ACH 2147 — Desenvolvimento de Sistemas de Informação Distribuídos

Aula 13: Coordenação (parte 2)

Prof. Renan Alves

Escola de Artes, Ciências e Humanidades — EACH — USP

15/04/2024

A relação do tipo ”ocorreu-antes” (happened-before)

Questão

O que geralmente importa não é que todos os processos concordem exatamente com que hora é, mas que eles concordem com a ordem em que os eventos ocorrem. Requer uma noção de ordenação.

A relação do tipo ”ocorreu-antes” (happened-before)

Questão

O que geralmente importa não é que todos os processos concordem exatamente com que hora é, mas que eles concordem com a ordem em que os eventos ocorrem. Requer uma noção de ordenação.

A relação do tipo ocorreu-antes

- Se a e b são dois eventos de um mesmo processo, e a vem antes de b, então $a \rightarrow b$.
- Se a é o envio de uma mensagem e b é o recebimento dessa mensagem, então $a \rightarrow b$.
- Se $a \rightarrow b$ e $b \rightarrow c$, então $a \rightarrow c$.

Nota

Isso introduz uma ordem parcial de eventos em um sistema com processos operando concorrentemente.

Relógios lógicos

Problema

Como podemos manter uma visão global do comportamento do sistema que seja consistente com a relação de ocorreu-antes?

Relógios lógicos

Problema

Como podemos manter uma visão global do comportamento do sistema que seja consistente com a relação de ocorreu-antes?

Associar um timestamp $C(e)$ a cada evento e, satisfazendo as seguintes propriedades:

- P1 Se a e b são dois eventos no mesmo processo, e $a \rightarrow b$, então exigimos que $C(a) < C(b)$.
- P2 Se a corresponde ao envio de uma mensagem m, e b ao recebimento dessa mensagem, então também $C(a) < C(b)$.

Relógios lógicos

Problema

Como podemos manter uma visão global do comportamento do sistema que seja consistente com a relação de ocorreu-antes?

Associar um timestamp $C(e)$ a cada evento e, satisfazendo as seguintes propriedades:

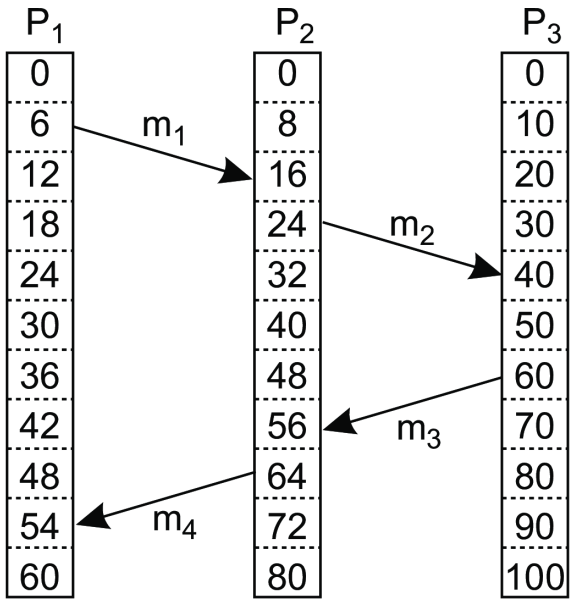
- P1 Se a e b são dois eventos no mesmo processo, e $a \rightarrow b$, então exigimos que $C(a) < C(b)$.
- P2 Se a corresponde ao envio de uma mensagem m, e b ao recebimento dessa mensagem, então também $C(a) < C(b)$.

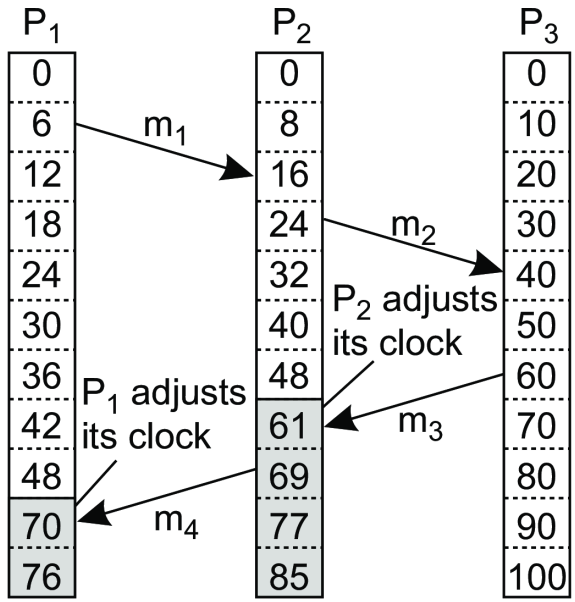
Problema

Como associar um timestamp a um evento quando não há relógio global $\Rightarrow$ manter um conjunto consistente de relógios lógicos, um por processo.

Relógios lógicos: ideia básica

Considere três processos com contadores de eventos operando em taxas diferentes





Relógios lógicos: solução

Cada processo P_i mantém um contador local C_i e:

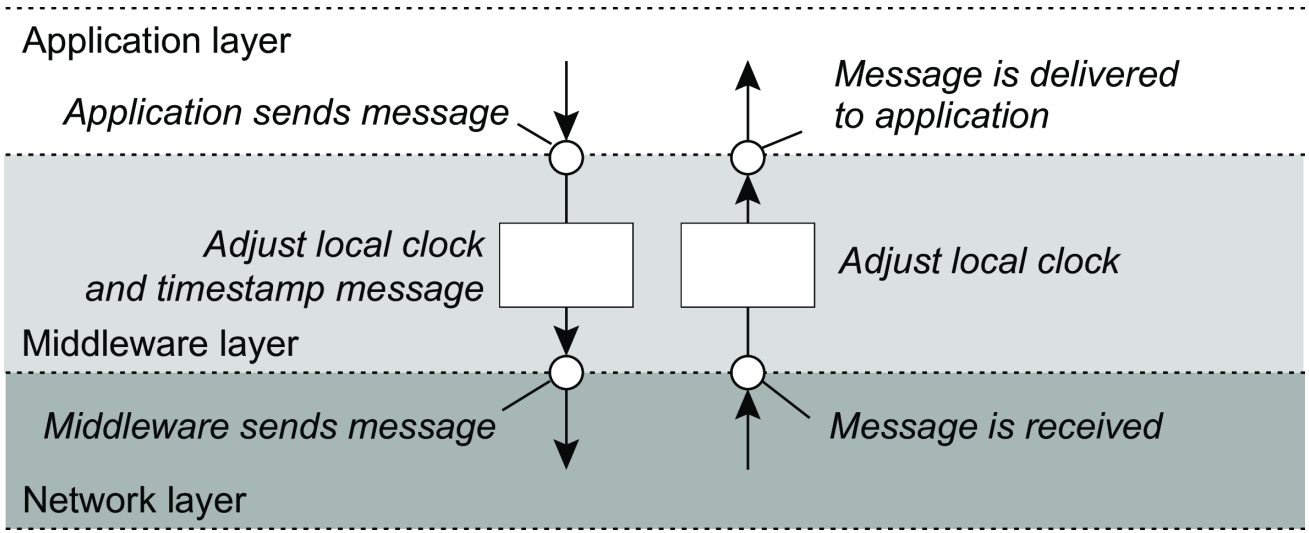
1. Para cada novo evento que ocorre dentro de P_i, C_i é incrementado em 1.
2. Cada vez que uma mensagem m é enviada pelo processo P_i, a mensagem recebe um timestamp $ts(m) = C_i$.
3. Sempre que uma mensagem m é recebida por um processo P_j, P_j ajusta seu contador local C_j para $\max\{C_j, ts(m)\}$; então executa o passo 1 antes de passar m para a aplicação.

Notas

- A propriedade P1 é satisfeita por (1); a propriedade P2 por (2) e (3).
- Ainda pode ocorrer que dois eventos aconteçam ao mesmo tempo. É possível desempatar usando os PIDs.

Relógios lógicos: onde são implementados

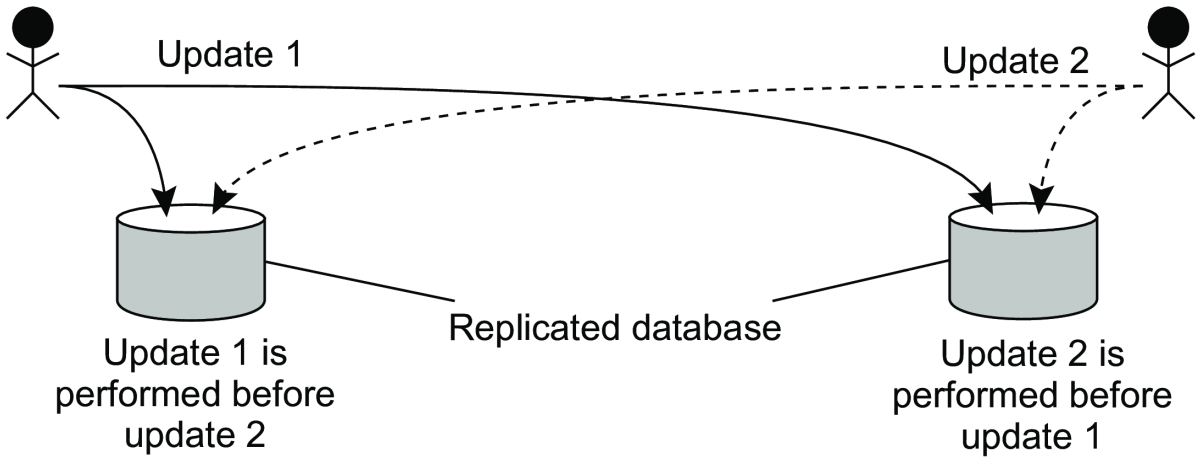
Os ajustes de relógio são implementados no middleware



Exemplo: Multicast completamente ordenado

Atualizações concorrentes em um banco de dados replicado são vistas na mesma ordem em todos os lugares

- P_1 adiciona \$100 a uma conta (valor inicial: \$1000)
- P_2 incrementa a conta em 1%
- Existem duas réplicas



Resultado

Na ausência de uma sincronização adequada:
réplica #1 $\leftarrow$ \$1111, enquanto na réplica #2 $\leftarrow$ \$1110.

Exemplo: Multicast completamente ordenado

Solução

- O processo P_i envia uma mensagem com timestamp m_i para todos os outros. A mensagem em si é colocada na fila local $queue_i$.
- Qualquer mensagem recebida em P_j é enfileirada em sua $queue_j$, de acordo com seu timestamp. Uma confirmação de recebimento é enviada para cada um dos outros processos.

Exemplo: Multicast completamente ordenado

Solução

- O processo P_i envia uma mensagem com timestamp m_i para todos os outros. A mensagem em si é colocada na fila local $queue_i$.
- Qualquer mensagem recebida em P_j é enfileirada em sua $queue_j$, de acordo com seu timestamp. Uma confirmação de recebimento é enviada para cada um dos outros processos.

P_j passa uma mensagem m_i para sua aplicação se:

(1) m_i for a primeira de $queue_j$
(2) para cada outro processo P_k, há uma mensagem m_k na $queue_j$ com um timestamp maior (pode ser uma mensagem de confirmação).

Exemplo: Multicast completamente ordenado

Solução

- O processo P_i envia uma mensagem com timestamp m_i para todos os outros. A mensagem em si é colocada na fila local $queue_i$.
- Qualquer mensagem recebida em P_j é enfileirada em sua $queue_j$, de acordo com seu timestamp. Uma confirmação de recebimento é enviada para cada um dos outros processos.

P_j passa uma mensagem m_i para sua aplicação se:

(1) m_i for a primeira de $queue_j$
(2) para cada outro processo P_k, há uma mensagem m_k na $queue_j$ com um timestamp maior (pode ser uma mensagem de confirmação).

Nota

Estamos assumindo que a comunicação é confiável e ordenada (FIFO).

Relógios de Lamport para exclusão mútua

```
1  class Process:
2    def __init__(self, chanID, procID, procIDSet):
3      self.chan.join(procID)
4      self.procID     = int(procID)
5      self.otherProcs.remove(self.procID)
6      self.queue      = []                        # The request queue
7      self.clock      = 0                         # The current logical clock
8
9    def requestToEnter(self):
10     self.clock = self.clock + 1                             # Increment clock value
11     self.queue.append((self.clock, self.procID, ENTER)) # Append request to q
12     self.cleanupQ()                                         # Sort the queue
13     self.chan.sendTo(self.otherProcs, (self.clock, self.procID, ENTER)) # Send request
14
15   def ackToEnter(self, requester):
16     self.clock = self.clock + 1                             # Increment clock value
17     self.chan.sendTo(requester, (self.clock, self.procID, ACK)) # Permit other
18
19   def release(self):
20     tmp = [r for r in self.queue[1:] if r[2] == ENTER]  # Remove all ACKs
21     self.queue = tmp                                        # and copy to new queue
22     self.clock = self.clock + 1                             # Increment clock value
23     self.chan.sendTo(self.otherProcs, (self.clock, self.procID, RELEASE)) # Release
24
25   def allowedToEnter(self):
26     commProcs = set([req[1] for req in self.queue[1:]]) # See who has sent a message
27     return (self.queue[0][1] == self.procID and len(self.otherProcs) == len(commProcs))
```

Relógios de Lamport para exclusão mútua

```
1   def receive(self):
2     msg = self.chan.recvFrom(self.otherProcs)[1]      # Pick up any message
3     self.clock = max(self.clock, msg[0])              # Adjust clock value...
4     self.clock = self.clock + 1                       # ...and increment
5     if msg[2] == ENTER:
6       self.queue.append(msg)                          # Append an ENTER request
7       self.ackToEnter(msg[1])                         # and unconditionally allow
8     elif msg[2] == ACK:
9       self.queue.append(msg)                          # Append a received ACK
10    elif msg[2] == RELEASE:
11      del(self.queue[0])                              # Just remove first message
12    self.cleanupQ()                                   # And sort and cleanup
```

Relógios de Lamport para exclusão mútua

Analogia com multicast completamente ordenado

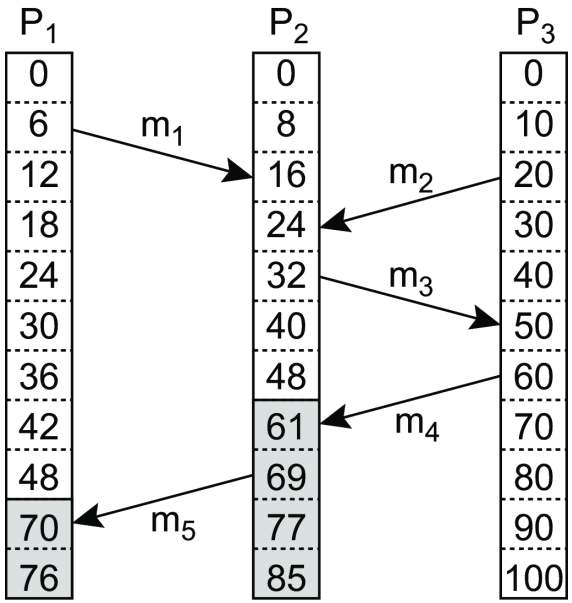
- Com o multicast completamente ordenado, todos os processos constroem filas idênticas, entregando mensagens na mesma ordem
- A exclusão mútua refere-se ao acordo sobre a ordem em que os processos podem entrar em uma região crítica

Relógios vetoriais

Observação

Os relógios de Lamport não garantem que *a* precedeu causalmente *b*, mesmo se a condição $C(a) < C(b)$ for verdadeira

Transmissão de mensagens concorrentes usando relógios lógicos



Observação 1

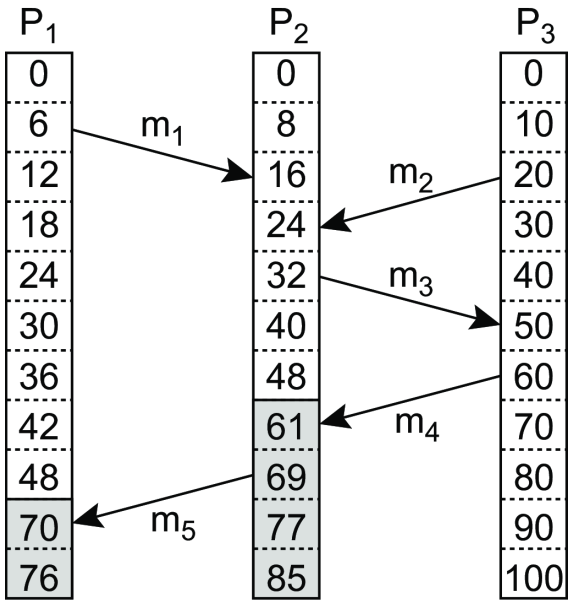
Evento *a*: m_1 foi recebida em $T = 16$;
Evento *b*: m_2 foi enviada em $T = 20$.

Relógios vetoriais

Observação

Os relógios de Lamport não garantem que *a* precedeu causalmente *b*, mesmo se a condição $C(a) < C(b)$ for verdadeira

Transmissão de mensagens concorrentes usando relógios lógicos



Observação 1

Evento *a*: m_1 foi recebida em $T = 16$;
Evento *b*: m_2 foi enviada em $T = 20$.

Observação 2

$C(a) < C(b)$, porém não podemos concluir que *a* precede causalmente *b*.

Capturando causalidade potencial

Solução: cada P_i mantém um vetor VC_i

- $VC_i[i]$ é o relógio lógico local no processo P_i.
- Se $VC_i[j] = k$ então P_i sabe que k eventos ocorreram em P_j.

Mantendo relógios vetoriais

1. Antes de executar um evento, P_i executa $VC_i[i] \leftarrow VC_i[i] + 1$.
2. Quando o processo P_i envia uma mensagem m para P_j, ele define o timestamp (vetorial) de m $ts(m)$ igual a VC_i, após ter executado o passo 1.
3. Ao receber uma mensagem m, o processo P_j faz $VC_j[k] \leftarrow \max\{VC_j[k], ts(m)[k]\}$ para cada k, executa o passo 1, e por fim entrega a mensagem para a aplicação.

Dependência causal

Definição

Dize-se que b pode depender causalmente de a se $ts(a) < ts(b)$, com:

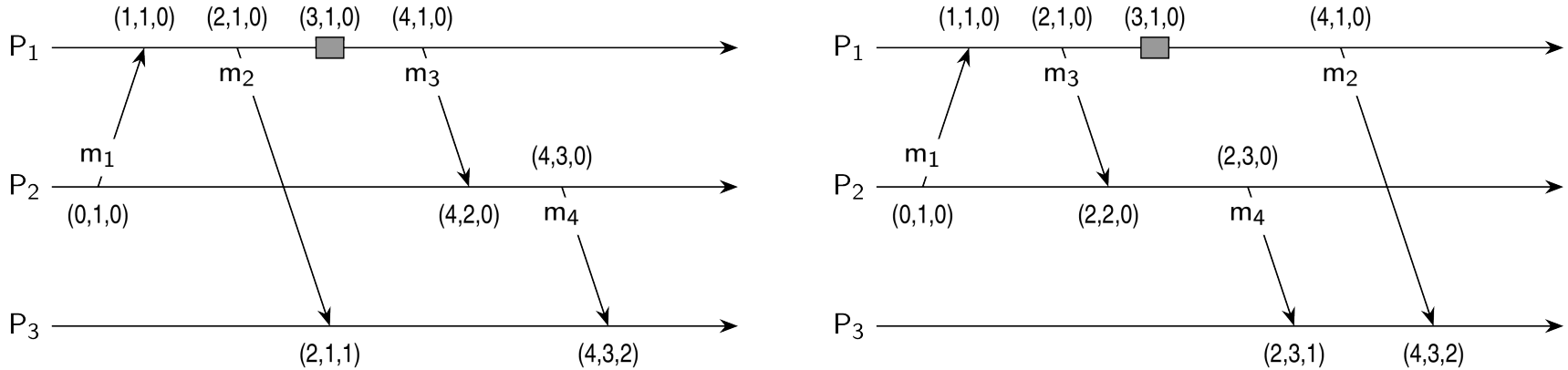
- para todo k, $ts(a)[k] \leq ts(b)[k]$ e
- existe pelo menos um índice k' para o qual $ts(a)[k'] < ts(b)[k']$

Precedência vs. dependência

- Dizemos que a precede causalmente b.
- b pode depender causalmente de a, pois pode haver informações de a que são propagadas para b.

Relógios vetoriais: Exemplo

Capturando causalidade potencial ao trocar mensagens



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

Multicast ordenado causalmente

Observação

Agora podemos garantir que uma mensagem é entregue apenas se todas as mensagens causalmente anteriores já foram entregues.

Ajuste

P_i incrementa $VC_i[i]$ apenas ao enviar uma mensagem, e P_j ”ajusta” VC_j ao receber uma mensagem (ou seja, efetivamente não altera $VC_j[j]$).

Multicast ordenado causalmente

Observação

Agora podemos garantir que uma mensagem é entregue apenas se todas as mensagens causalmente anteriores já foram entregues.

Ajuste

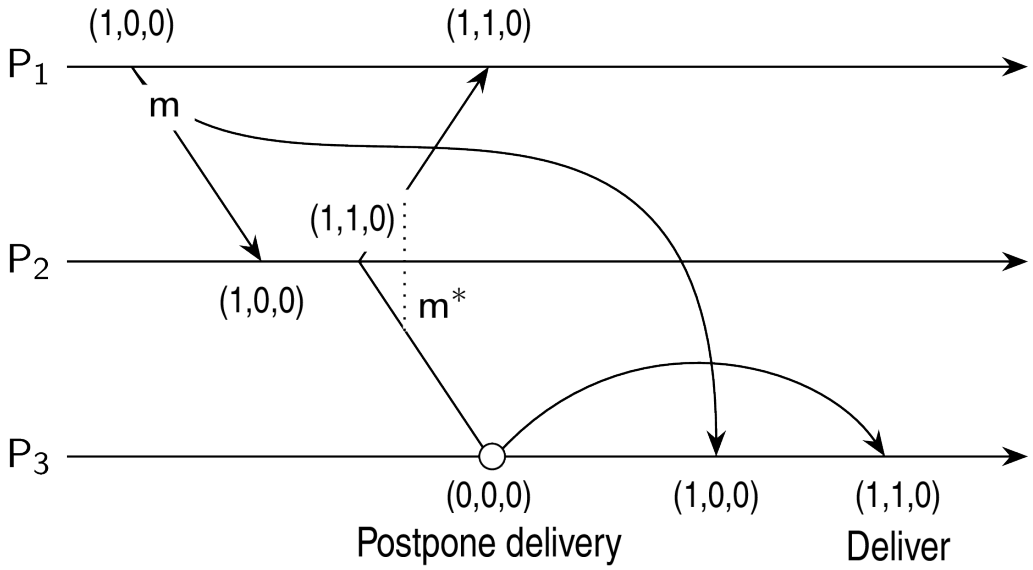
P_i incrementa $VC_i[i]$ apenas ao enviar uma mensagem, e P_j "ajusta" VC_j ao receber uma mensagem (ou seja, efetivamente não altera $VC_j[j]$).

P_j adia a entrega de m até que:

1. $ts(m)[i] = VC_j[i] + 1$
2. $ts(m)[k] \leq VC_j[k]$ para todos os $k \neq i$

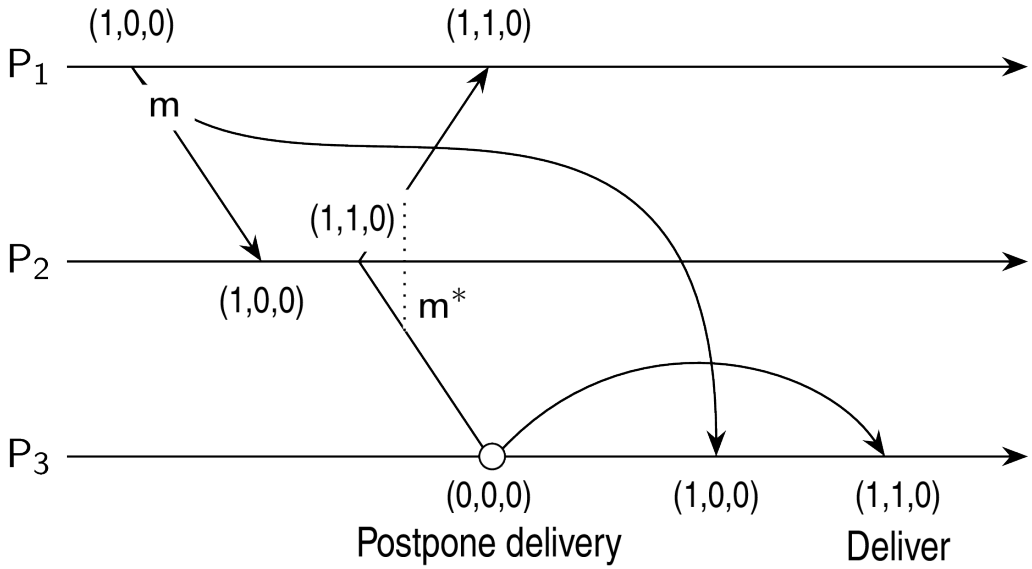
Multicast ordenado causalmente

Realizando comunicação causal



Multicast ordenado causalmente

Realizando comunicação causal



Exemplo

Partindo da situação em que $VC_3 = [0,2,2], ts(m) = [1,3,0]$ de P_1.
Que informações P_3 possui, e o que fará ao receber m (de P_1)?