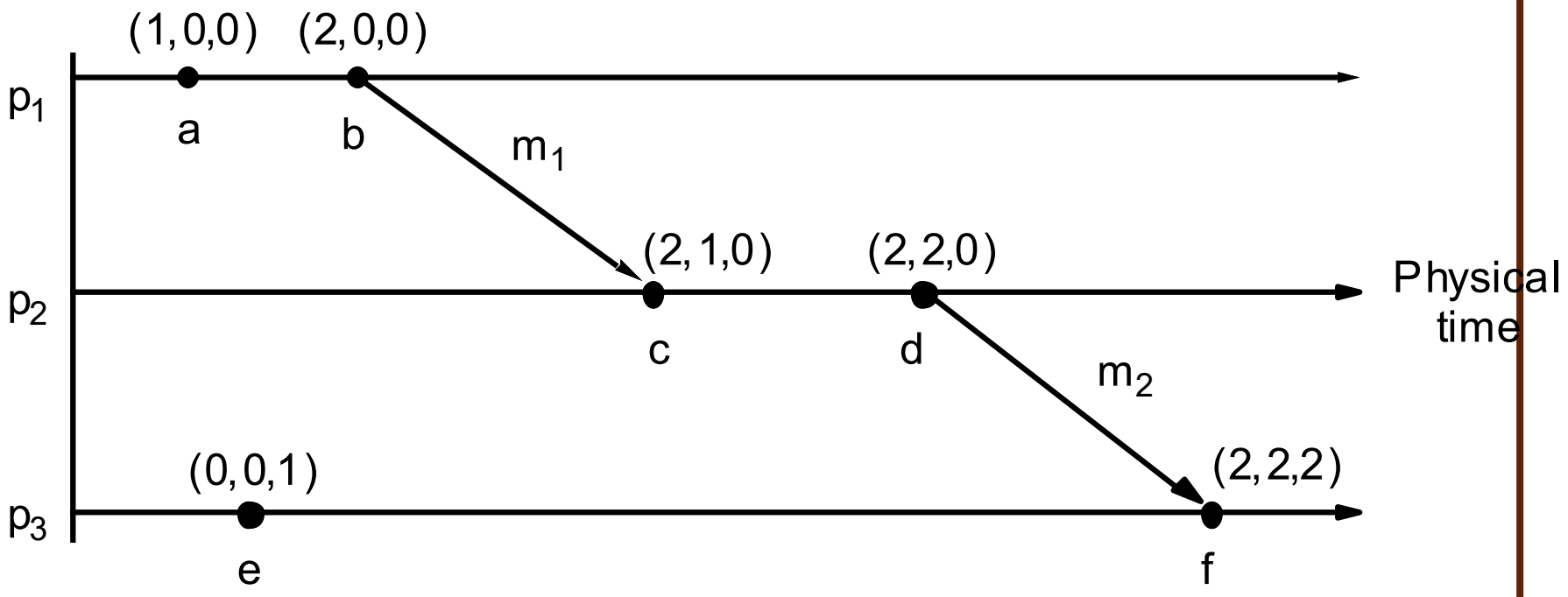
R2: Imediatamente antes de Pi atribuir um *timestamp* a um evento, incrementa o valor da posição i, Vi[i] := Vi[i] + 1.

R3: Pi inclui o valor t = Vi em cada mensagem que envia.

R4: Quando Pi recebe um timestamp t numa mensagem, faz Vi[j] := max (vi[j], t[j]), para j=1, 2, …, N. Aplica R2.

Relógios Vectoriais

Exemplo:



Relógios Vectoriais

Notas:

Para um relógio vectorial Vi:

- Vi[i] conta o número de eventos a que o processo pi atribuiu um tempo.

- Vi[j] (i !=j) é o número de eventos que ocorreram em pj que potencialmente afectaram pi.

Comparação de *timestamps*:

V=V' sse V[j] = V'[j], para j=1, 2, ..., N

V<=V' sse V[j] <= V'[j], para j=1, 2, ..., N

V<V' sse V[j] <= V'[j] ^ V!=V'

Relógios Vectoriais

Sejam e e e' dois eventos,

Se e→e' Então V(e)< V(e')

Pode demonstrar-se que

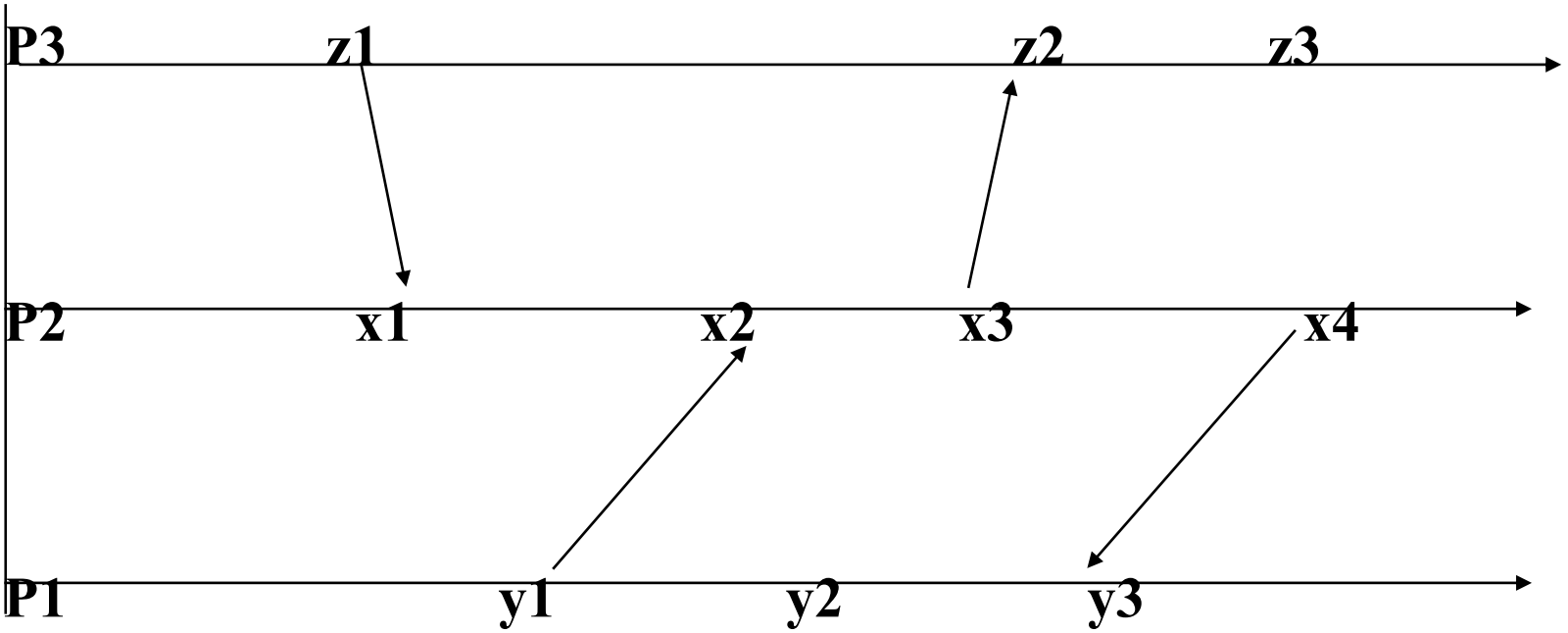
Se V(a) < V(b) Então a → b

Do exemplo anterior (pág 36),

nem V(b) < V(e) nem V(e) < V(b) portanto são eventos concorrentes b||e

Relógios Vectoriais

Exercício 1: Suponha um sistema distribuído onde ocorreu um conjunto de eventos ilustrados baixo. Atribua os respetivos *timestamps*:



Relógios Vectoriais

Exercício 2: Suponha um sistema distribuído onde ocorreu um conjunto de eventos (x, y, z, w, k) aos quais foram atribuídos os seguintes *timestamps*:

x – V(0, 1, 1)
y – V(2, 2, 2)
z – V(1, 0, 2)
w – V(0, 0, 1)
k – V(0, 0, 2)
u – V(2, 0, 2)

- Represente a sequência pela qual os eventos ocorreram na *timeline* de cada processo do sistema.