TÉCNICO LISBOA

inesc id lisboa

# *DAD*

# Tempo em Sistemas Distribuídos

# Índice

- Introdução
- Relógios, eventos, estados de processos
- Sincronização de relógios físicos

- Tempo lógico
- Relógios lógicos
- Relógios vectoriais
- Relação de causalidade *(happened-before)*

# Introdução

- **É necessário conhecer o tempo com precisão para:**
  - Saber quando um dado evento ocorreu
- **Para tal é preciso:**
  - Sincronizar um dado relógio com uma fonte fidedigna
- **Algoritmos para efectuar a sincronização são úteis pois há algoritmos/mecanismos que dependem da medição do tempo:**
  - No controle de concorrência baseado em *timestamping*
  - Nos mecanismos de autenticação (e.g. Kerberos, *single sign-on*)
- **Mas não há um relógio global num sistema distribuído**
- **Uma alternativa é utilizar tempo lógico:**
  - Permite ordenar os eventos entre si
  - Útil no âmbito da consistência de dados replicados

# Relógios Físicos

- **Cada computador tem o seu relógio interno:**
  - Usado pelos processos para obter o tempo actual
  - Processos podem associar *timestamps* a certos eventos
  - No entanto, os relógios têm desvios (*drift*) e ritmos de desvio (*drift rate*) distintos
  - Portanto, os relógios têm de ser corrigidos

# Relógios, Eventos e Estado de Processos

- **Sistema distribuído:**
  - Conjunto P de N processos $\mathbf{p_i}$, i = 1,2, ,,, N
  - Cada processo $\mathbf{p_i}$ tem um estado $\mathbf{s_i}$ que consiste nas suas variáveis
  - Processos comunicam através de mensagens trocadas na rede
  - Acções dos processos:
    - send, receive, change-state (alterar o seu próprio estado, computação com as suas variáveis)
  - Evento:
    - Ocorrência de uma acção que um processo executa
  - Os eventos em cada processo $p_i$ podem ser ordenados (ordem total):
    - Relação *happened before*
    - $e \rightarrow_i e'$ sse $e$ ocorre antes de $e'$ em $p_i$
- **História de um dado processo $\mathbf{p_i}$ :**
  - Série de eventos ordenados por $\rightarrow_i$
  - history($p_i$)= $h_i = \langle e_i^0, e_i^1, e_i^2, \ldots \rangle$

# Relógios

- **Para associar *timestamps* aos eventos:**
  - Usar o relógio do computador
  - No momento t, o sistema operativo lê o relógio hardware obtendo $H_i(t)$
  - Em seguida calcula o tempo software: $C_i(t) = \alpha H_i(t) + \beta$
    - Número com 64 bits que são os nanosegundos desde um dado momento
  - Em geral, este valor não é completamente exacto
  - Mas se $C_i$ tem um comportamento "razoável" então pode ser usado para efectuar o *timestamping* de eventos em $p_i$
  - **Razoável:** significa que o seu valor não diminui e que tem uma resolução adequada:
    - resolução < intervalo de tempo entre eventos sucessivos

# Desvio de Relógios nos Sistemas Distribuídos

- **Os relógios dos computadores não estão sincronizados**
- ***Skew*:**
  - Diferença (atraso/avanço) entre dois relógios
- **Relógios dos computadores estão sujeitos a desvio (*drift*):**
  - Contam o tempo a ritmos distintos
- **Ritmo de desvio (*drift rate*):**
  - Diferença, por unidade de tempo, em relação a um relógio de referência
- **Relógios de quartzo correntes:**
  - *Drift rate* de 1 seg em 11-12 dias ($10^{-6}$ segs/seg)
- **Relógios de quartzo de alta precisão:**
  - *Drift rate* entre $10^{-7}$ e $10^{-8}$ segs/seg

# Tempo Universal Coordenado (*Coordinated Universal Time* - UTC)

- ***International Atomic Time*** é baseado em relógio de grande precisão (*drift rate* $10^{-13}$)
- **UTC** é um standard internacional baseado em relógios atómicos e ocasionalmente ajustado de acordo com medidas astronómicas
- É difundido através de sinal rádio terrestres e a partir de satélites (e.g. GPS)
- Computadores podem ter receptores que lhes permitem sincronizar os seus relógios com o UTC
- Sinais rádio de estações terrestres têm precisão entre 0.1-10 millisegundos
- Sinais de satélite (GPS) tem precisão de cerca de 1 microsegundo

# Sincronização de Relógios Físicos

- **Sincronização externa:**
  - Relógio de computador $C_i$ é sincronizado com uma fonte externa S:
    - $|S(t) - C_i(t)| < D$ para $i = 1, 2, \ldots N$ num intervalo $I$ de tempo real
    - Os relógios $C_i$ são exactos dentro do limite $D$
- **Sincronização interna*:***
  - Os relógios de vários computadores sincronizam-se entre si:
    - $| C_i(t) - C_j(t)| < D$ para $i = 1, 2, \ldots N$ num intervalo $I$ de tempo real
    - Os relógios $C_i$ *e* $C_j$ *estão sincronizados dentro do limite* $D$
  - relógios sincronizados internamente podem não estar sincronizados externamente:
    - Pode existir drift colectivo
- **Se um conjunto de processos P está sincronizado externamente com um limite D, então esses processos estão sincronizados internamente com um limite ? :**

# Sincronização de Relógios Físicos

- **Sincronização externa:**
  - Relógio de computador $C_i$ é sincronizado com uma fonte externa S:
    - $|S(t) - C_i(t)| < D$ para $i = 1, 2, \ldots N$ num intervalo $I$ de tempo real
    - Os relógios $C_i$ são exactos dentro do limite $D$
- **Sincronização interna:**
  - Os relógios de vários computadores sincronizam-se entre si:
    - $| C_i(t) - C_j(t)| < D$ para $i = 1, 2, \ldots N$ num intervalo $I$ de tempo real
    - Os relógios $C_i$ *e* $C_j$ *estão sincronizados dentro do limite* $D$
  - relógios sincronizados internamente podem não estar sincronizados externamente:
    - Pode existir drift colectivo
- **Se um conjunto de processos P está sincronizado externamente com um limite D, então esses processos estão sincronizados internamente com um limite ? R: 2D**
  - $C_i(t) = S(t) + D$, e $C_j(t) = S(t) - D$
  - $C_i(t) - C_j(t) = 2D$

# Correcção dos Relógios

- **Um relógio hardware é correcto se o seu *drift rate* (ritmo de desvio) é limitado:**
  - $0 < \rho <$ MAX (e.g. $10^{-6}$ segs/ seg)
- **Significa que o erro de medição do intervalo entre *t* e *t'* é limitado:**
  - $(1 - \rho)\ (t' - t) \leq H(t') - H(t) \leq (1 + \rho)\ (t' - t)$ (onde $t'>t$)
  - isto impede saltos no valor do relógio
- **Condição mais fraca de monotonia (*monotonicity*):**
  - $t' > t \Rightarrow C(t') > C(t)$
  - e.g. requirida para o make (Unix)
- **Um relógio incorrecto é o que não cumpre a sua condição de correcção**
- ***Crash failure*:**
  - O relógio deixa de contar
- ***Arbitrary failure:***
  - Qualquer outra falha (e.g. saltos no tempo, voltar atrás)

# Sincronização num Sistema Distribuído Síncrono (1)

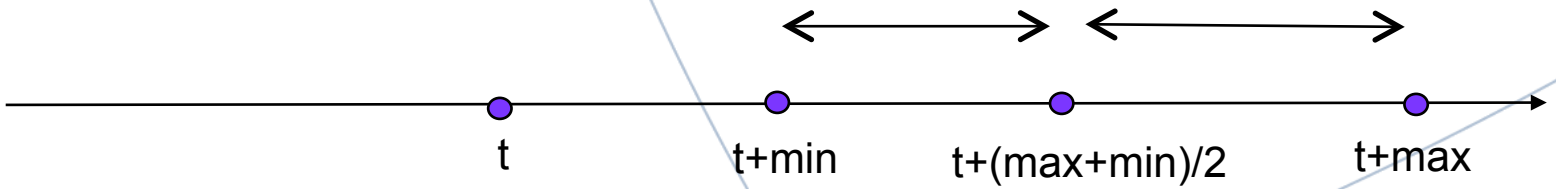
- **Sistema distribuído síncrono é aquele em que os limites seguintes estão definidos:**
  - Tempo de execução de cada passo num processo tem limites inferior e superior
  - Cada mensagem transmitida na rede é entregue dentro de um limite máximo de tempo
  - Cada processo tem um relógio local cujo desvio (drift) em relação ao tempo real tem um limite conhecido
- **Sincronização interna num sistema síncrono:**
  - O processo p1 envia o seu tempo local t para o processo p2 numa mensagem m
  - p2 ajusta o seu relógio para $t + T_{trans}$ onde $T_{trans}$ é o tempo de transmissão da mensagem m
  - $T_{trans}$ é desconhecido mas $min \leq T_{trans} \leq max$
  - Incerteza é: $u = max - min$
  - Ajustar relógio para o valor: $t + (max + min)/2$
  - A diferença entre os relógios é: $skew \leq u/2$

# Sincronização num Sistema Distribuído Síncrono (2)

- $p_2$ pode fazer $t_2 = t + max$:
  - skew pode ser = u
- $p_2$ pode fazer $t_2 = t + min$:
  - skew pode ser = u
- $p_2$ faz $t_2 = t + ((max+min)/2)$:
  - (t+max)-(t+(max+min)/2) =
  - t+max-t-max/2-min/2=
  - max/2-min/2=u/2
- $p_2$ faz $t_2 = t + ((max+min)/2)$:
  - t+((max+min)/2)-(t+min) =
  - t+max/2+min/2-t-min=
  - max/2-min/2 = u/2

A diferença entre os relógios é:

- skew ≤ u/2



# Internet

- **Mas será a Internet um sistema síncrono?**
  - Apenas podemos dizer que

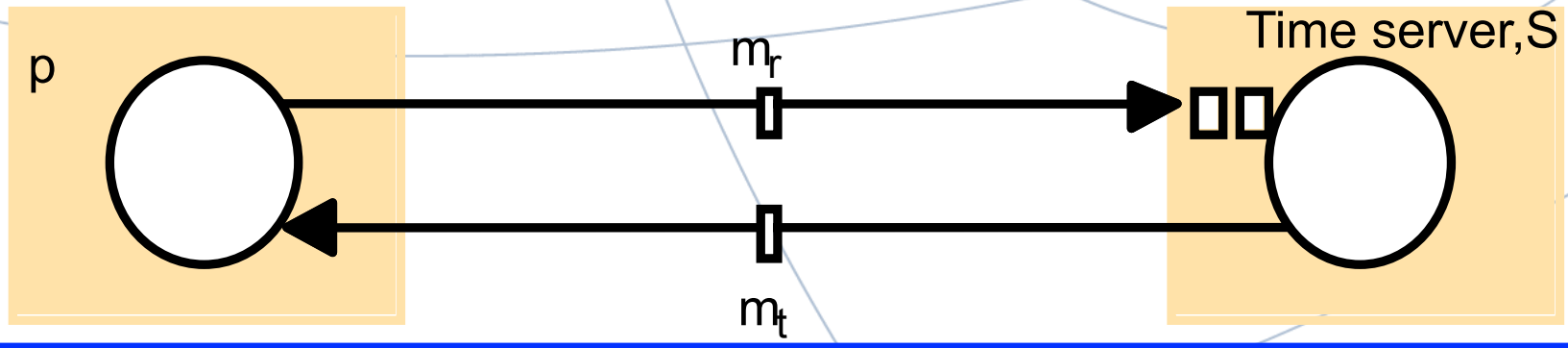
    $T_{trans} = min + x$

    onde x >= 0, sem limite fixo
- **É claramente um sistema assíncrono !!!**

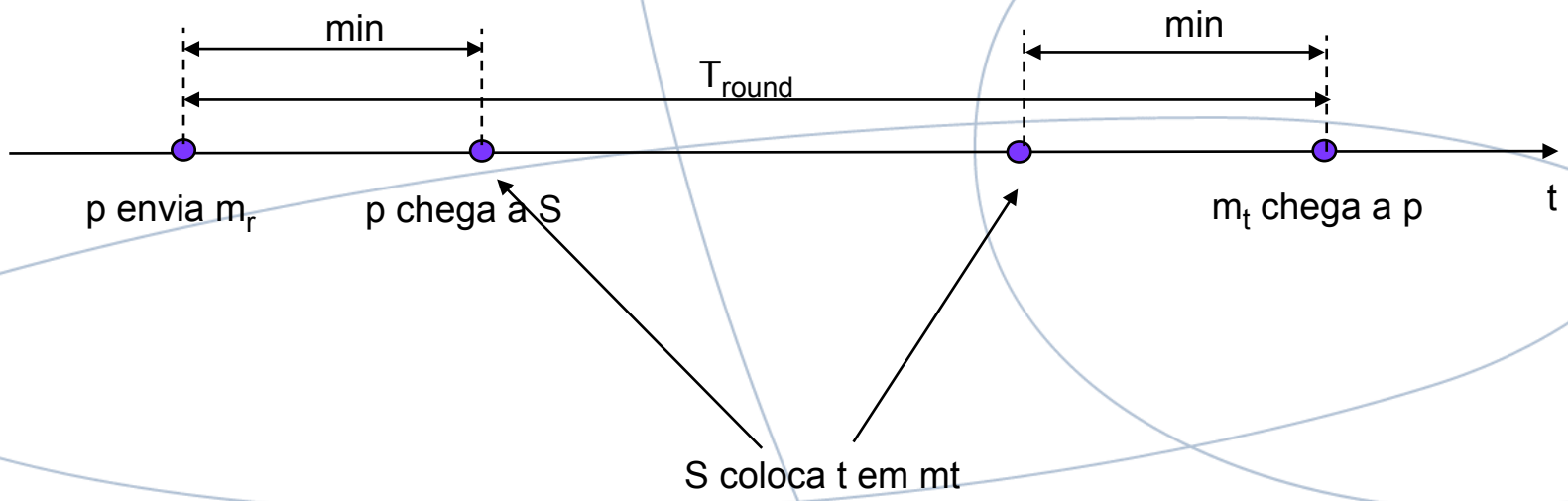
# Algoritmo de Cristian (1989) para um Sistema Distribuído Assíncrono (1)

- **Um servidor S recebe informação de uma fonte UTC:**
  - Processo *p* requisita tempo em $m_r$ e recebe *t* em $m_t$ de *S*
  - *p* ajusta o seu relógio para $t + T_{round}/2$
  - Precisão é ± ($T_{round}/2$ - *min*):
    - O instante mais recente em que S coloca *t* na messagem $m_t$ é *min depois de p enviar* $m_r$
    - O tempo mais atrasado foi *min antes de* $m_t$ *ter chegado ao processo p*
    - *O tempo segundo o relógio de S quando* $m_t$ chega a p está no intervalo [$t$+*min*, $t + T_{round}$ - *min*]

$T_{round}$ é o round trip time registado por *p*



# Algoritmo de Cristian (1989) para um Sistema Distribuído Assíncrono (2)



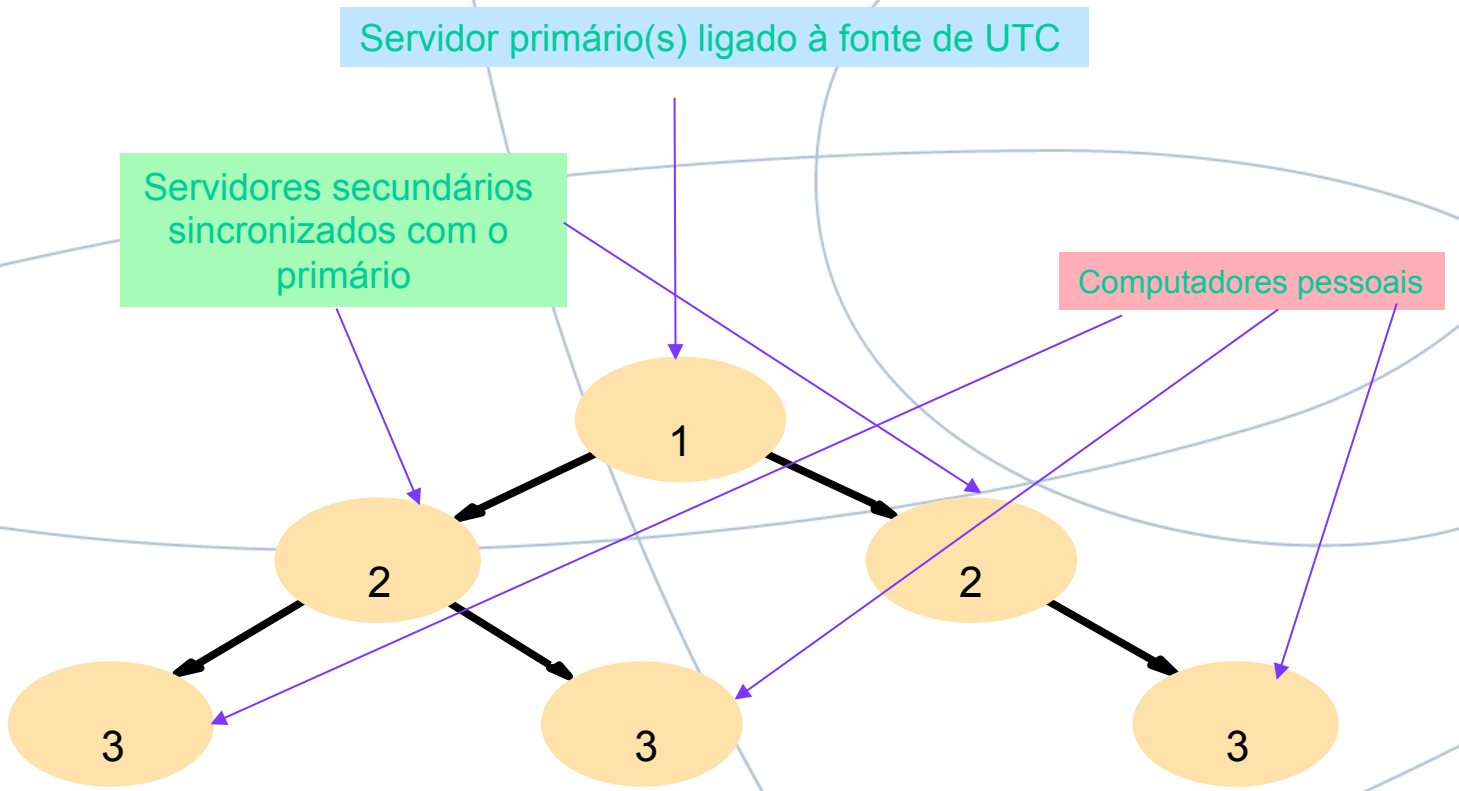

- largura do intervalo é $T_{round}$-2min
- precisão é ± ($T_{round}$/2 - *min*)
- *p* ajusta o seu relógio para $t + T_{round}/2$

# Algoritmo de Berkeley

- **Algoritmo de Cristian:**
  - Um servidor único é um ponto de falha que pode levar à indisponibilidade do serviço
  - A solução consiste em usar um grupo de servidores
- **Algoritmo de Berkeley (1989):**
  - Permite fazer a sincronização interna de um grupo de computadores
  - O servidor *master:*
    - faz poll aos restantes (slaves) para obter os respectivos valor dos relógios
    - usa round trip times para estimar o valor dos relógios dos processos slaves
    - Obtém a média dos valores obtidos
    - Envia os ajustes requeridos para cada um dos slaves
      - notar que não envia o tempo para acerto do relógio pois isso dependeria do round trip time. **Q: Qual o problema que surgiria?**
  - Exemplo:
    - 15 computadores, drift rate < $2 \times 10^{-5}$
    - Se o master falha é preciso eleger um novo

# Network Time Protocol (NTP)

- **Serviço de tempo para a Internet:**
  - Permite que os seus clientes sincronizem os relógios de acordo com o UTC
  - Robustez e escalabilidade resultante da redundância nos caminhos de disseminação

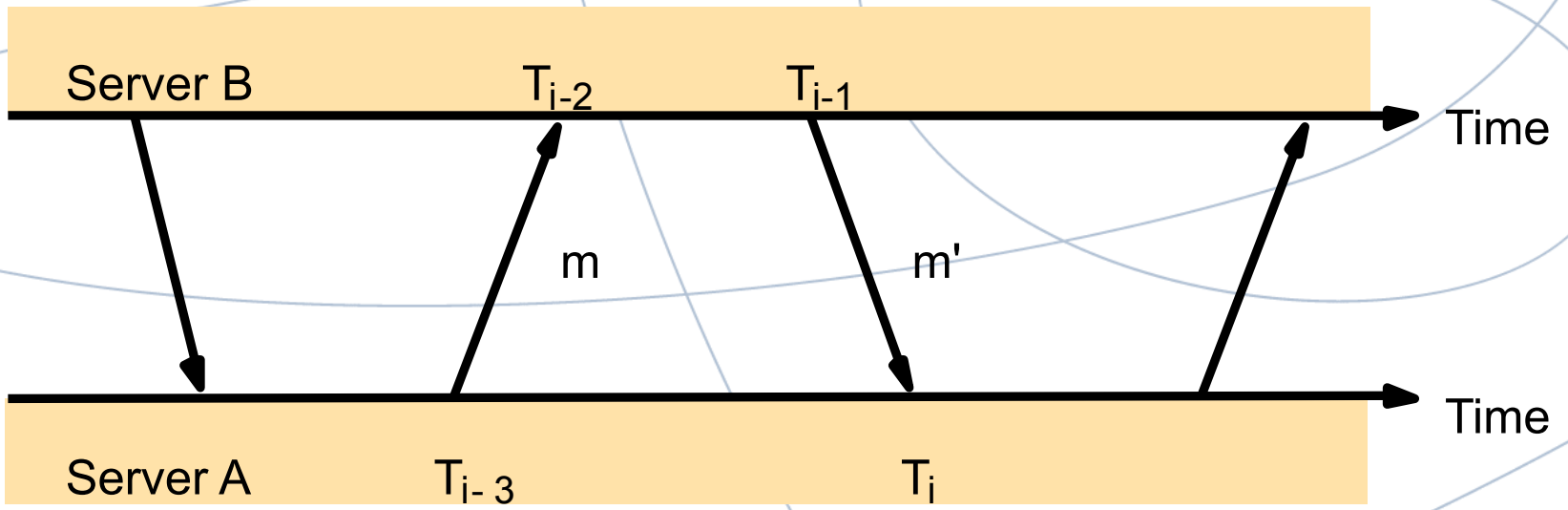


# NTP – sincronização dos servidores

- **A rede pode reconfigurar-se quando ocorrem falhas:**
  - Se um servidor primário perde a ligação com a fonte de UTC então torna-se um servidor secundário
  - Um secundário que perca o seu servidor primário pode usar outro primário
- **Modos de sincronização:**
  - Multicast (vocacionado para uma rede LAN)
    - Servidor faz LAN multicast para outros servidores que ajustam os seus relógios assumindo um dado delay (não é muito preciso, aproximação de síncrono)
  - Invocação remota (análogo ao Cristian)
    - Servidor aceita pedidos de outros
    - Responde com o seu tempo actual
    - Útil se não houver multicast hardware
  - Simétrica:
    - Pares de servidores trocam mensagens que contêm informação sobre o tempo
    - usado quando é necessário maior precisão
  - Em todos os casos é usado UDP

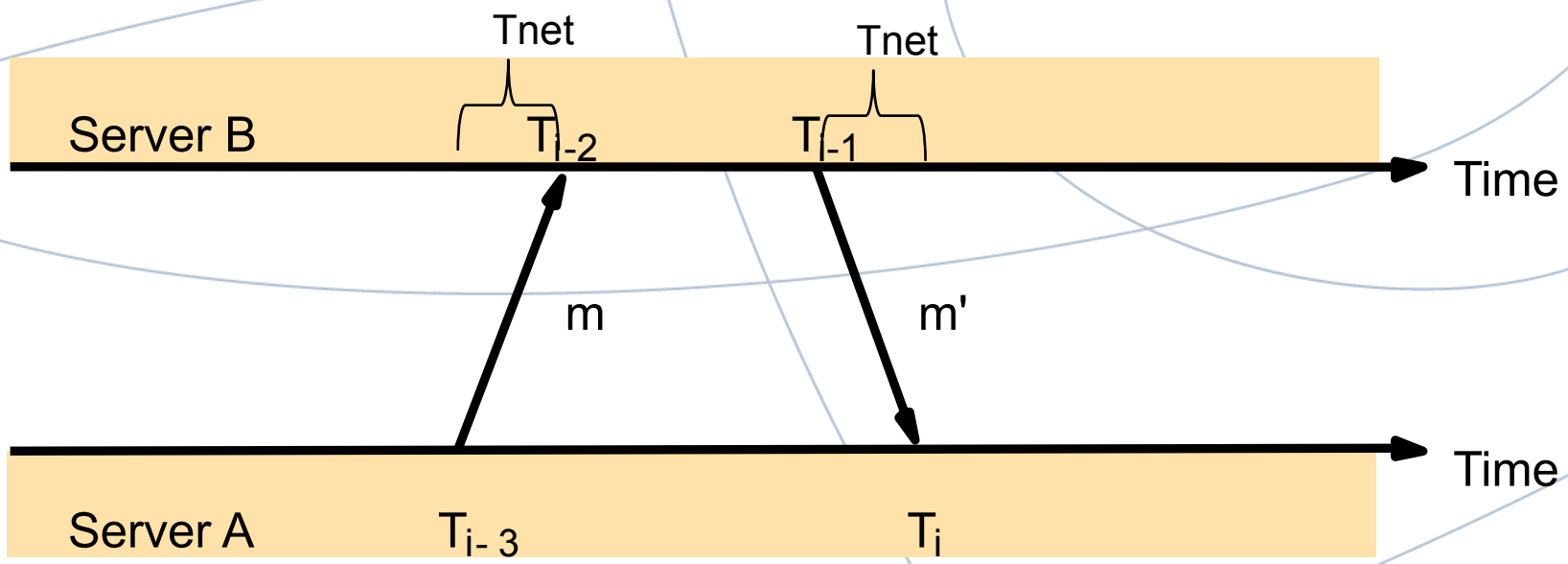
# Mensagens entre Pares NTP

- **Todos os modos usam UDP**
- **Cada mensagem leva os timestamps de eventos recentes:**
  - Tempo local de send e receive da mensagem anterior
  - Tempo local de send da mensagem actual
- **Servidor receptor anota o momento da recepção $T_i$ ( tem-se $T_{i-3}$, $T_{i-2}$, $T_{i-1}$, $T_i$)**
- **No modo simétrico pode haver um delay significativo entre mensagens**
  - e entre a recepção de uma mensagem e o envio da resposta



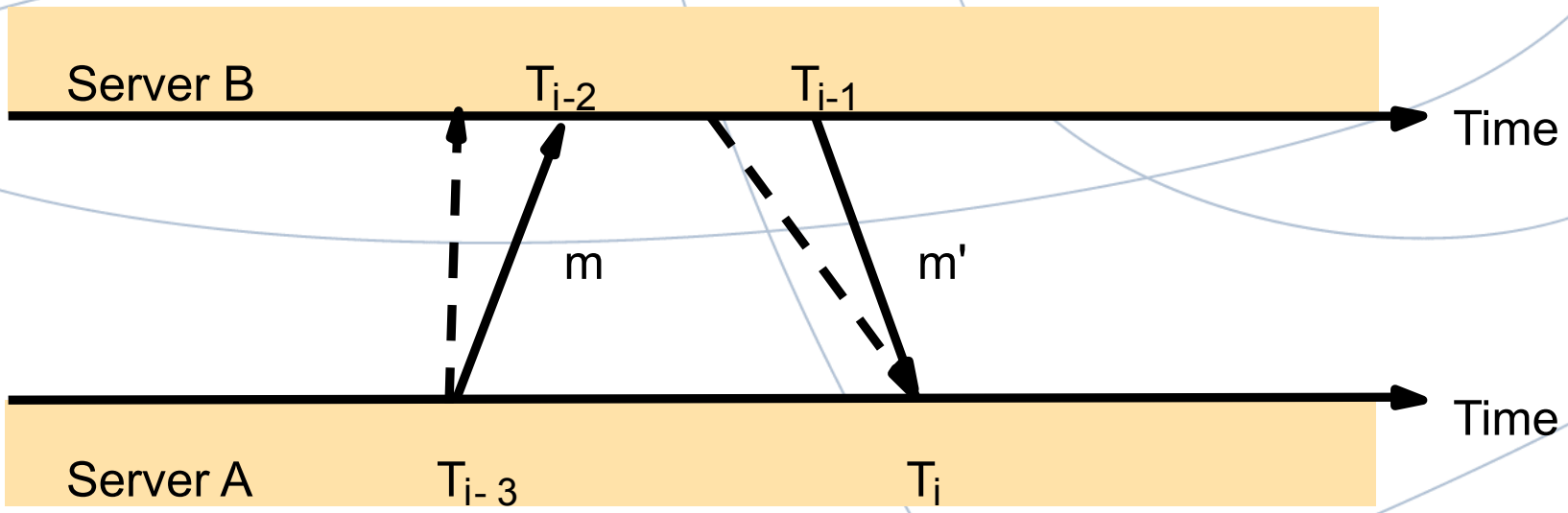
# Analysis

- Time spent in network: Tnet = $(T_i - T_{i-3}) - (T_{i-1} - T_{i-2})$



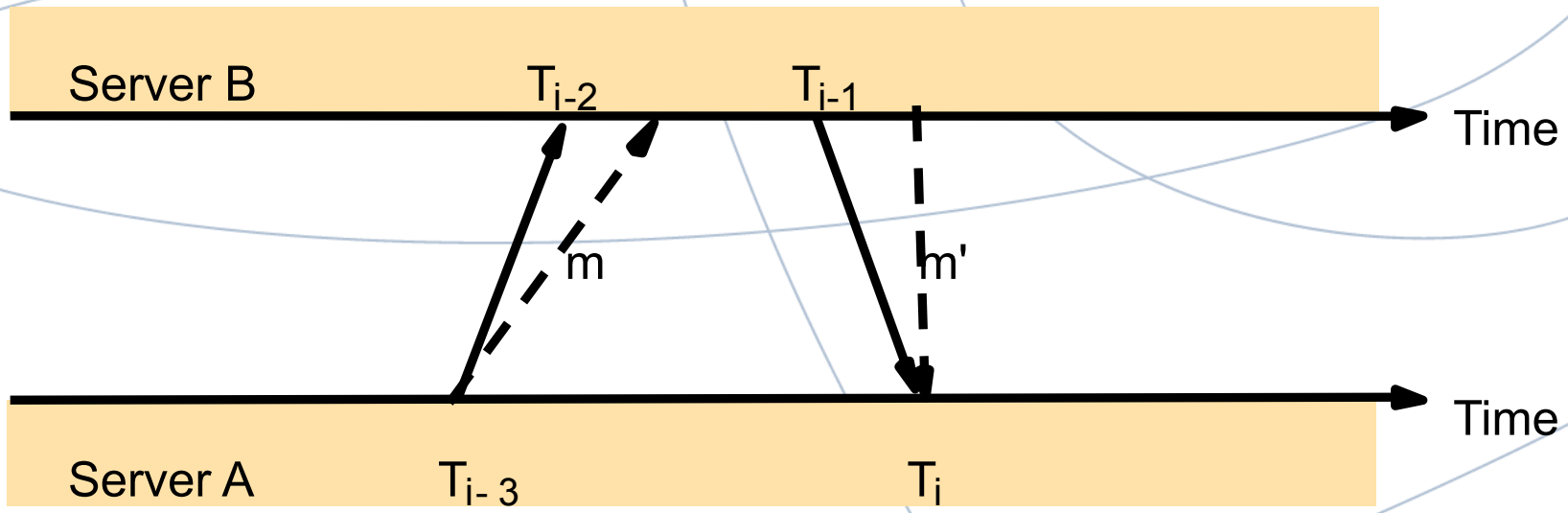
# Analysis

- Time spent in network: Tnet = $(T_i - T_{i-3}) - (T_{i-1} - T_{i-2})$
- Tnet refers to a two way communication:
  - Two extreme scenarios
    - latency 0 from Server A to Server B ➔ latency Tnet from Server B to Server A



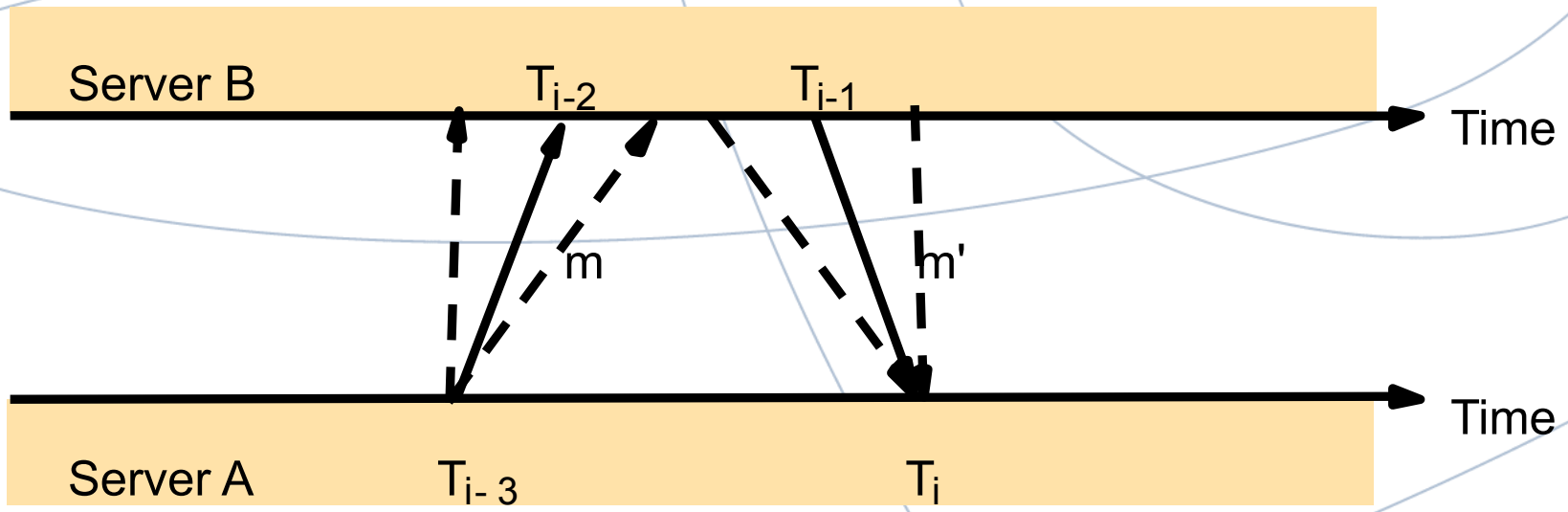
# Analysis

- Time spent in network: Tnet = $(T_i - T_{i-3}) - (T_{i-1} - T_{i-2})$
- Tnet refers to a two way communication:
  - Two extreme scenarios
    - latency 0 from Server A to Server B ➔ latency Tnet from Server B to Server A
    - latency 0 from Server B to Server A ➔ latency Tnet from Server A to Server B



# Analysis

- To minimize error vs worst case scenarios (assumed equiprobable):
  - one-way latency from A→B is estimated as Tnet / 2
  - accuracy:Tnet/2
- This allows us to estimate the offset between the clocks at B and A
  - What is the time at Server A when at Server B time i $T_{i-2}$?
    - $T_{i-3}$ + Tnet/2 ➔ offset= $T_{i-2}$ – ($T_{i-3}$ + Tnet/2)



# Precisão do NTP

- **Servidores NTP filtram os pares <$o_i$, $d_i$>, verificando quais os seus valores de modo a selecionar os pares mais interessantes:**
  - são mantidos 8 pares mais recentes
  - seleccionado $o_i$ que minimiza o valor $d_i$ (o round-trip entre duas mensagens)
    - por quê não a média?

- **É possível obter precisão de:**
  - Dezenas de milisegundos na Internet
  - 1 milisegundo em LANs

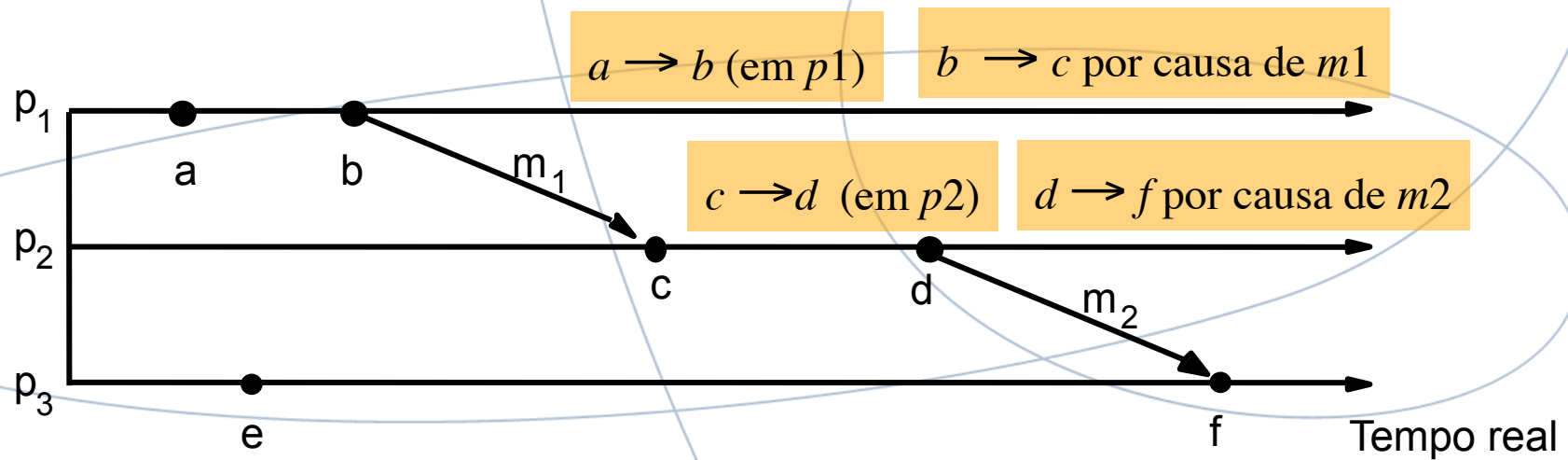
# Índice

- Introdução
- Relógios, eventos, estados de processos
- Sincronização de relógios físicos

- **Tempo lógico**
- **Relógios lógicos**
- **Relógios vectoriais**
- **Relação de causalidade *(happened-before)***

# Tempo Lógico e Relógios Lógicos (Lamport 1978)

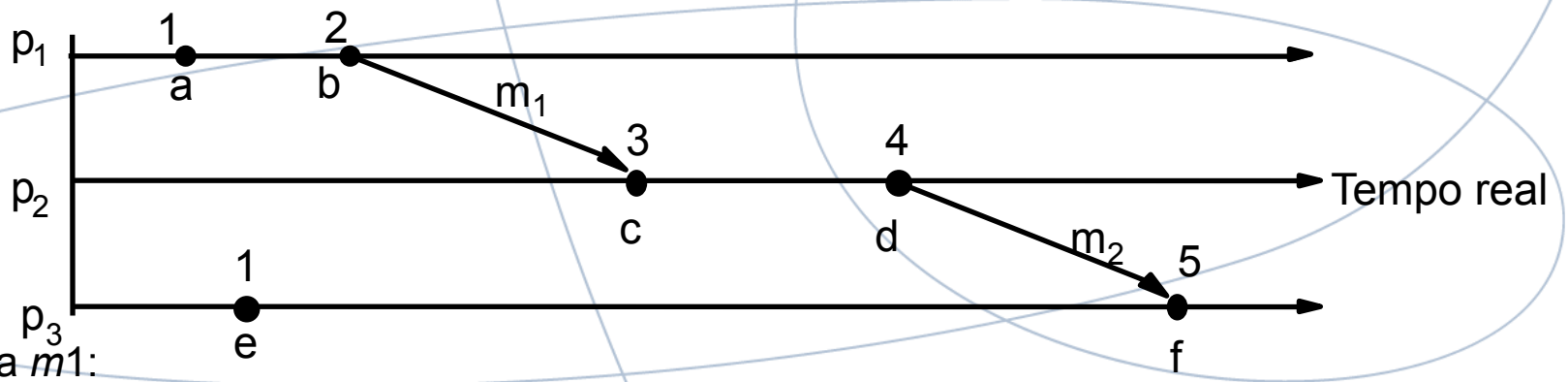
- **Em vez de contar o tempo real, com relógios nos computadores, pode-se usar tempo lógico para ordenar eventos:**
  - **Se dois eventos ocorreram no mesmo processo $p_i$ ($i$ = 1, 2, … $N$) então eles ocorreram na ordem em que foram observados por $p_i$ (relação *happened before*)**
  - **Quando a mensagem m é trocada entre dois processos, *send(m)* happened before *receive(m),***
  - **A relação *happened before* é transitiva**




- **A relação *happened* before é a relação de ordem causal**
- **Nem todos os eventos estão relacionados causalmente:**
  - Eventos *a* e *e* ocorrem em processos diferentes e não há nenhuma cadeia de mensagens que os relacione
  - Portanto, não há uma relação de happened before
  - Estes eventos dizem-se concorrentes (a || e)

# Relógios Lógicos de Lamport

- **Um relógio lógico é um contador (software) monotónico (não é preciso dispor de um relógio físico nem se relaciona com tal)**
- **Cada processo $p_i$ tem um relógio lógico $L_i$, inicializado a zero, que é usado para carimbar (timestamping) os eventos:**
  - **LC1: $L_i$ é incrementado de 1 unidade antes de cada evento no processo $p_i$**
  - **LC2: (a) quando o processo $p_i$ envia uma mensagem *m*, faz piggyback $t = L_i$**
    - **(b) quando $p_j$ recebe *(m,t) faz* $L_j := max(L_j, t)$ e aplica LC1 antes de carimbar o evento *receive* (*m*)**




- Para *m*1:
  - O valor 2 é piggybacked de modo que c ocorre no momento max(0,2)+1 = 3
- Notar que:
  - $e \rightarrow e'$ implica $L(e)<L(e')$ mas o inverso não é verdade
  - $L(e)<L(e')$ não implica $e \rightarrow e'$

$L(e) < L(b)$ mas $e \parallel b$

# Relógios Vectoriais (1)

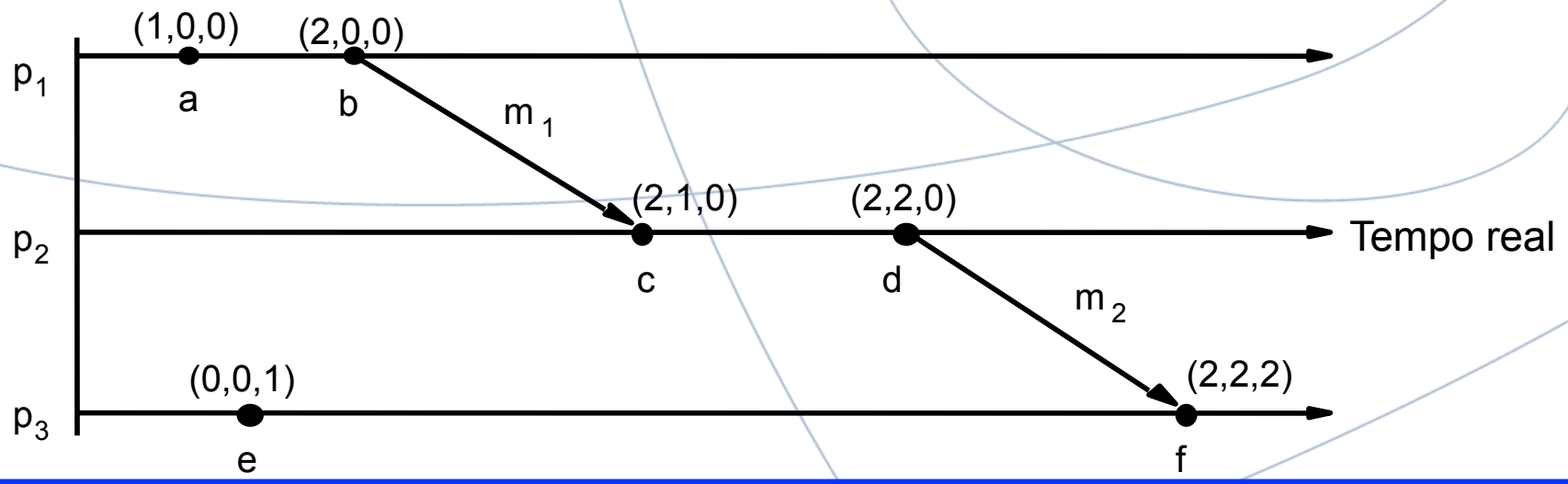
- **Surgem para resolver desvantagem do relógio lógico de Lamport:**
  - *L*(e) < *L*(*e*') não implica que *e happened before e*'
- **Relógio vectorial $V_i$ no processo $p_i$ é um array de *N* inteiros**
- **Regra VC1:**
  - inicialmente $V_i[j] = 0$ para $i, j = 1, 2, \ldots N$
- **Regra VC2:**
  - antes de $p_i$ carimbar um evento faz $V_i[i] := V_i[i] + 1$
- **Regra VC3:**
  - $p_i$ faz piggyback de $t = V_i$ em cada mensagem enviada
- **Regra VC4:**
  - quando $p_i$ recebe($m$,$t$) faz $V_i[j] := \max(V_i[j], t[j]), \ j = 1, 2, \ldots N$
  - depois, antes do próximo evento, adiciona 1 usando VC2

# Relógios Vectoriais (2)

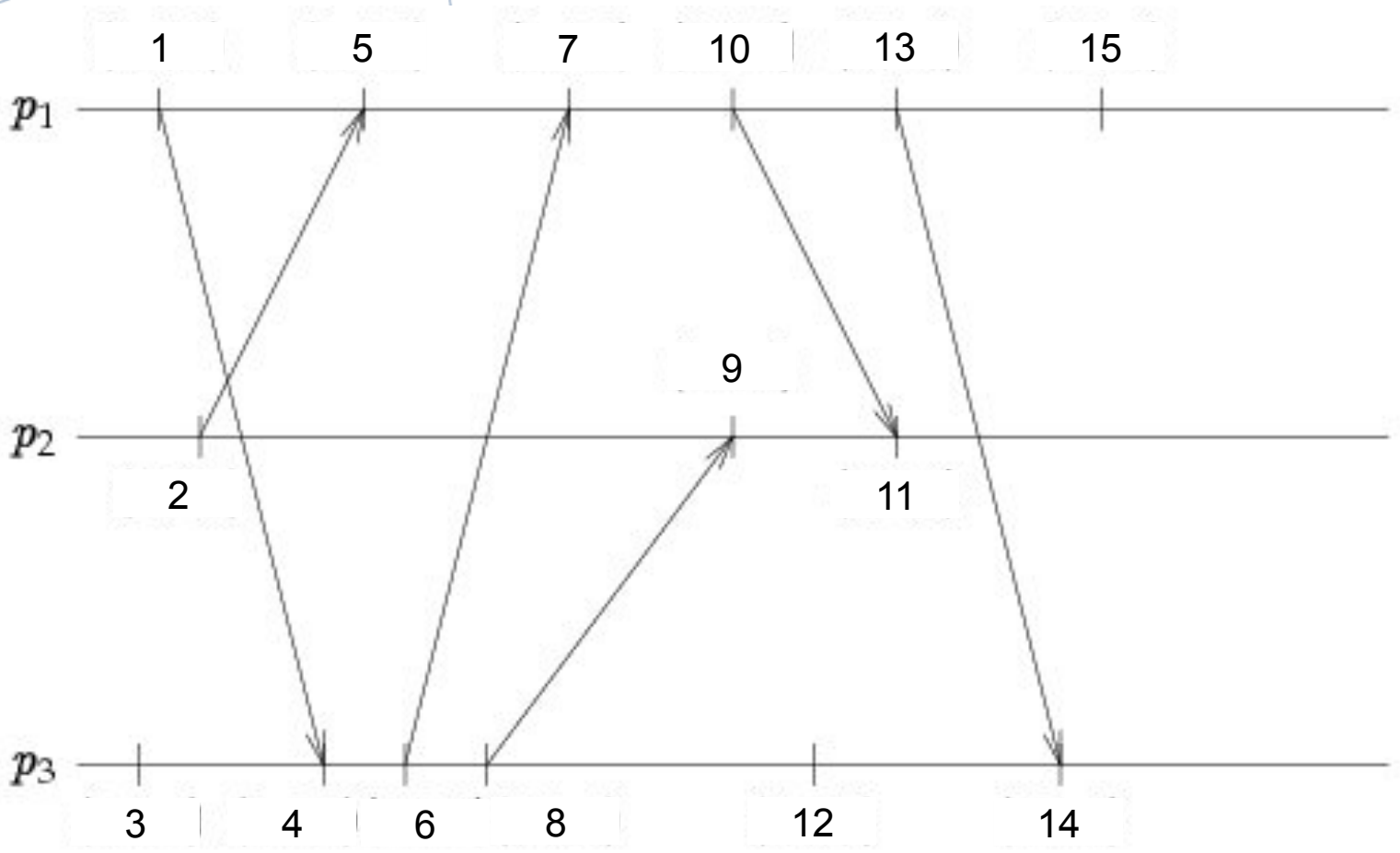
- **São usados para carimbar eventos locais e têm aplicação em:**
  - protocolos de coerência para dados replicados (e.g. Coda,Gossip)
  - protocolos de comunicação causal
- $V_i[i]$ é o número de eventos que $p_i$ já carimbou
- $V_i[j]$ ( $j \neq i$) é o número de eventos em $p_j$ e que já afectaram $p_i$
- Significado de =, <=, < para relógios vectoriais:
  - 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<= V' e V !=V'

# Relógios Vectoriais (3)

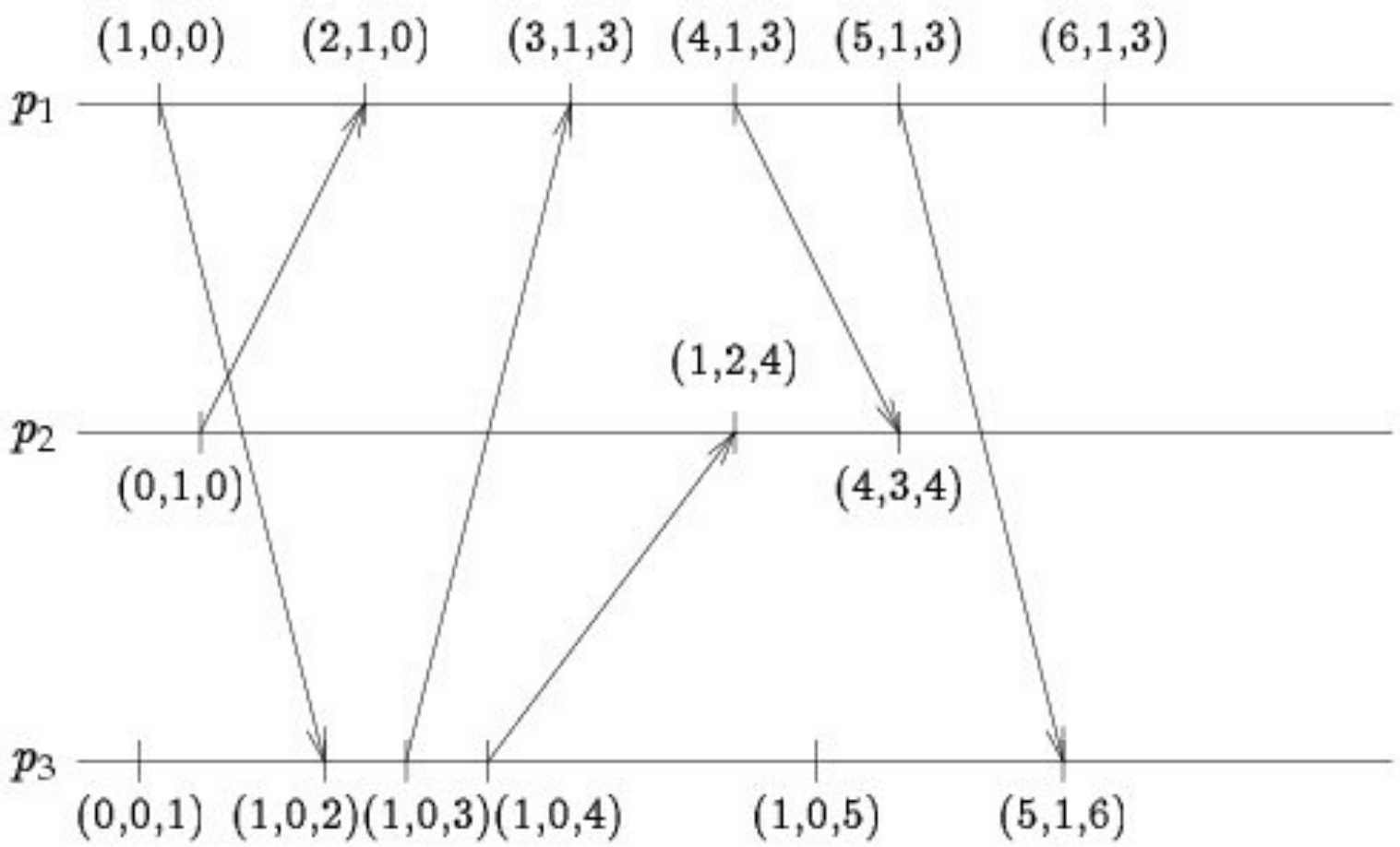
- **Em p1:**
  - *a*(1,0,0), *b* (2,0,0), piggyback (2,0,0) em *m*1
- **Em p2, quando recebe m1:**
  - *max* ((0,0,0), (2,0,0)) = (2, 0, 0)
  - adicionar 1 no elemento correspondente a p2: (2,1,0)
- **Notar que:**
  - *c* || *e* (concorrentes) **Q: Há outros pares de eventos concorrentes?**



# Relógios Vectoriais (4)



# Relógios Vectoriais (4)



# Causalidade na Entrega de Mensagens (1)

- **Quando um processo $p_i$ recebe uma mensagem M de $p_j$, esta só pode ser entregue a $p_i$ depois de:**
  - Ter a certeza que todas as mensagens que precedem causalmente M, foram de facto entregues a $p_i$
- **Para tal, antes de entregar M, $p_i$ espera que:**
  - lhe tenham sido entregues todas as mensagens que foram antes enviadas por $p_j$
  - lhe tenham sido entregues todas as mensagens que tenham sido entregues a $p_j$ até ao momento em que $p_j$ enviou M
- **Aplicável no âmbito de comunicação multicast**
  - Apenas são marcados os eventos correspondentes a envio de mensagens
    - Anteriormente marcámos todos os eventos (envio, recepção, internos)

# Causalidade (2)

- ***e* → *e'* implica *V*(*e*)<*V*(*e'*)**
- ***V*(*e*)<*V*(*e'*) implica *e* → *e'***
- **Entrega Causal (causal delivery):**
  - mensagem m enviada por pj é entregue assim que p0 verifique que não há outras mensagens cujo envio preceda causalmente o envio da mensagem m

each message $m$ carries a timestamp $TS(m)$ which is the vector clock value of the event being notified by $m$. All messages that have been received but not yet delivered by the monitor process $p_0$ are maintained in a set $\mathcal{M}$, initially empty.

Let $m'$ be the last message delivered from process $p_k$, where $k \neq j$. Before message $m$ of process $p_j$ can be delivered, $p_0$ must verify two conditions:

1. there is no earlier message from $p_j$ that is undelivered, and
2. there is no undelivered message $m''$ from $p_k$ such that

$$send(m') \rightarrow send(m'') \rightarrow send(m), \forall k \neq j.$$

# Causalidade (3)

Let $m'$ be the last message delivered from process $p_k$, where $k \neq j$. Before message $m$ of process $p_j$ can be delivered, $p_0$ must verify two conditions:

1. there is no earlier message from $p_j$ that is undelivered, and
2. there is no undelivered message $m''$ from $p_k$ such that

$$send(m') \rightarrow send(m'') \rightarrow send(m), \forall k \neq j.$$

- **condição 1 é respeitada se:**
  - exactamente **TS(m)[ j ] - 1** mensagens enviadas por pj já tiverem sido entregues
- **condição 2 é respeitada se:**
  - **TS(m')[ k ] >= TS(m)[k]**, p/ todo k ≠ j (não há eventos entre m' e m relacionados causalmente)
- **Portanto, a regra é a seguinte**

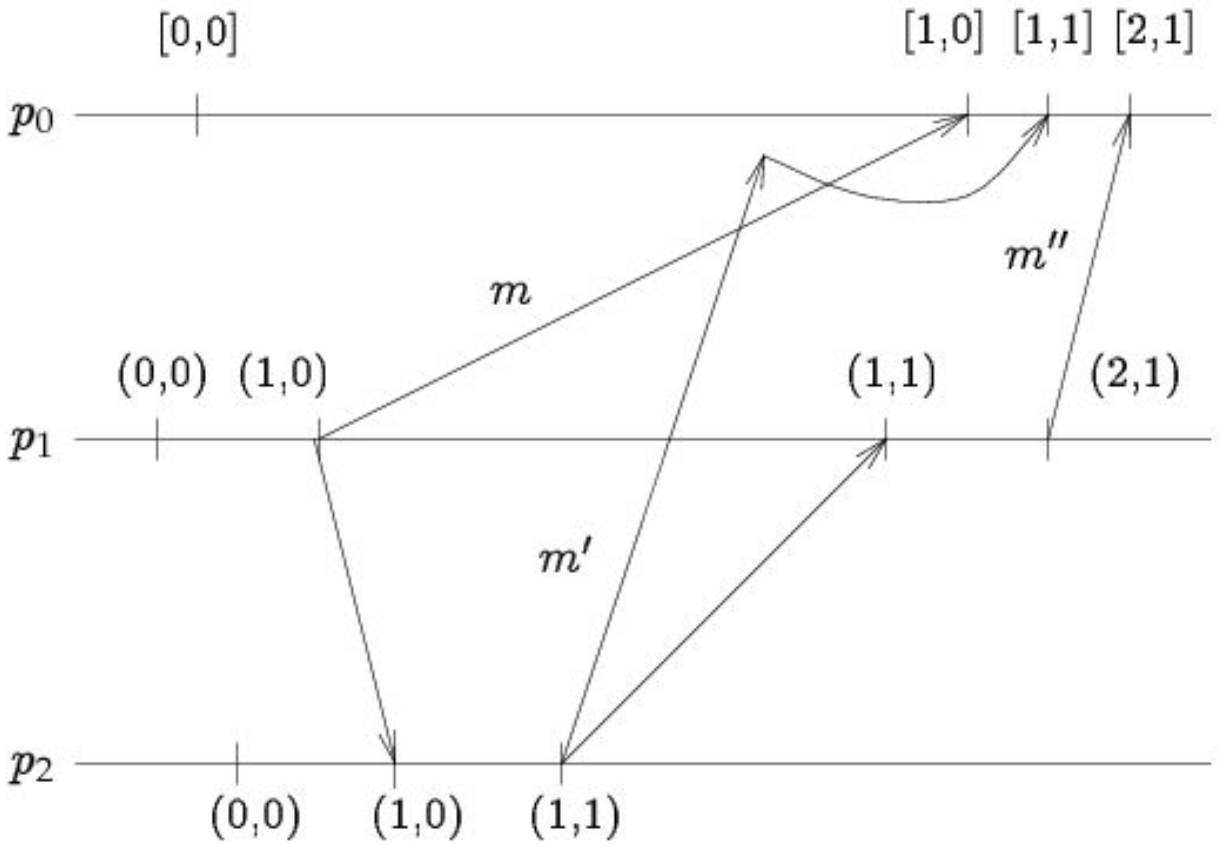
(Causal Delivery) Deliver message $m$ from process $p_j$ as soon as both of the following conditions are satisfied

$$\begin{aligned} D[j] &= TS(m)[j] - 1 \\ D[k] &\geq TS(m)[k], \forall k \neq j. \end{aligned}$$

D é array de contadores em p0

When $p_0$ delivers $m$, array $D$ is updated by setting $D[j]$ to $TS(m)[j]$.

# Causalidade (4)



# Causalidade (5)

Cada processo tem o seu vector

Algorithm for group member $p_i$ $(i = 1, 2..., N)$

processo i envia mensagens

*On initialization*

$V_i^g[j] := 0\ (j = 1, 2..., N);$

*To CO-multicast message m to group g*

$V_i^g[i] := V_i^g[i] + 1;$

*B-multicast*$(g, <V_i^g, m>);$

Envio de *m* para *g*: processo *i* ("sender") adiciona 1 à sua entrada no vector e envia *m* mais o vector

Quando é entregue a um processo $p_i$ a mensagem *m*, esta é colocada numa fila de espera para garantir que outras mensagens que a precedem causalmente sejam entregues:

a) lhe tenham sido entregues todas as mensagens que foram antes enviadas pelo mesmo "sender"

b) lhe tenham sido entregues todas as mensagens que foram entregues ao "sender" até ao momento em que o "sender" enviou a mensagem

processo i recebe mensagens

*On B-deliver*$(<V_j^g, m>)$ *from* $p_j$ $(j \neq i)$, *with* $g = group(m)$

place $<V_j^g, m>$ in hold-back queue;

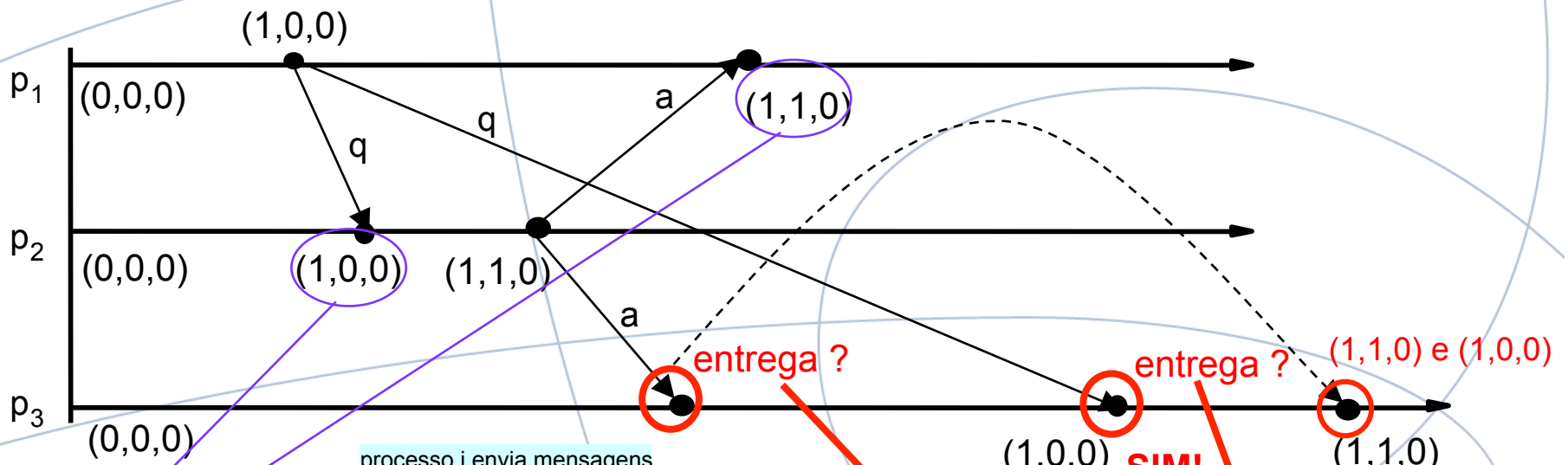
wait until $V_j^g[j] = V_i^g[j] + 1$ and $V_j^g[k] \leq V_i^g[k]\ (k \neq j)$;

*CO-deliver* $m$; // after removing it from the hold-back queue

$V_i^g[j] := V_i^g[j] + 1;$

Entrega a mensagem e actualiza o seu vector

# Causalidade (6)




processo i envia mensagens

- CO-multicast

$V_i^g[i] := V_i^g[i] + 1;$

*B-multicast*$(g, <V_i^g, m>)$

- B-deliver

processo i recebe mensagens enviadas por j

place $<V_j^g, m>$ in hold-back queue;

wait until $V_j^g[j] = V_i^g[j] + 1$ and $V_j^g[k] \leq V_i^g[k]$ $(k \neq j)$;

*CO-deliver* $m$; // after removing it from the hold-back queue

$V_i^g[j] := V_i^g[j] + 1;$

Regras diferentes dos relógios vectoriais vistos anteriormente!

# Resumo

- **Conhecimento/contagem do tempo é importante em sistemas distribuídos**
- **Há algoritmos que permitem sincronizar os relógios físicos de computadores diferentes:**
  - Apesar do seu desvio (drift) e variabilidade no tempo de transmissão das mensagens na rede
  - Cristian, Berkeley
  - NTP
- **Para ordenar um qualquer par de eventos em computadores diferentes a sincronização de relógios físicos não é absolutamente necessária**
- **A relação happened-before permite:**
  - ter uma ordenação parcial dos eventos reflectindo o fluxo de informação entre eles
- **Os relógios de Lamport são contadores que:**
  - são incrementados de acordo com a relação ***happened-before*** entre eventos
- **Relógios vectoriais são uma evolução do relógio de Lamport:**
  - Permitem dizer se dois eventos estão relacionados pela relação ***happened-before*** ou se são concorrentes comparando os vectores respectivos
  - Com pequenas alterações, permitem assegurar entrega causal de mensagens num dado grupo de processos