# Logical Time and Ordered Multicast

Sistemas Distribuídos 2013/2014

# Logical Time and Logical Clocks

# Logical time

- For many purposes it is sufficient to agree on the same time (e.g., internal consistency) which need not be UTC time
- Can deduce causal event ordering

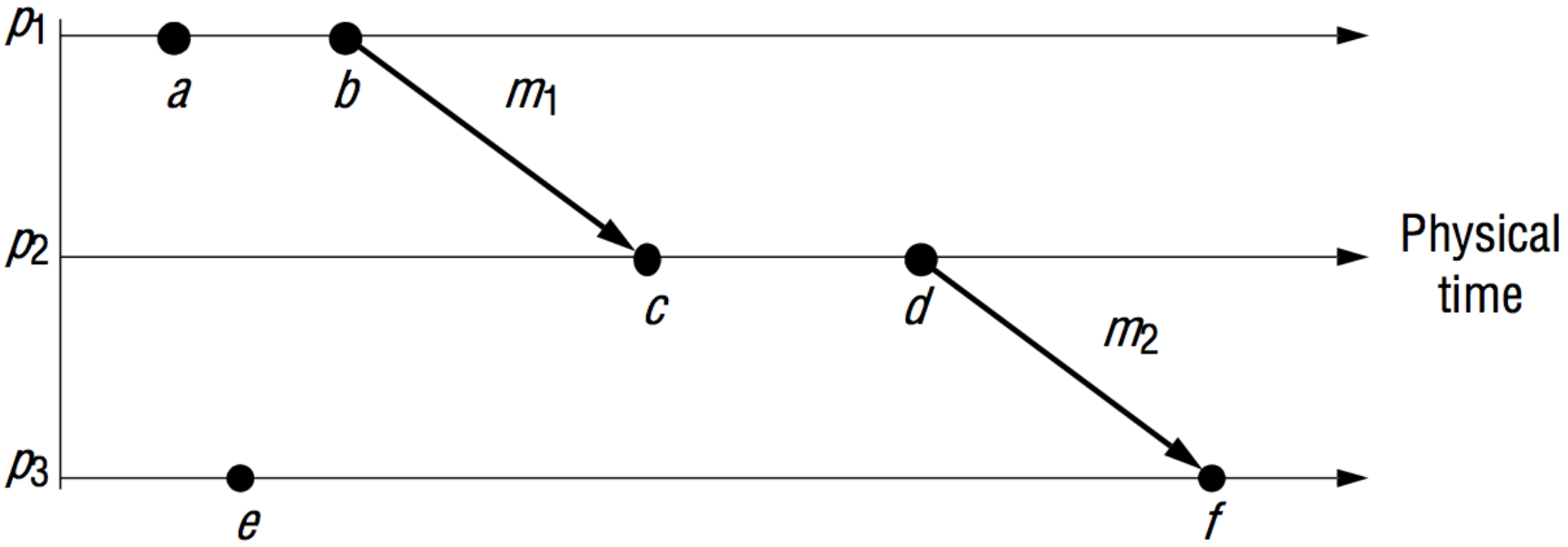
  a $\rightarrow$ b (a occurs before b)

- Logical time denotes causal relationships

# Causal ordering of events

- Define a $\rightarrow$ b (a happened before b):
  - If a and b are events in the same process and a occurs before b, then a $\rightarrow$ b.
  - If a is the event of message send from process A, and b is the event of message receipt by process B, then a $\rightarrow$ b.
  - If a $\rightarrow$ b and b $\rightarrow$ c, then a $\rightarrow$ c (transitivity).
- $\rightarrow$ is a strict partial order.
- Events for which neither a $\rightarrow$ b nor b $\rightarrow$ a hold are said to be concurrent, denoted a $||$ b.

# Example




- a $\rightarrow$ b, c $\rightarrow$ d
- b $\rightarrow$ c, d $\rightarrow$ f
- a $||$ e

# Lamport clocks

- Lamport defined the happened-before relation for distributed systems.
- $A \rightarrow B$ means "A happens before B".
- If A and B are events in the same process and A occurs before B then $A \rightarrow B$ is true.
- If A is the event of a message being sent by one process and B is the event of that message being received by another process, then $A \rightarrow B$ is true. (A message must be sent before it is received.)
- Transitive closure: if $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$.
- Any other events are said to be concurrent.

# Lamport clocks

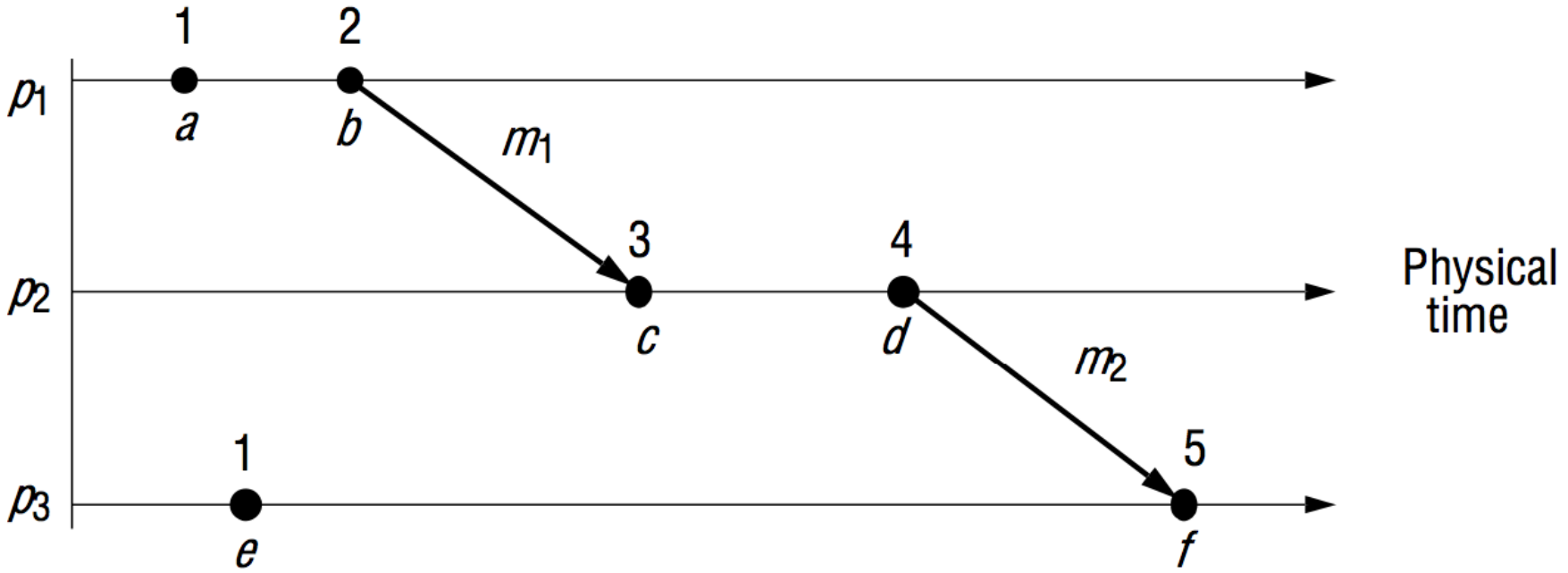
Desired properties:

- (1) anytime A $\rightarrow$ B, C(A) $<$ C(B), that is the logical clock value of the earlier event is smaller
- (2) the clock value C is monotonically increasing (never runs backwards)

# Lamport clock rules

- An event is an internal event or a message send or receive. The local clock is increased for an internal event.
- The local clock is increased by one for each message sent and the message carries that timestamp with it (piggyback).
- When a message is received, the current local clock value, C, is compared to the message timestamp, T.
  - If the message timestamp, $T = C$, then set $C=C+1$.
  - If $T > C$, set the clock to $T+1$.
  - If $T < C$, set the clock to $C+1$.

# Lamport clocks

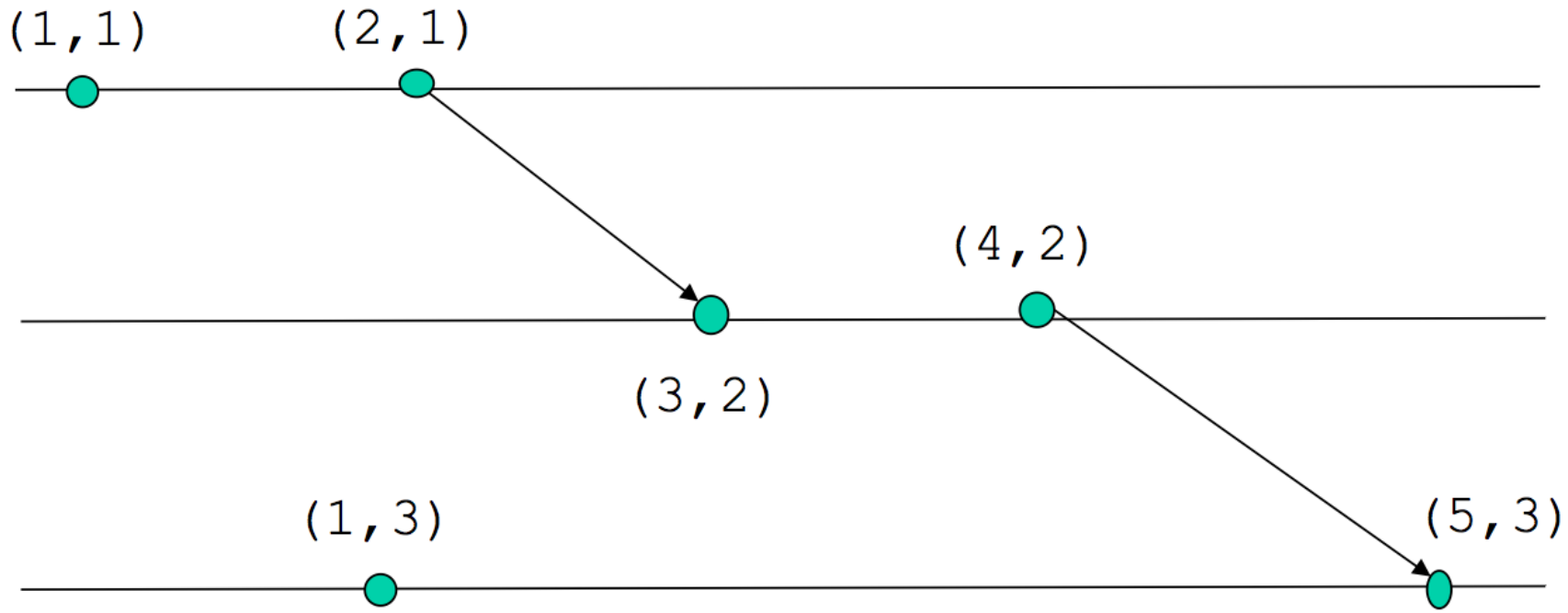



- b $\rightarrow$ c means that C(b) $<$ C(c)
- d $\rightarrow$ f means that C(d) $<$ C(f)
- However, C(i) $<$ C(j) doesn't mean i $\rightarrow$ j
- (e.g., C(e) $<$ C(b) but it is not true that e $\rightarrow$ b)

# Totally ordered logical clocks

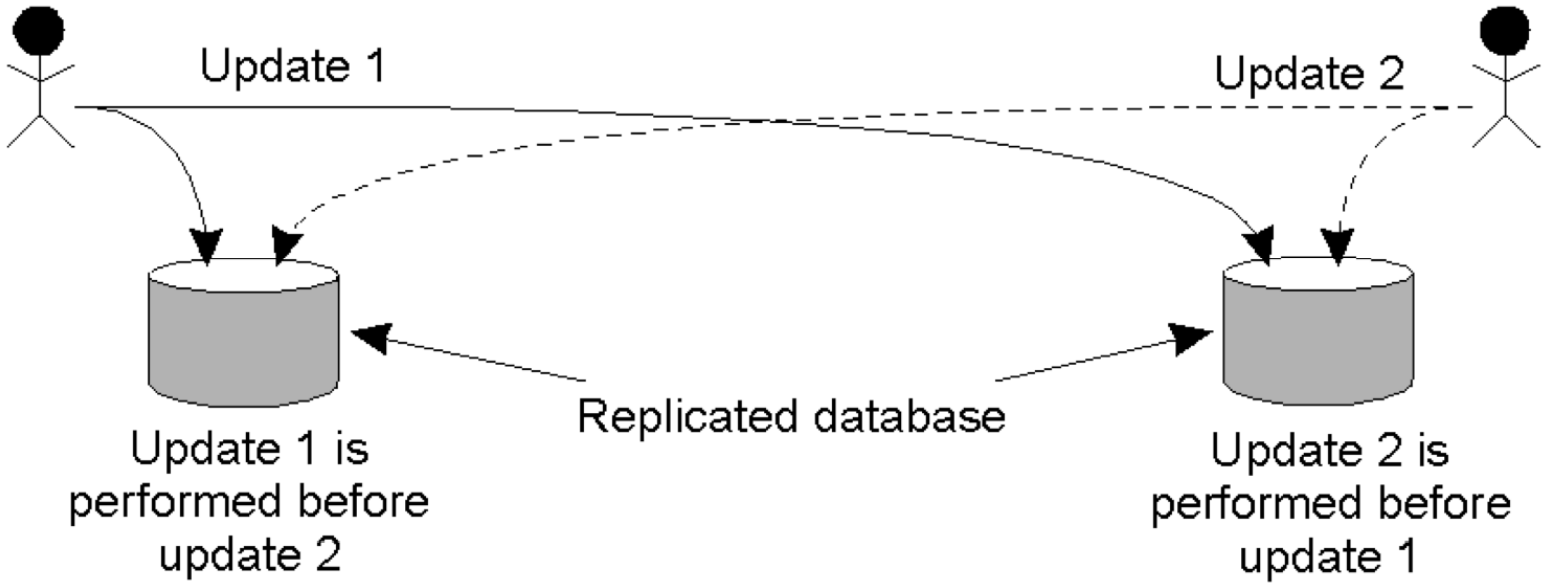
- Some pairs of distinct events, generated by different processes, have numerically identical Lamport timestamps.
- If we need a total ordering, we can use global logical timestamps.
- Global logical timestamp of event A at process i is (C(A), i).
- For any 2 timestamps T1=(C(A),I) and T2=(C(B),J)
  - If C(A) $>$ C(B) then T1 $>$ T2.
  - If C(A) $<$ C(B) then T1 $<$ T2.
  - If C(A) $=$ C(B) then consider process IDs.
    - If I $>$ J then T1 $>$ T2.
    - If I $<$ J then T1 $<$ T2.
    - If I $=$ J then the two events occurred at the same process, so since their clock C is the same, they must be the same event.

# Totally ordered logical clocks




- The order will be (1,1), (1,3), (2,1), (3,2), (4,2), (5,3)

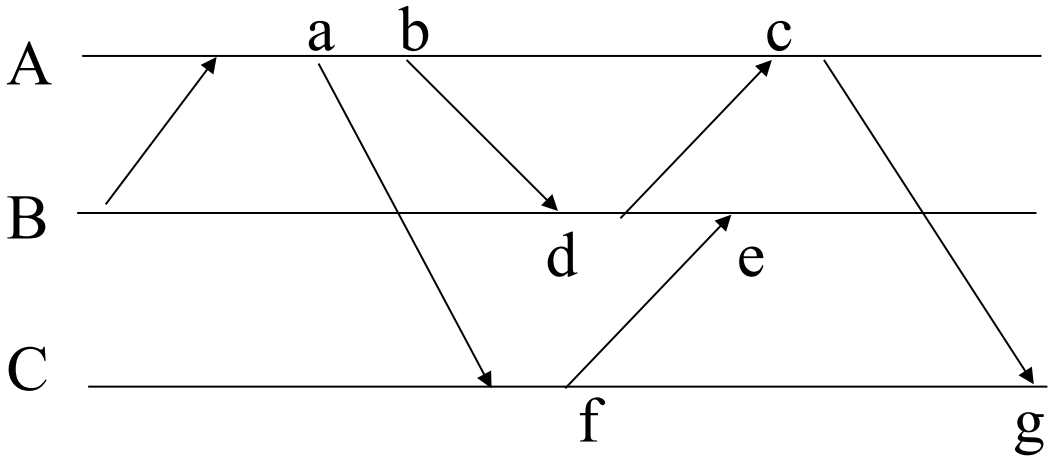
# Why total order?




Database updates need to be performed in the same order at all sites of a replicated database.

- Process P1 adds \$100 to an account (initial value: \$1000)
- Process P2 increments account by 1%
- Replica #1 will end up with \$1111, while replica #2 ends up with \$1111.

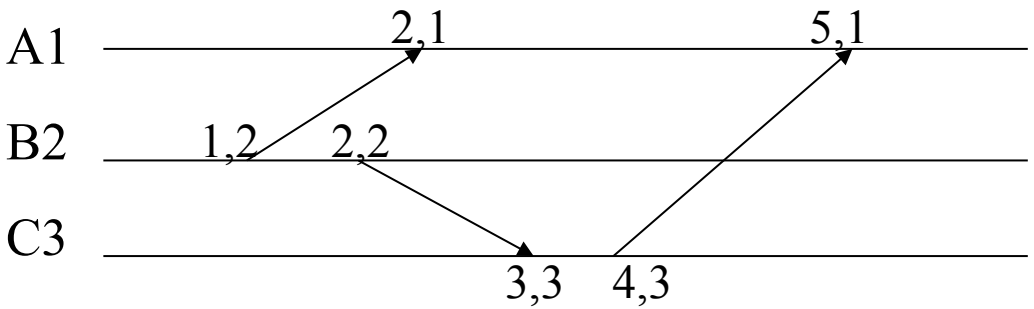
# Exercise: Lamport Clocks




- Assuming the only events are message send and receive, what are the clock values at events a-g?

# Limitation of Lamport Clocks

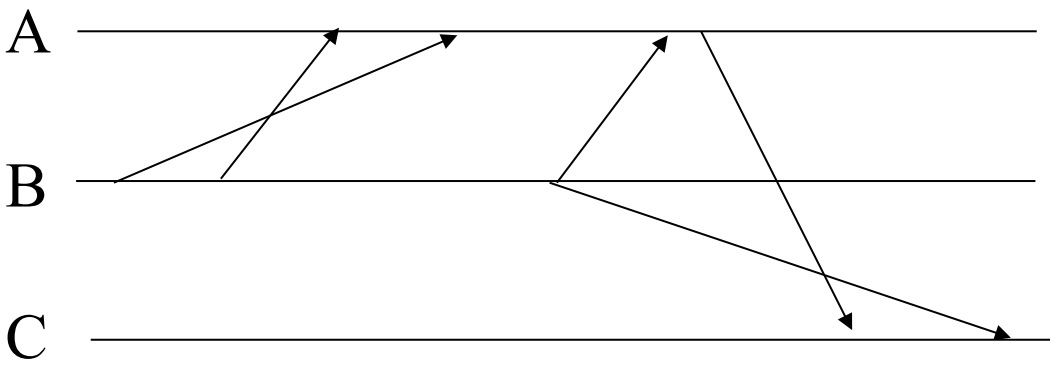
- Total order Lamport clocks gives us the property if A → B then C(A) < C(B).
- But it doesn't give us the property if C(A) < C(B) then A→B.
- If C(A) < C(B), A and B may be concurrent or incomparable, but never B→A.




Lamport timestamp of 2,1 < 3,3 but the events are unrelated

# Limitation

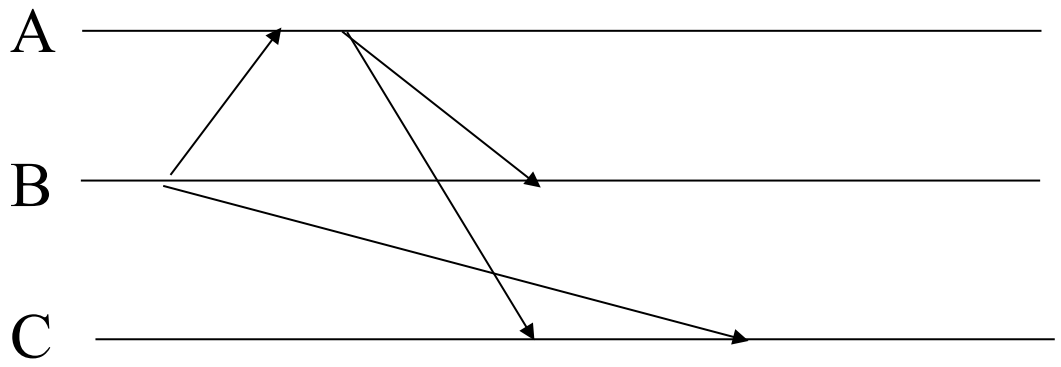
- Also, Lamport timestamps do not detect causality violations. Causality violations are caused by long communications delays in one channel that are not present in other channels or a non-FIFO channel.




A and C will never know messages were out of order

# Causality Violation

- Causality violation example: A gets a message from B that was sent to all nodes. A responds by sending an answer to all nodes. C gets A's answer to B before it receives B's original message.
- How can B tell that this message is out of order?
  - Assume one send event for a set of messages



# Causality: Solution

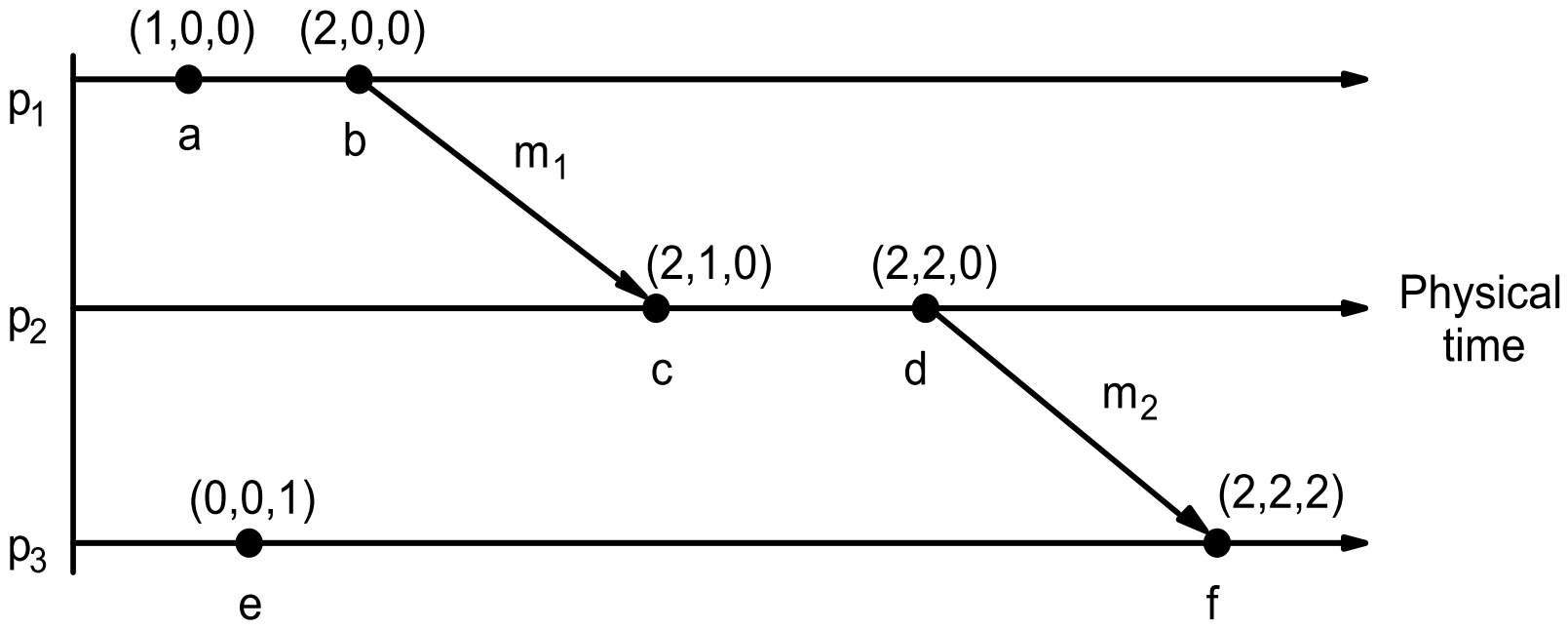
- The solution is vector timestamps: Each node maintains an array of counters.
- If there are N nodes, the array has N integers VT(N). VT(I) = C, the local clock, if I is the designation of the local node.
- In general, VT(X) is the latest info the node has on what X's local clock is.
- Gives us the property e → f iff VT(e) < VT(f)

# Vector Timestamps

- Each site has a local clock incremented at each event.
- The vector clock timestamp is **piggybacked** on **each** message sent.

- **RULES:**
  - Local clock is incremented for a local event and for a send event. The message carries the vector time stamp.
  - When a message is received, the local clock is incremented by one. Each other component of the vector is increased to the received vector timestamp component if the current value is less. That is, the maximum of the two components is the new value.

# Vector Clocks



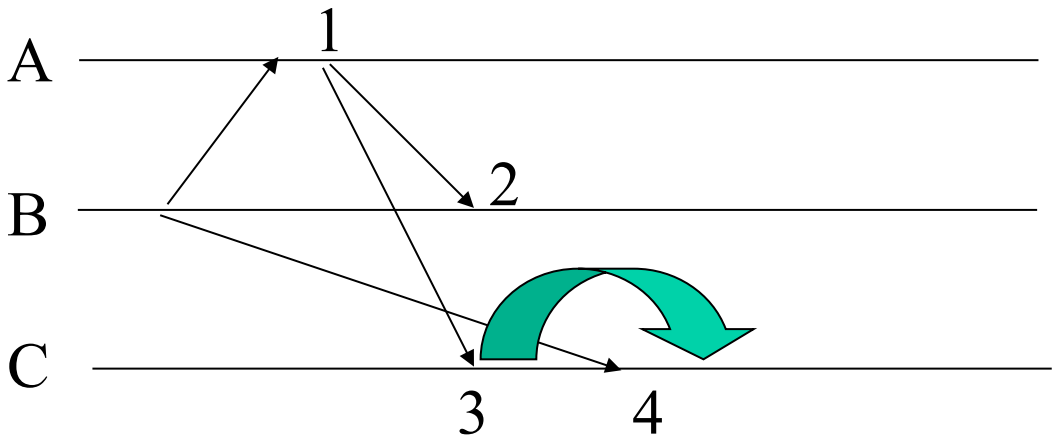

VT(a) < VT(b), hence a→ b

neither VT(x) < VT(y), nor VT(y) < VT(x), hence x || y

# Vector Clock Comparison

- VT1 > VT2 if for each component j, VT1[j] >= VT2[j], and for some component k, VT1[k] > VT2[k]
- VT1 = VT2 if for each j, VT1[j] = VT2[j]
- Otherwise, VT1 and VT2 are incomparable and the events they represent are concurrent

Assume that only sends e recvs are the counting events

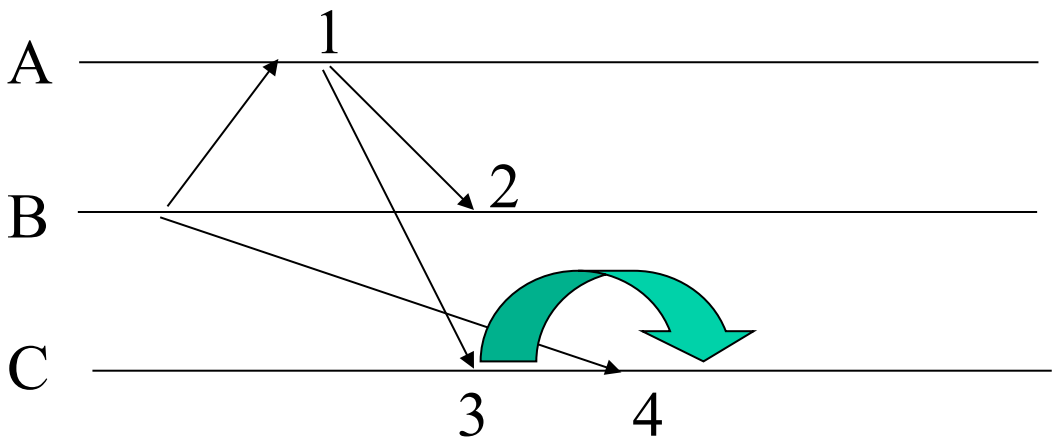



Clock at point
1= (2,1,0)
2= (2,2,0)
3= (2,1,1)
4= (2,1,2)

??

# Vector Clock Comparison

Assume that only send_message() is the counting event

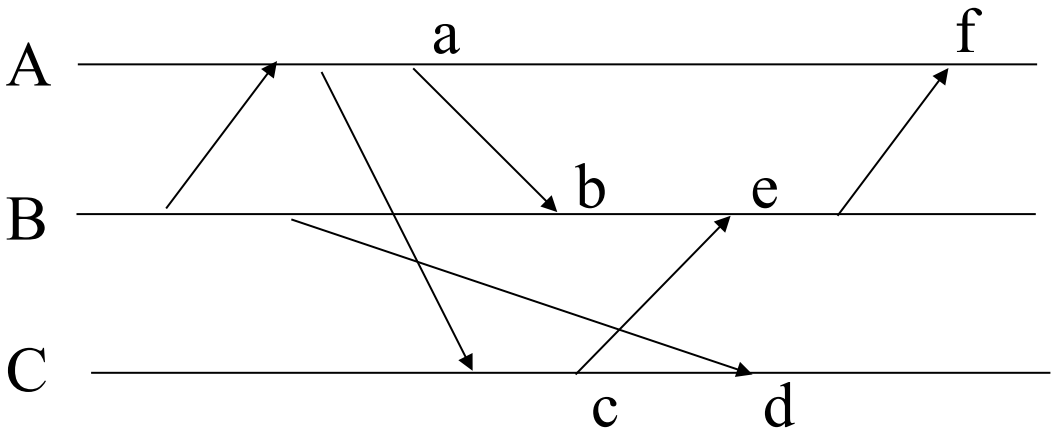



Clock at point
1= (0,1,0)
2= (1,1,0)
3= (1,1,0)
4= (0,1,0)

??

# Vector Clock Exercise

- Assuming the counting events are send and receive:
- What is the vector clock at events a-f?
- Which events are concurrent?

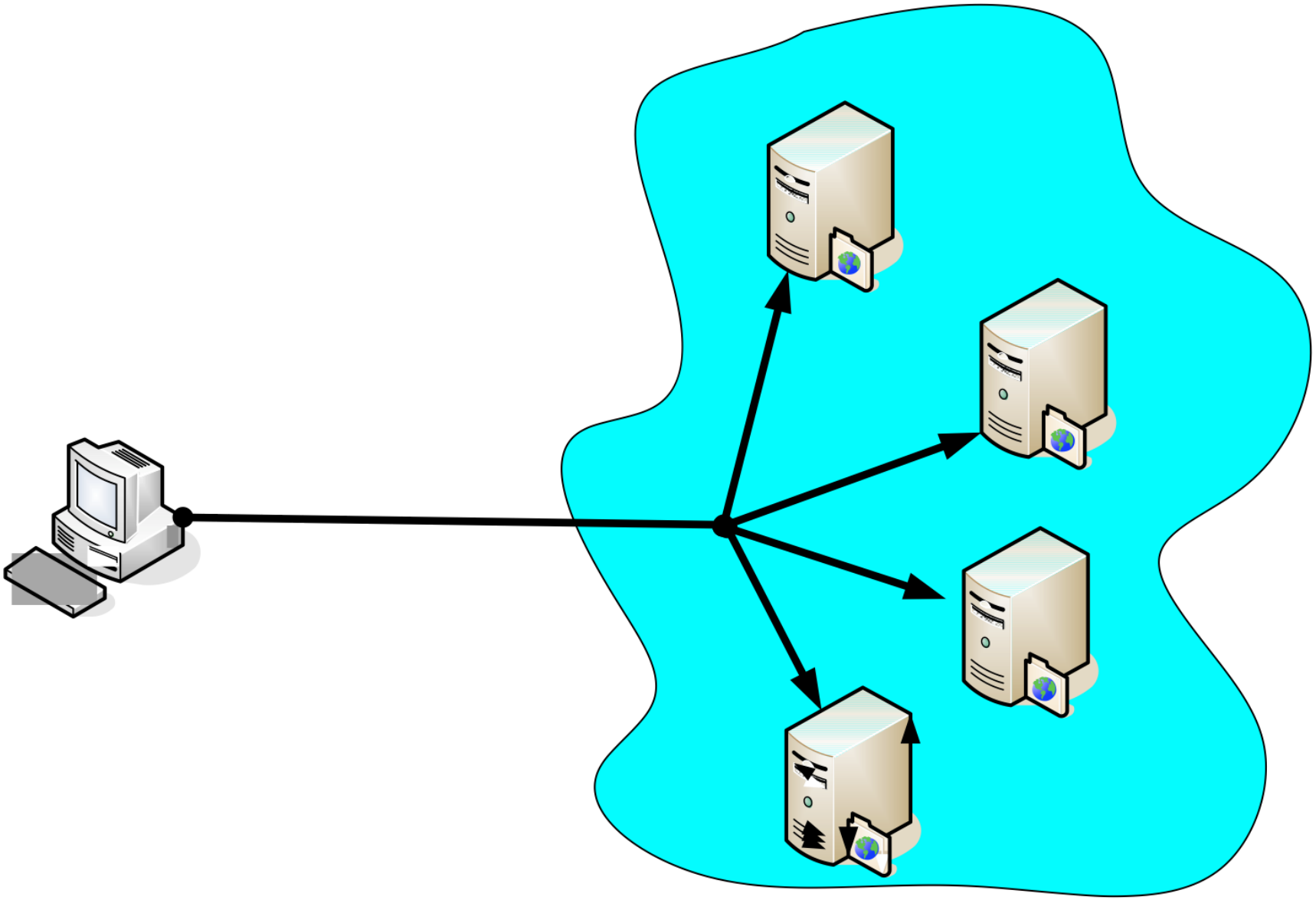


# Ordem em Comunicação Multicast

# Objectives

- Achieve reliability and ordering of message delivery across all members of a group.
- Reliability in terms of:
  - The properties of validity, integrity and agreement.
- Ordering in terms of:
  - FIFO ordering, causal ordering and total ordering.

# Multicast: Comunicação por grupos



# Comunicação por Grupos

- Multicasting é útil:
  - Tolerância a falhas baseada em servidores replicados
  - Aumentar o desempenho através da replicação de dados
  - Multiple update
- Modos de Comunicação:
  - One-to-Many
  - Many-to-One
  - Many-to-Many

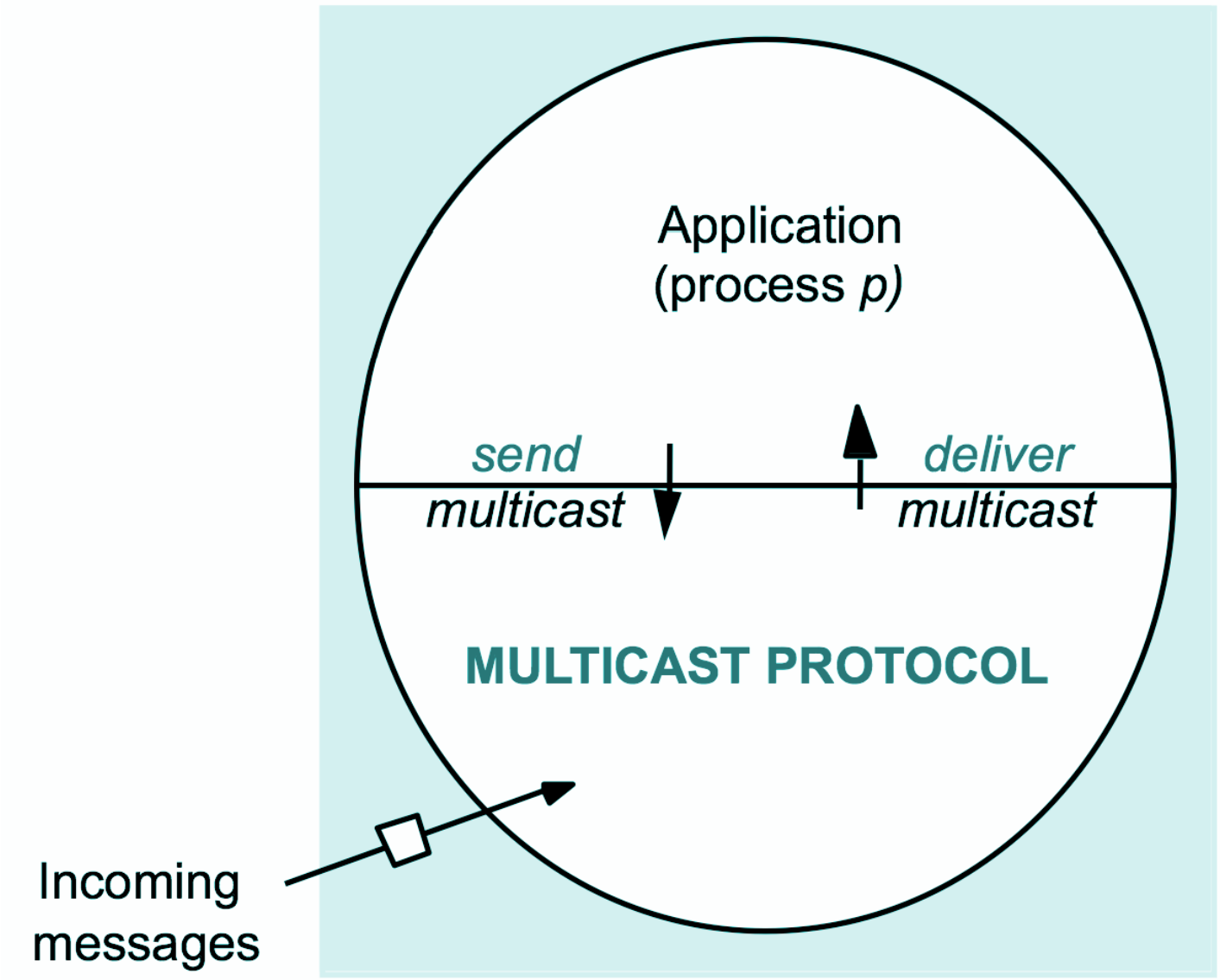
# Tipos de grupos

- Grupos Estáticos vs Grupos Dinâmicos
- Grupos Fechados vs Grupos Abertos
- Gestão do Grupo: Centralizada ou Distribuída
- A gestão do grupo requere um algoritmo de membership

# Fiabilidade na comunicação por grupos

- 0-reliable
- 1-reliable
- $m$-out-of-$n$ reliable
- All-reliable

# Multicast layer



# Reliable multicast

- **Integridade**: Um processo p faz a entrega de uma messagem m apenas uma vez (at-most-once). Isto implica que é obrigatório ter filtragem de duplicados.
- **Validade**: Se um processo faz o multicast de uma mensagem m, então eventualmente vai fazer a entrega dessa mensagem à aplicação.
- **Agreement**: Se um processo faz a entrega da mensagem m, então todos os outros processos no grupo irão fazer a entrega da mensagem m (propriedade de "all-or-nothing").

# Algoritmo de reliable multicast

- Começa-se por implementar uma primitiva *B-multicast*, e a correspondente primitiva *B-deliver*.
- Uma possibilidade é criar uma operação fiável de *send*, one-to-one, que aguarda sempre por um acknowledgement.

To *B-multicast*($g$, $m$): for each process $p \in g$ , *send*($p$, $m$);

On *receive*($m$) at $p$: *B-deliver*($m$) at $p$.

# Algoritmo de reliable multicast

*On initialization*
    *Received* := {};

*For process p to R-multicast message m to group g*
    *B-multicast*(*g*, *m*);  // $p \in g$ is included as a destination

*On B-deliver*(*m*) *at process q with g = group*(*m*)
    *if* ($m \notin$ *Received*)
    *then*
        *Received* := *Received* $\cup$ {*m*};
        *if* ($q \neq p$) *then B-multicast*(*g*, *m*); *end if*
        *R-deliver m*;
    *end if*

Este algoritmo garante as 3 propriedades, mas é muito ineficiente.
Cada mensagem é enviada $n$ vezes a cada processo ($n = sizeof(g)$).

# Garantia das propriedades

- **Integridade**: Decorre da integridade dos canais usados na primitiva *B-multicast* e da filtragem de duplicados.
- **Validade**: Um processo correcto irá certamente fazer *B-deliver* da mensagem a si próprio.
- **Agreement**: Cada processo correcto faz *B-multicast* da mensagem aos outros processos depois de ter feito *B-deliver*; se um processo correcto não faz *R-deliver* também não poderá ter feito *B-deliver*; portanto nenhum outro processo fez *B-deliver* (nem *R-deliver*).

# Algoritmo de reliable multicast mais eficiente

- Usar IP multicast, sempre que possível.
- Um processo não envia um ACK quando recebe uma mensagem.
  - Faz piggyback de ACKs em mensagens futuras.
  - Um processo só envia uma mensagem separada (NACK) quando detecta que não recebeu uma determinada mensagem.
  - NACK: Negative Acknowledgement
  - As mensagens devem ter um número de sequência para permitir a detecção de mensagens perdidas.
  - Isto evita o efeito de *ack-implosion* resultante da implementação da primitiva de *send*.
- Ver Sequence-Number no algoritmo FIFO, mais à frente

# Ordem na Entrega de Mensagens Multicast

- **FIFO: Se um processo envia uma mensagem *multicast(g,m)* e depois *multicast(g,m')* , então todos os outros processos vão fazer a entrega de m' depois de terem feito a entrega da mensagem m.**

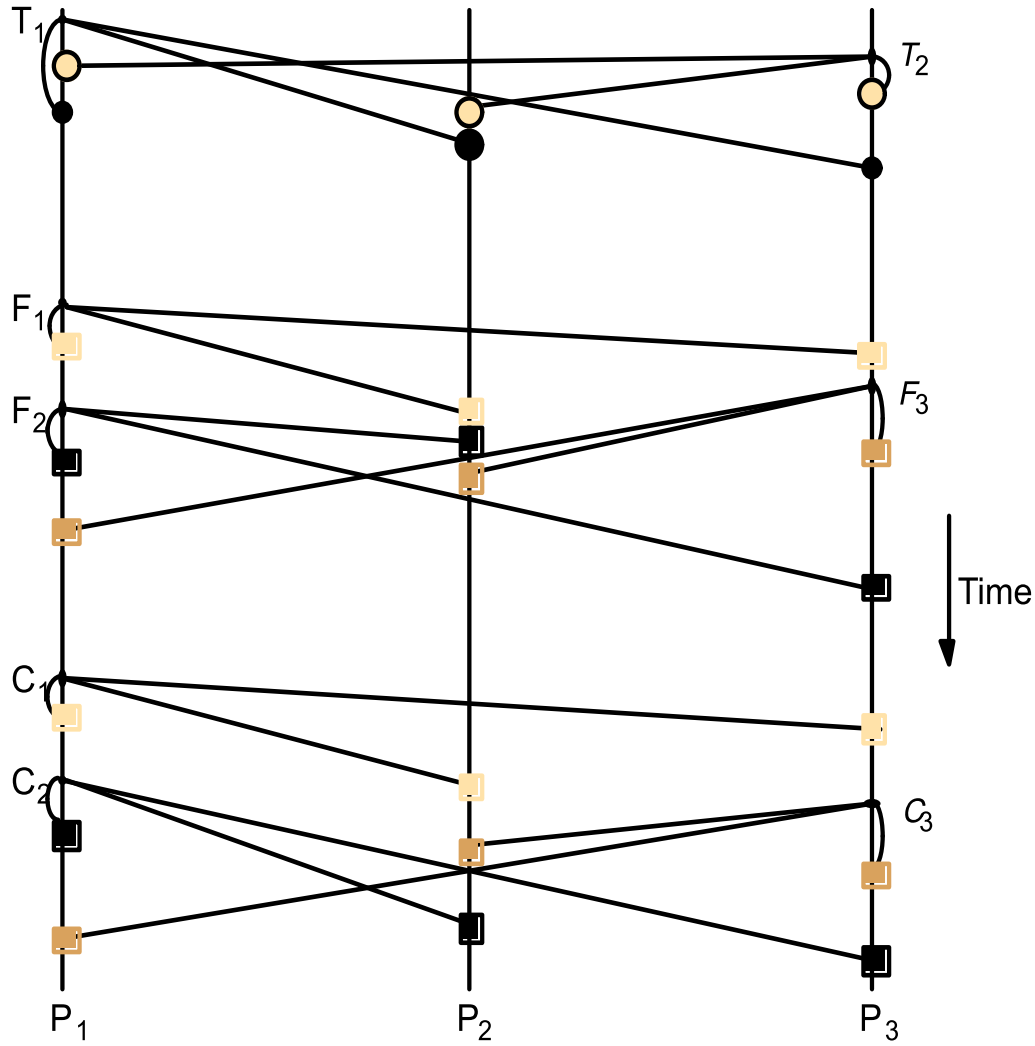
- **CAUSAL: Se *multicast(g,m)* → *multicast(g,m')* então todos os processos vão fazer a entrega de m' de pois da mensagem m.**

- **TOTAL: Se um processo faz a entrega de duas mensagens por esta ordem (m, m') então todos os restantes processos devem fazer a entrega exactamente pela mesma ordem.**

- **ABSOLUTA: corresponde à ordem total + tempo real.**

# Total, FIFO and Causal Ordering

- **Ordem Total: $T_1$ e $T_2$.**
- **Ordem FIFO: $F_1$ e $F_2$.**
- **Ordem Causal: $C_1$ e $C_3$**




Ordem Causal implica ordem FIFO.

Ordem Total não implica ordem Causal.

Ordem Causal não implica ordem total.

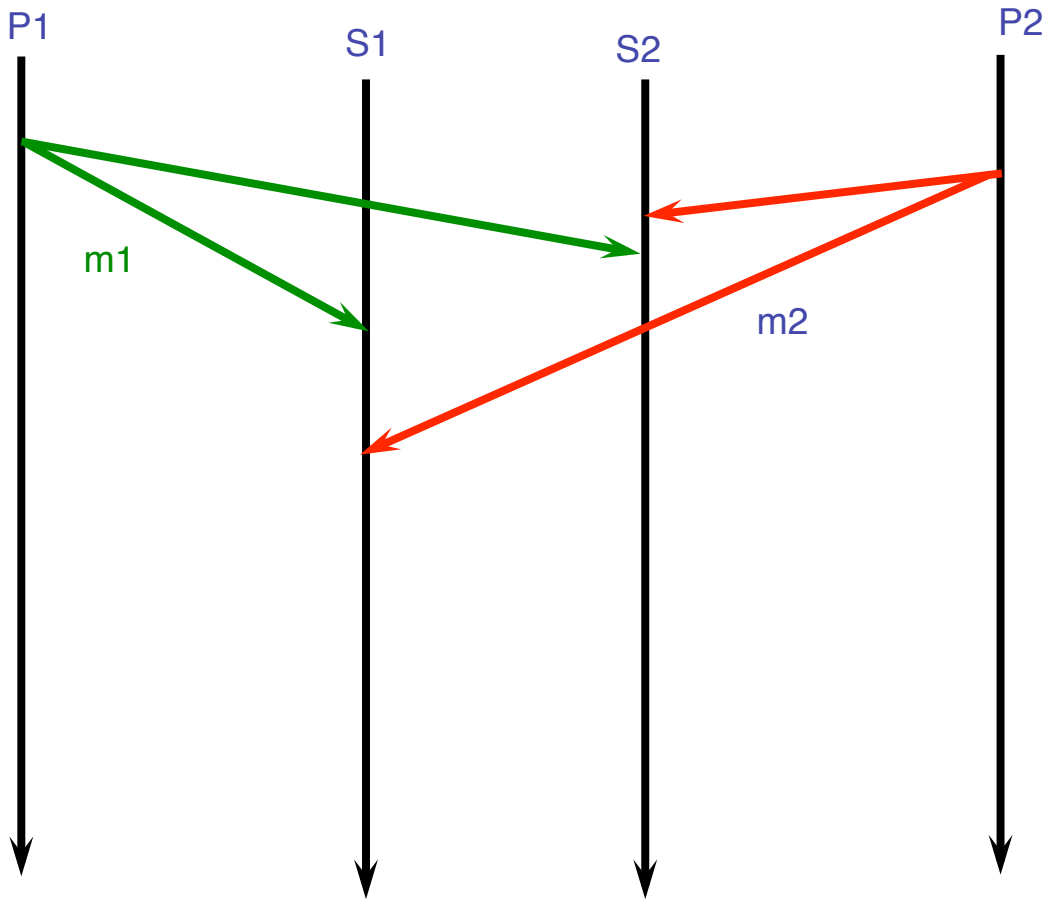
É possível ter: ordem causal-total e ordem FIFO-total.

# Bulletin Board: Qual a ordem a garantir?

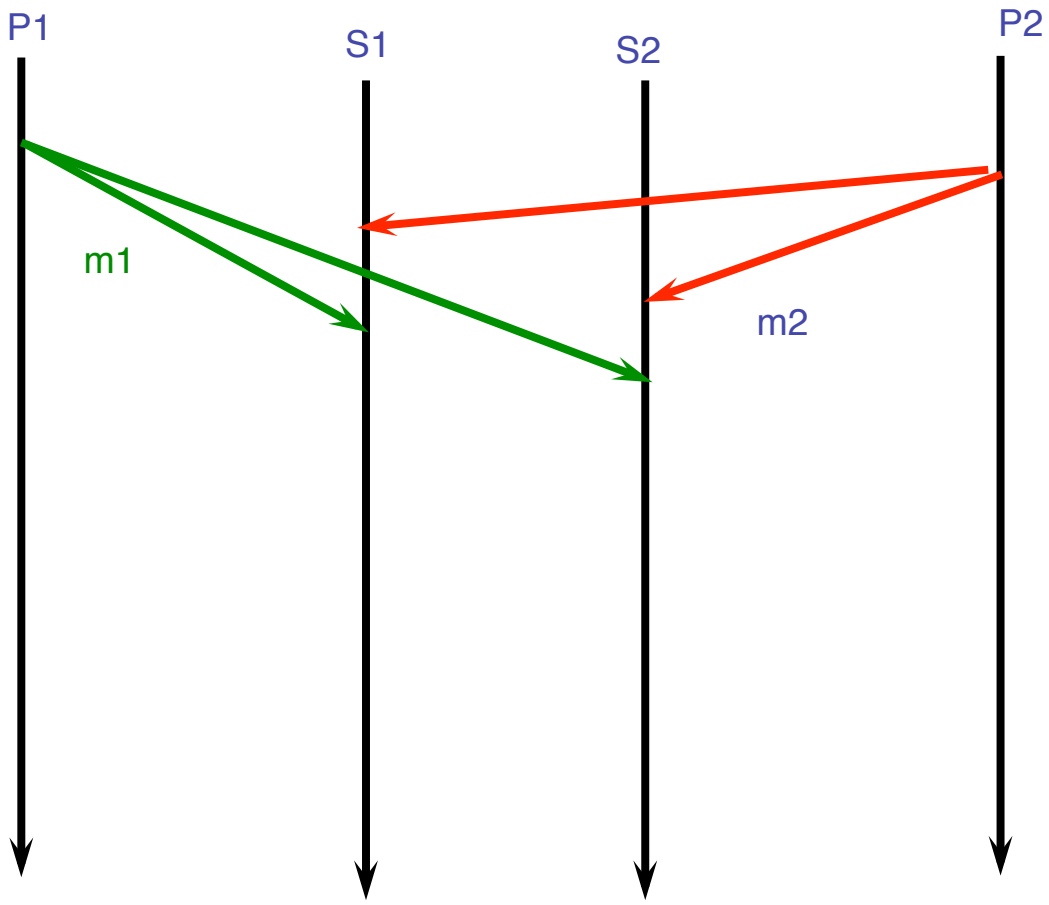
| Bulletin board: *os.interesting* | | |
|---|---|---|
| Item | From | Subject |
| 23 | A.Hanlon | Mach |
| 24 | G.Joseph | Microkernels |
| 25 | A.Hanlon | Re: Microkernels |
| 26 | T.L'Heureux | RPC performance |
| 27 | M.Walker | Re: Mach |
| end | | |

Qual a ordem mais apropriada para esta aplicação de Bulletin Board?
(a) FIFO (b) causal (c) total

# MCast que não garante a ordem total

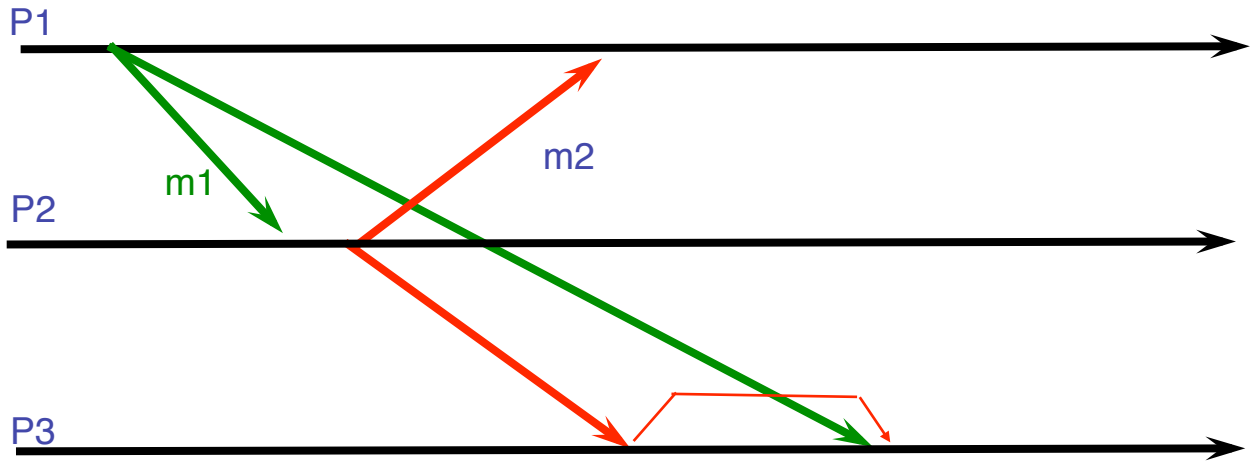


# MCast que garante a ordem total



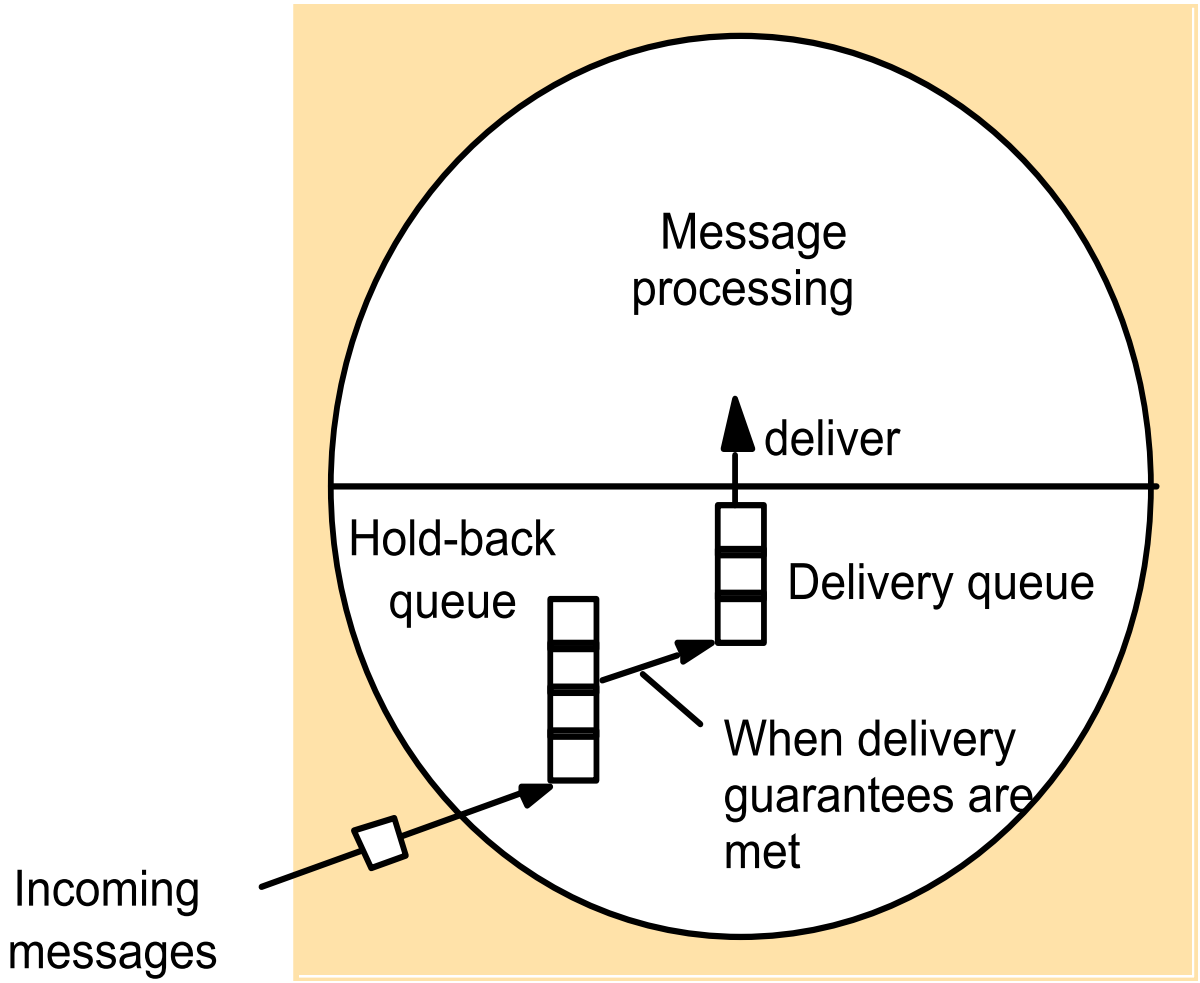

Os dois servidores recebem as mensagens pela mesma ordem

# Ordem Causal

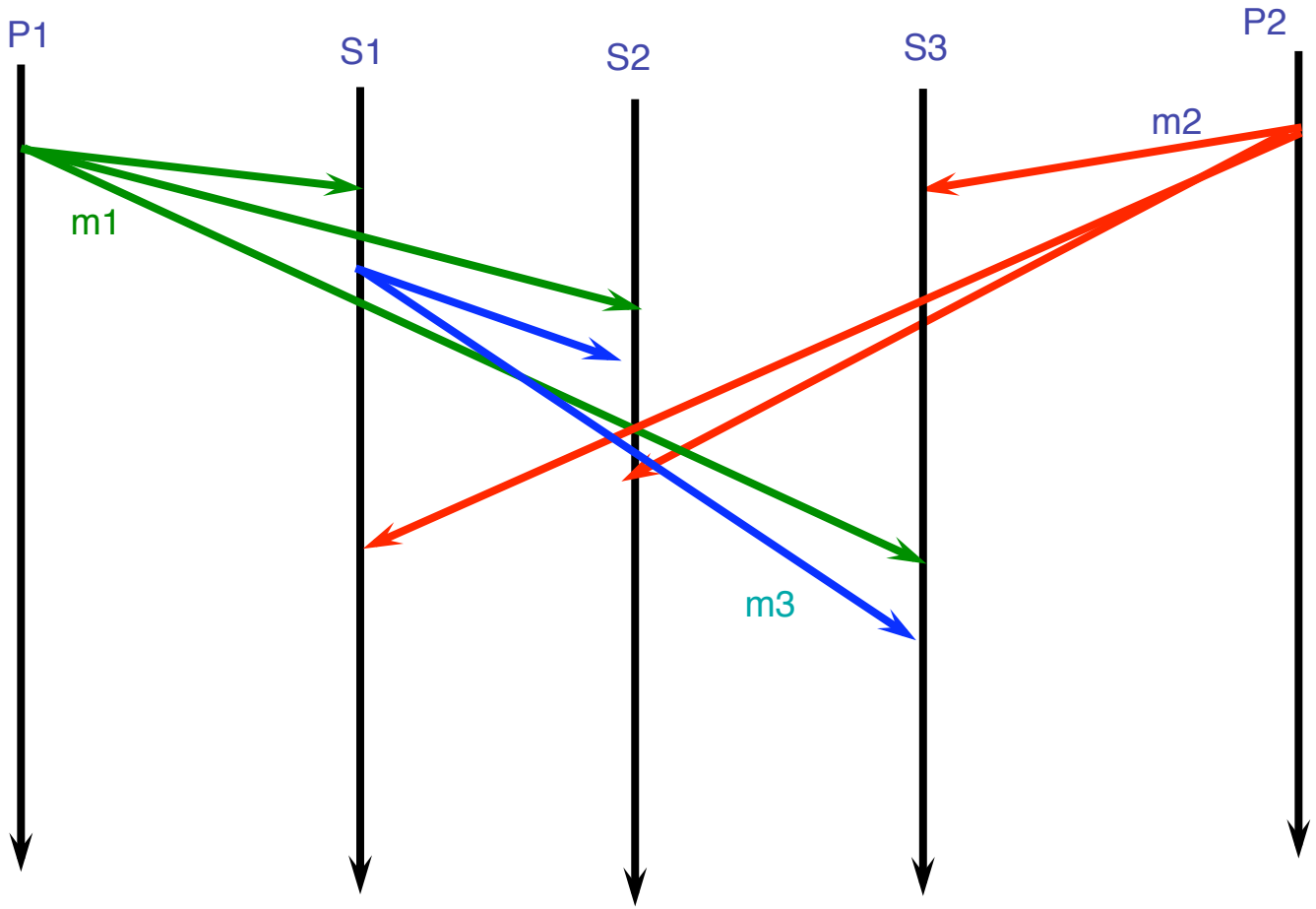



Para garantir a ordem causal, a mensagem m2 só pode ser entregue em P3 depois deste receber e entregar a mensagem m1.

# Hold-back Queue para as Mensagens que vão chegando



# Ordem Causal não implica Ordem Total




Ordem Causal: m1 → m3

# Ordem FIFO

- As mensagens são entregues pela ordem em que foram enviadas.
- Cada processo mantém um número de sequência para cada um dos outros processos.
- Quando uma mensagem chega:

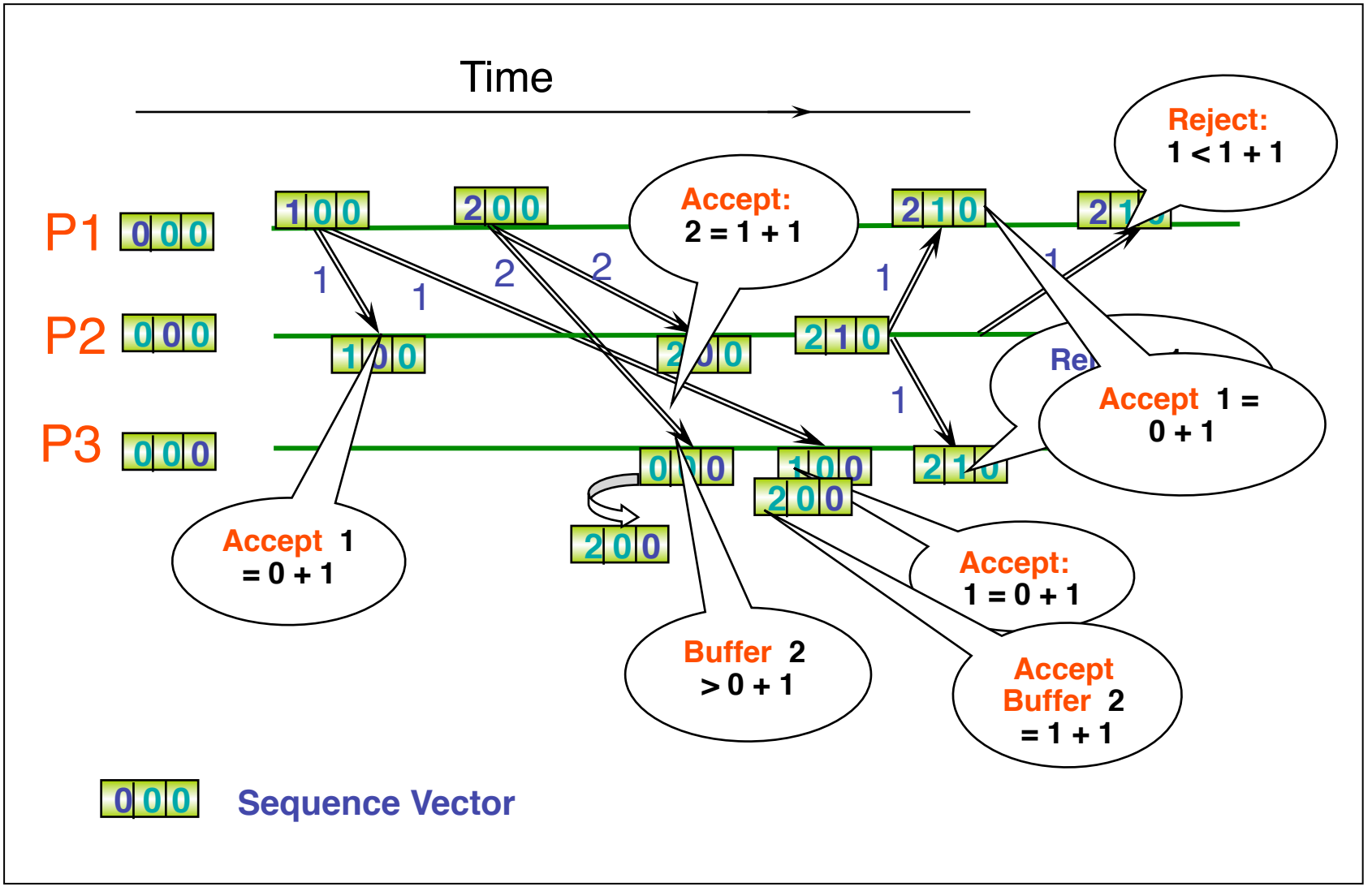
**If Message# is**

- Como esperado: ACEITA e ENTREGA
- Número maior do que esperado: GUARDA NA QUEUE
- Número menor do que esperado: REJEITA

# Algoritmo para Ordem FIFO

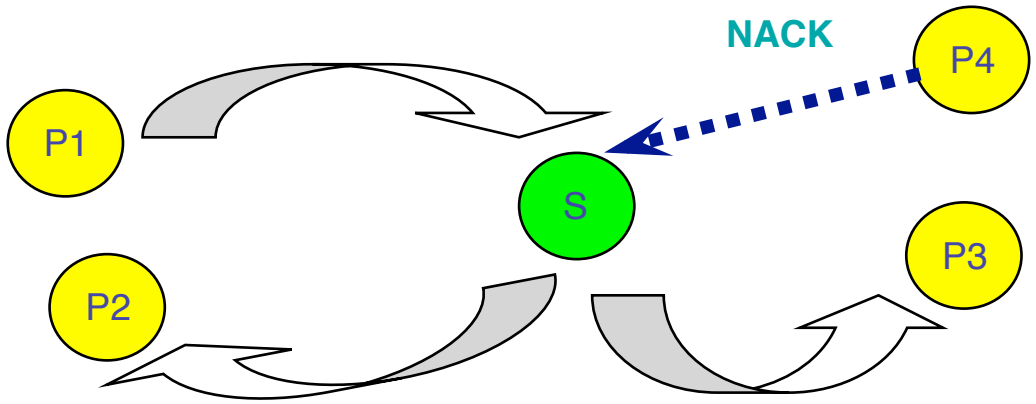
- $S^p_g$: the number of messages *p* has sent to *g*.
- $R^q_g$: the sequence number of the latest group-*g* message *p* has delivered from *q*.
- For *p* to FO-multicast *m* to *g*
  - *p increments* $S^p_g$ *by 1.*
  - p piggy backs the value $S^p_g$ *onto the message.*
  - *p B-multicasts m to g.*
- Upon receipt of *m* from *q* with sequence number *S*:
  - *p checks whether* $S = R^q_g + 1$. *If so, p FO-delivers m and increments* $R^q_g$
  - *If* $S > R^q_g + 1$, *p places the message in the hold-back queue until the intervening messages have been delivered and* $S = R^q_g + 1$.

# Example: FIFO Multicast




*(NÃO CONFUNDIR SEQUENCE_NUMBERS COM VECTOR_TIMESTAMPS)*

# Protocolo Ordem Total do Amoeba: usa um Sequenciador




**P1 envia a mensagem ao sequenciador. Este adiciona um Número de Sequência à mensagem e faz multicast. O número de sequência é usado para assegurar que as mensagens são entregues pela mesma ordem a todos os processos do grupo.**

**O sequenciador mantém a seguinte informação:**

- **lista dos processos do grupo**
- **contador (número de sequência)**
- **history buffer (com as mensagens que já foram enviadas)**

# Protocolo Amoeba: uso de NACKs

- Processos não usam ACKs explícitos.

- Processos usam NACKs quando percebem que não receberam uma mensagem. Percebem isso através da análise do número de sequência.
- Se existir um hiato no número de sequência enviam um NACK.

- Processos entregam as mensagens que respeitarem o número de sequência.

- O sequenciado quando recebe um NACK de um processo vai ao HISTORY_BUFFER buscar a mensagem e retransmite essa mensagem ao processo que enviou o NACK.

# Protocolo Amoeba:
# como evitar Buffer Overflow no Sequenciador

- O maior número de sequência recebido por um processo é sempre colocado (piggyback) em todas as mensagens multicast que envia.
  - O sequenciador pode encontrar o maior número já recebido por todos e limpa todas as mensagens até esse número.

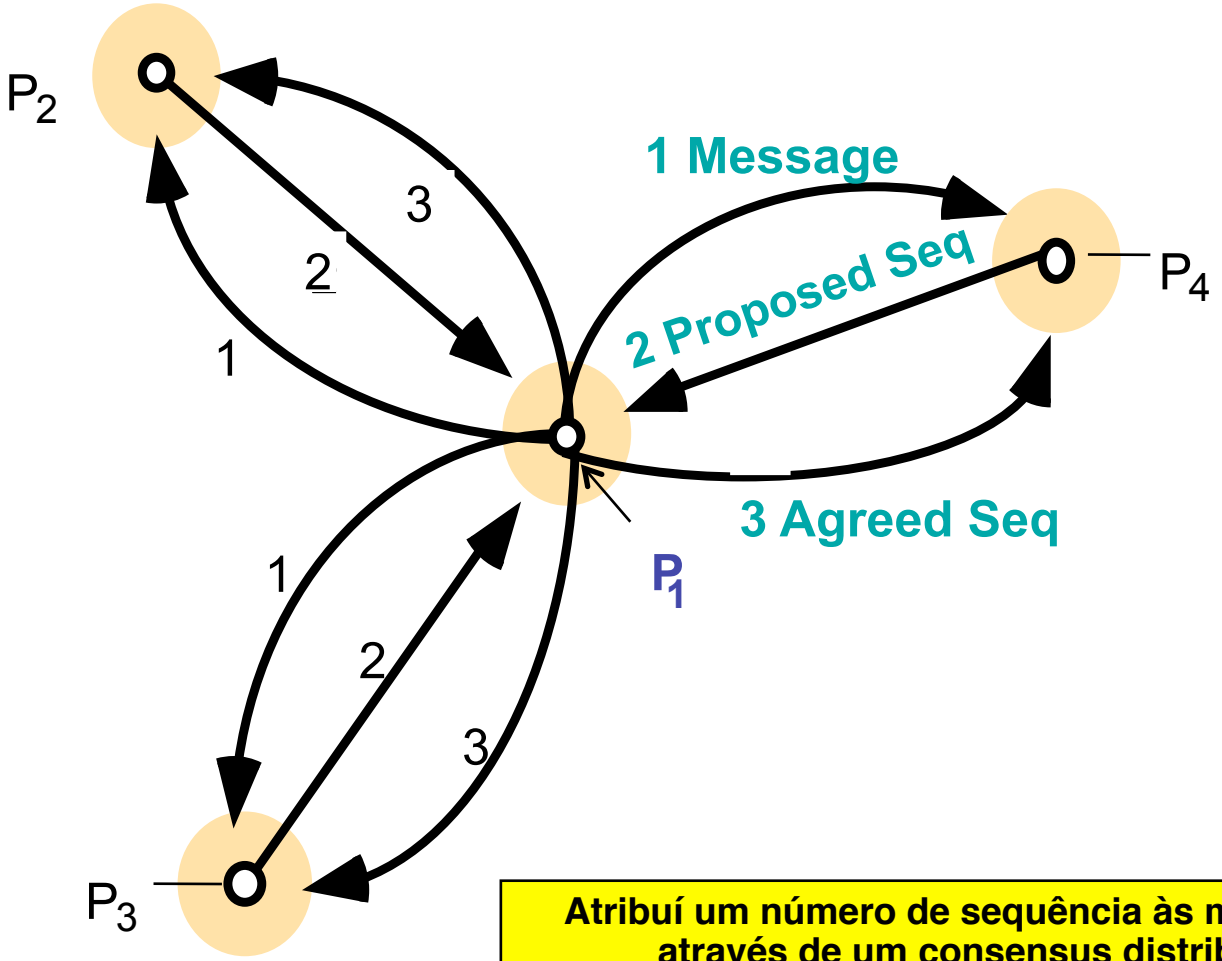
- Quando não há mensagens de multicast, os processos devem enviar umas mensagens do tipo "heartbeat" que incluem o maior número de sequência recebido.

- Se o tamanho do HISTORY_BUFFER exceder um determinado limite, o Sequenciador envia uma query a todos os processos do grupo para saber quais as mensagens que pode limpar.

# Limitações do Protocolo Amoeba

- Sequenciador: pode ser um bottleneck e um ponto central de falha.

# Protocolo ABCAST do ISIS para garantir Ordem Total




**Atribuí um número de sequência às mensagens através de um consensus distribuído.**

# Protocolo ABCAST: Ordem Total

- [1] Processo emissor atribuí um determinado número de sequência. Envia a mensagem com esse número para todos os membros do grupo. O número de sequência escolhido deve ser maior do que qualquer outro previamente escolhido.

- [2] Cada processo do grupo responde com uma proposta de número de sequência. O processo (Pi) calcula a sua proposta usando a seguinte função:

  max(Fmax, Pmax) + 1 + i/N;

  - N = número de processos do grupo
  - Fmax = maior identificador já obtido em consensus
  - Pmax = maior número proposto por Pi

# Protocolo ABCAST: Ordem Total

- [3] Quando o processo emissor recebe todas as propostas de todos os membros do grupo seleciona o maior número de sequência e envia-o para todos os processos numa mensagem do tipo “COMMIT”.

  O número final é único devido ao termo (i/N) na fórmula.

- [4] Quando um membro do grupo recebe a mensagem de “COMMIT” coloca o número final de sequência na mensagem multicast.

- [5] As mensagens COMMITED são entregues à aplicação pela ordem do número de sequência.

# Protocolo ABCAST do ISIS

- Não tem um ponto central de falha e congestão.
- Protocolo distribuído.
- Mas é um protocolo pesado em termos de mensagens que são usadas para obter os números de sequência.

- O algoritmo do AMOEBA é mais rápido do que este e usa menos mensagens.

# Protocolo Ordem Causal: CBCAST (ISIS)
# Faz uso de Vector Timestamps

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

*On initialization*

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

VTi[k]: número do último evento Executado por Pk e conhecido por Pi.

*To CO-multicast message m to group g*

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

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

Faz o piggyback do vector Vi na mensagem

*On B-deliver*$(<V_j^g, m>)$ *from* $p_j$, *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 $V_j^g[k] \leq V_i^g[k]$ $(k \neq j)$;

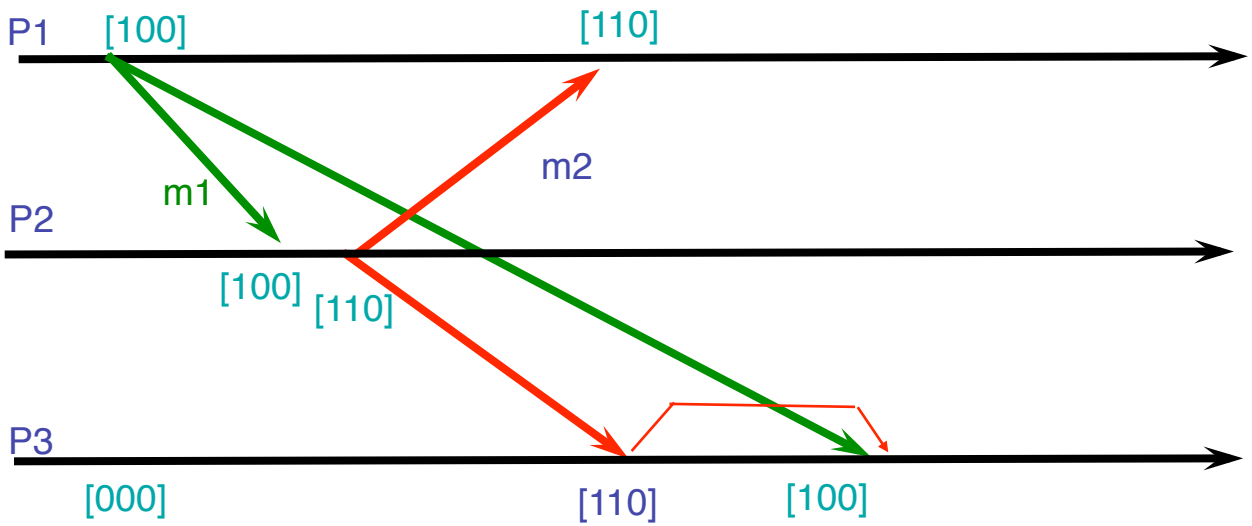
*CO-deliver m*; // after removing it from the hold-back queue

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

Para a mensagem respeitar a ordem causal tem que verificar os seguintes critérios:

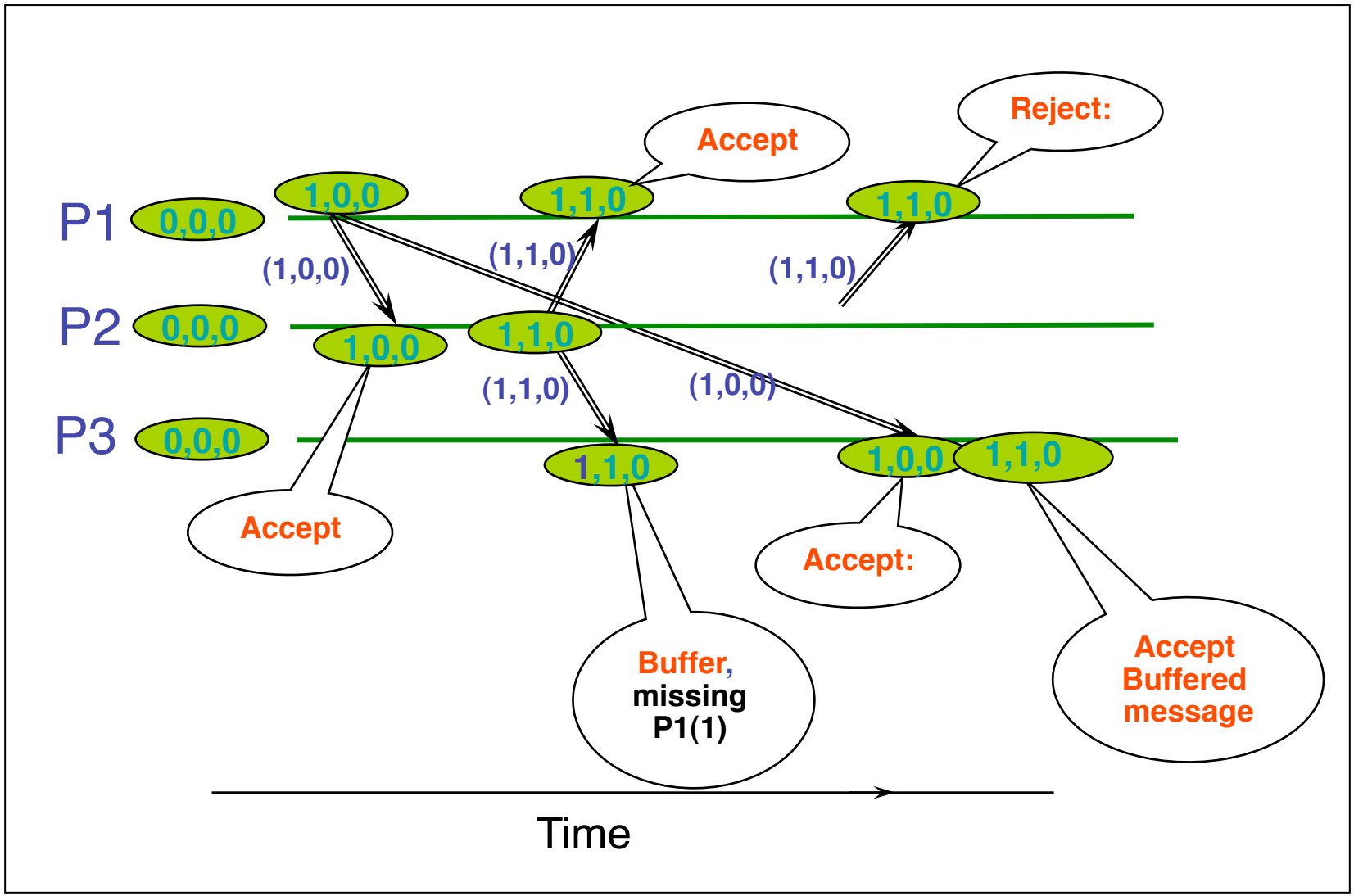
- VTmsg[i] = VTk[i] + 1
- VTk[j] >= VTmsg[j], j != i

# Ordem Causal




VTmsg[1] > VT3[1] ?...

# Causal Order Multicast



# Sumário

- Comunicação Multicast
- Reliable Multicast
- Ordem FIFO
- Ordem TOTAL
- Ordem CAUSAL

# Bibliography


- Coulouris *et al.*, Distributed Systems: Concepts and Design, 5th Edition, Chapter 14.4.
- Coulouris *et al.*, Distributed Systems: Concepts and Design, 5th Edition, Chapter 15.4.