

Sistemas Distribuídos

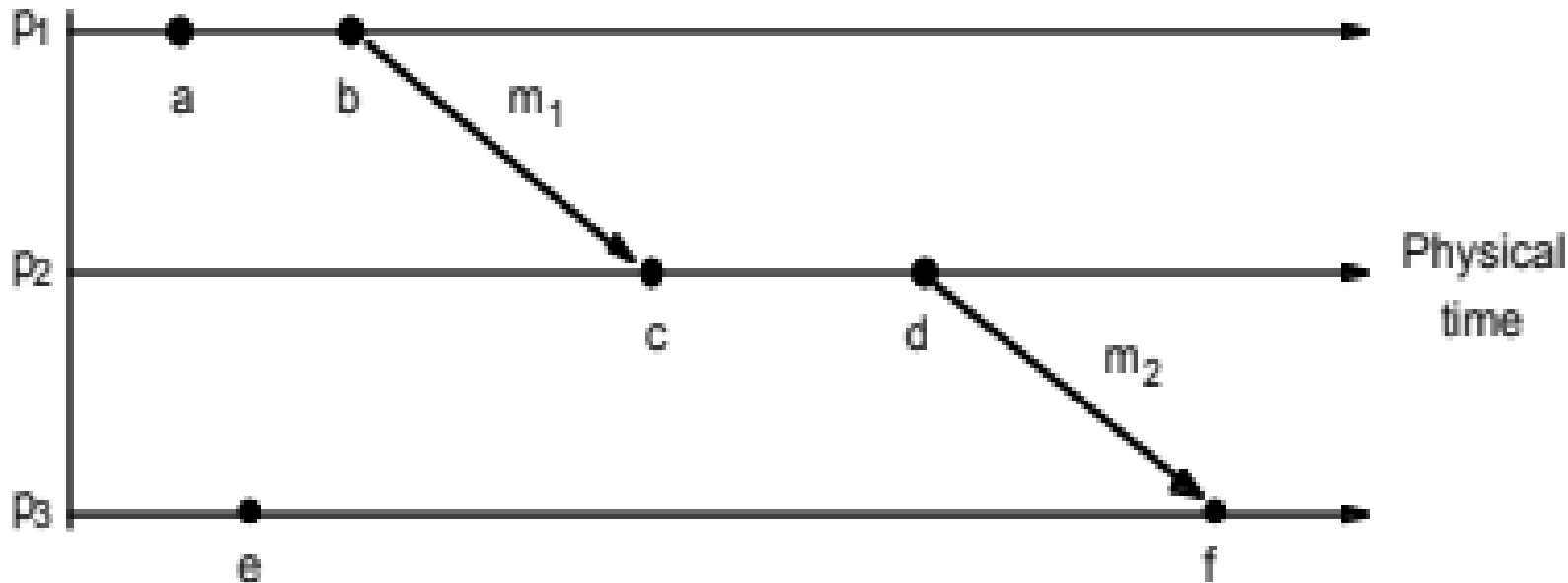
Aula 15

Roteiro

- Limitação de Lamport
- Relógio de vetores
- Propriedades
- Garantindo ordenação total
- *Totally Ordered Multicast*

Relógio de Lamport

- Algoritmo de relógio lógico distribuído para inferir relação entre eventos
 - se $e \rightarrow e'$, então $L(e) < L(e')$
 - se $L(e) = L(e')$, então $e \parallel e'$



- Mensagens e eventos definidos pelo sistema distribuído

Limitação de Lamport

- Não podemos inferir ordem dos eventos a partir dos valores dos relógios
 - $L(e) < L(e')$ não implica $e \rightarrow e'$
- Como garantir ordem parcial a partir dos valores de relógio?
 - $L(e) < L(e')$ implica $e \rightarrow e'$
- **Ideia:** adicionar informação no relógio lógico utilizado para rotular eventos
 - usar um vetor, ao invés de um número

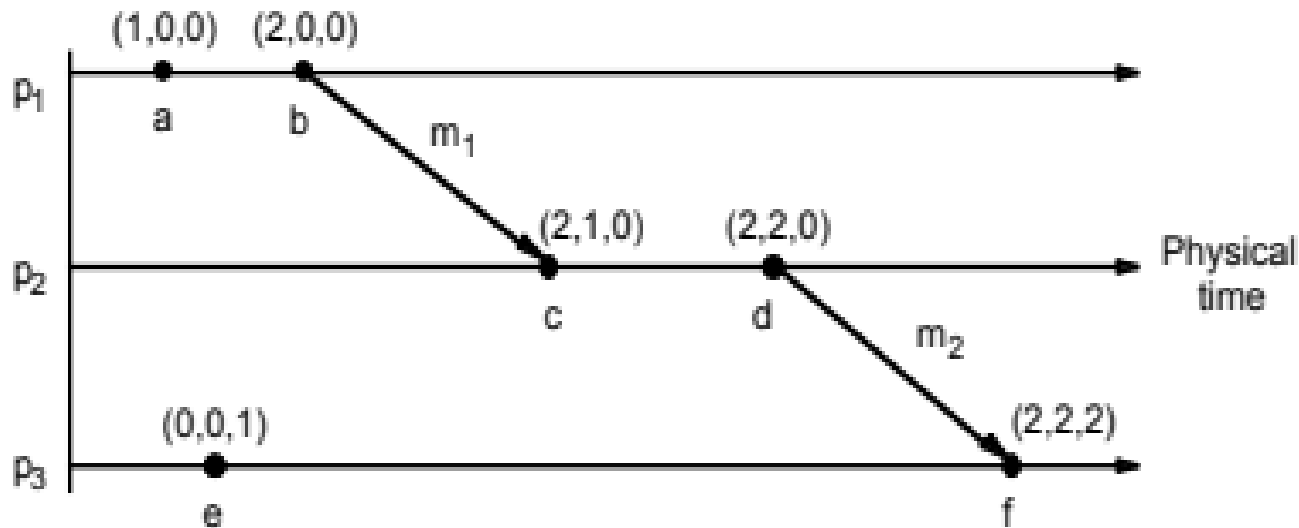
Relógio de Vetor

- Cada processo mantém um vetor
- Cada componente do vetor associado a um processo
 - dimensão do vetor é o número de processos, n
- Cada evento local rotulado com vetor local
- Evento e no processo j , $V(e) = [c_1, c_2, c_3, \dots, c_n]$
 - $[c_1, c_2, c_3, \dots, c_n]$: vetor do processo j
 - $c_i =$ valor do relógio no processo i que o processo j conhece (não necessariamente é o valor atual do relógio em i)
- Inicialmente, todos processos com vetor em 0

Algoritmo do Relógio de Vetor

- Para cada evento local em i , incrementa próprio c_i
 - receber mensagem é evento
- Para cada mensagem, transmite vetor atual
- Ao receber mensagem com vetor $[d_1, d_2, \dots, d_n]$, processo i faz
 - ajusta vetor para $c_k = \max(c_k, d_k)$ para todo k
 - incrementa c_i

Exemplo de Relógio de Vetor



- $V(a) = (1,0,0)$, $V(e) = (0,0,1)$, $V(b) = (2,0,0)$
- m_1 contém $(2,0,0)$
- $V(c) = \max\{ (0,0,0) , (2,0,0) \} + (0,1,0) = (2,1,0)$
- m_2 contém $(2,2,0)$
- $V(f) = \max\{ (0,0,1) , (2,2,0) \} + (0,0,1) = (2,2,2)$

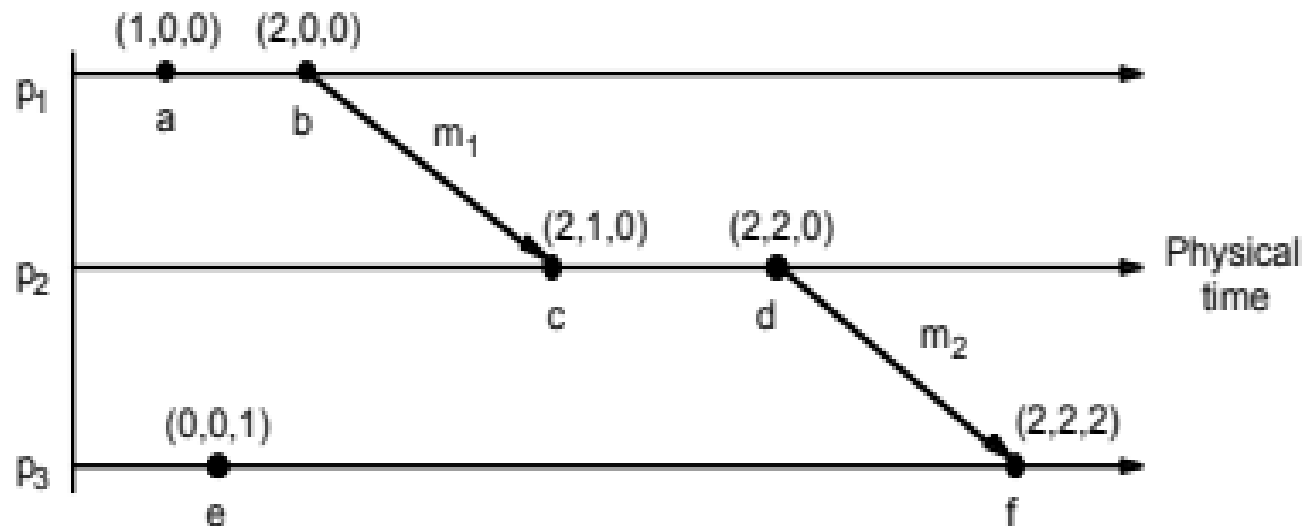
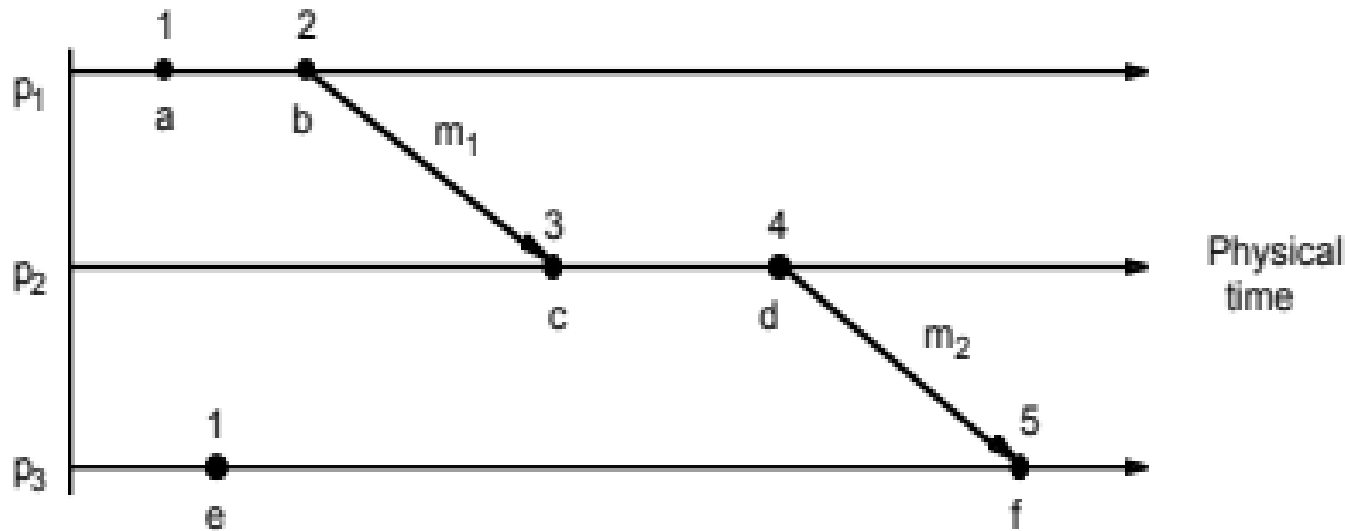
Comparando Vetores

- Cada evento está associado a um vetor
 - valor de relógio do evento
- Comparação componente a componente
 - Dizemos que $V(e) < V(e')$ se
 - (i) $V(e)[k] \leq V(e')[k]$ para todo k
 - (ii) existe z tal que $V(e)[z] < V(e')[z]$ (ao menos uma componente estritamente menor)
- Agora temos a propriedade
 - se $V(e) < V(e')$ então $e \rightarrow e'$
 - podemos usar o valor do tempo lógico do evento para resgatar a relação de “ocorreu antes”

Comparando Valores

- Definição de igualdade entre relógios
 - Dizemos que $V(e) = V(e')$ se
 - (i) não for o caso de $V(e) < V(e')$
 - (ii) não for o caso de $V(e') < V(e)$
- Concorrência
 - se $V(e) = V(e')$ então $e \parallel e'$
 - podemos usar o valor de tempo lógico do evento para resgatar a relação de concorrência

Exemplo de Relógio de Vetor



- Relógio de Lamport: $L(e) < L(c)$ mas não temos $e \rightarrow c$
- Relógio de Vetor $V(e) = V(c)$ então $e \parallel c$

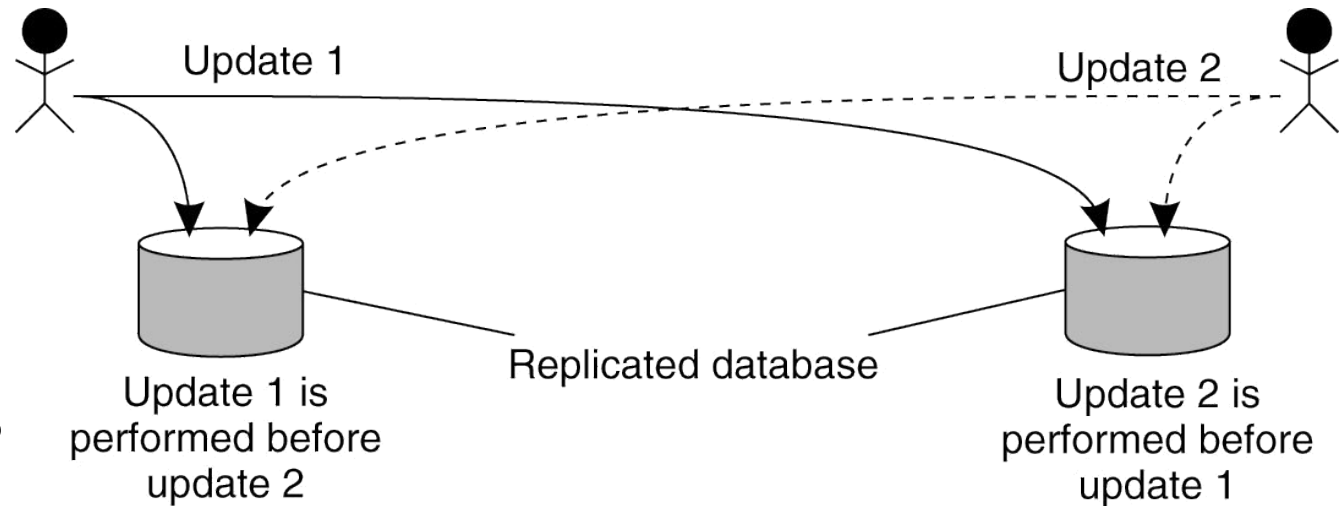
Outras Propriedades

- Seja $RT(x)$ o valor de tempo real da ocorrência do evento x (ex. UTC)
 - se $V(a) < V(b)$ então $RT(a) < RT(b)$
- Seja $L(x)$ o valor do relógio de Lamport de x
 - se $V(a) < V(b)$ então $L(a) < L(b)$
 - pois vetor recupera relação de ordem
- Mas oferece apenas ordenação parcial
 - não oferece ordenação total dos eventos (ainda temos eventos concorrentes)
 - mas faz o melhor possível, resgata a relação de ocorreu antes e concorrência

Sincronização de Relógios

- Relógios em sistemas diferentes serão sempre diferentes
 - até mesmo em relógios atômicos
- Relógios não sincronizados em sistemas distribuídos podem levar a comportamento errático
- Duas abordagens de solução
 - sincronização do relógio
 - consistência na ordem dos eventos (ordenação total dos eventos)
- Na prática, ambos são usados

BD Distribuído e Replicado



- Conta com saldo inicial de \$1000

- Dois usuários fazem transações

- Usuário 1: aumenta saldo em \$100

- Usuário 2: aumenta saldo em 1%

- Saldo final da conta?

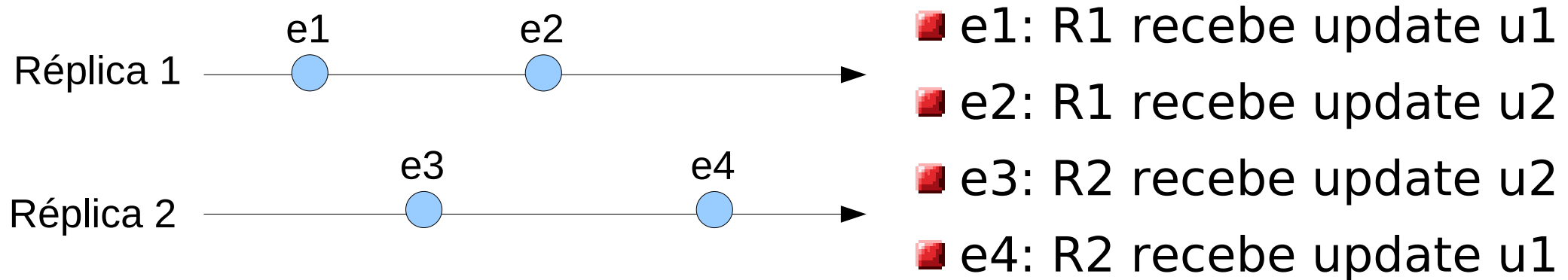
- se $u1 \rightarrow u2$: \$1111

- se $u2 \rightarrow u1$: \$1110

- Como obter consistência nos dois bancos?

- valor final não importa muito, consistência sim

Eventos e Tempos



■ $L(e1) < L(e2), L(e3) < L(e4)$

■ mas $L(e1) \parallel L(e3), L(e2) \parallel L(e4)$

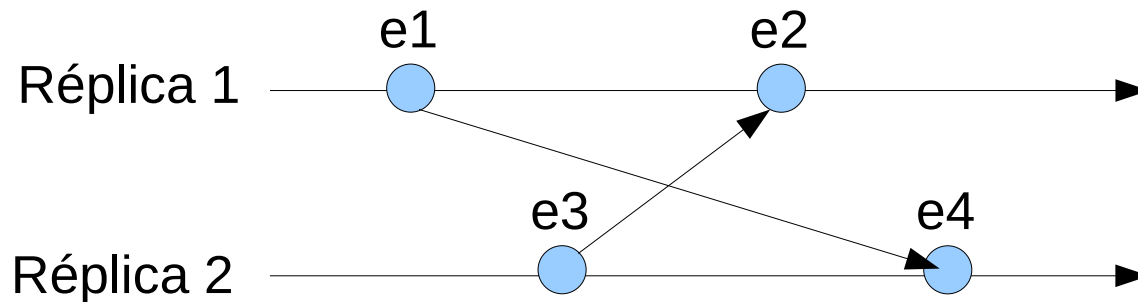
■ Não temos relação entre eventos nas diferentes réplicas

■ Como relacionar eventos (transações) das diferentes réplicas?

Réplicas enviam mensagens!

Trocando Mensagens

- Novo sistema distribuído
 - usuário envia transação para uma réplica
 - réplica envia transação para a outra



- e1: R1 recebe update u1, envia para R2
- e2: R1 recebe u2 de R2
- e3: R2 recebe update u2, envia para R1
- e4: R2 recebe u1 de R1

- Temos agora $L(e1) < L(e4)$, $L(e3) < L(e2)$
 - mas ainda $L(e2) \parallel L(e4)$
- Relógio de vetor também não resolve

Garantindo Ordem Total

- **Ideia:** processar transações distribuídas em uma ordem
 - todos processos executam transações na mesma ordem
 - transação será confirmada antes de ser executada
- **Algoritmo:**
 - manter em cada processo um relógio lógico de Lamport
 - cada transação tem valor de relógio
 - cada processo mantém uma fila com todas as transações (incluindo as suas)
 - ordenada pelo valor do relógio lógico da transação
 - cada transação em P_i é enviada a todos processos
 - via *multicast*, incluindo ao próprio processo
 - transação recebida por P_j é colocada na fila de transações e confirmada para todo processos
 - via *multicast*, incluindo o próprio processo

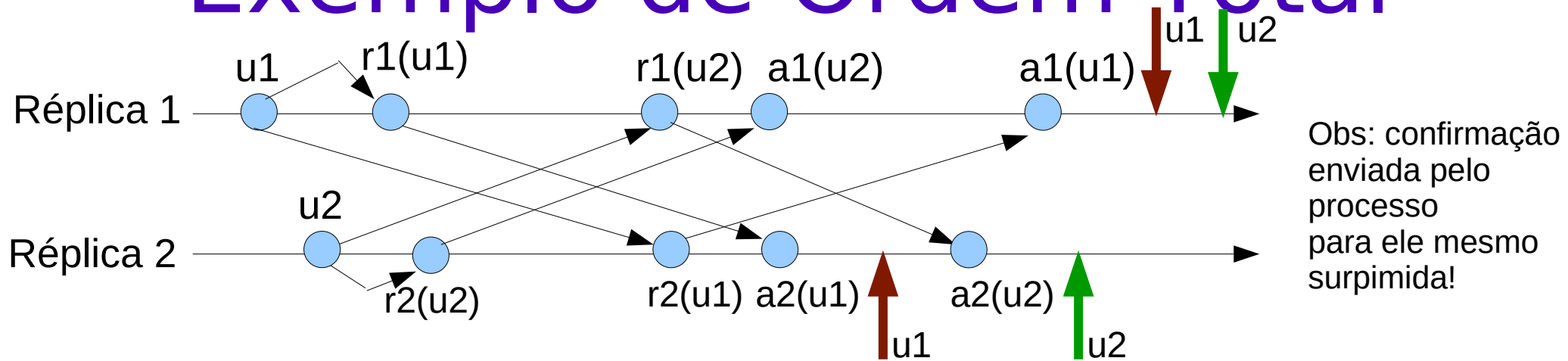
Garantindo Ordem Total

- **Multicast:** mensagem de *broadcast* enviada para todos os processos do sistema de forma eficiente
- Assumir rede FIFO e que mensagens não se perdem
 - m_1 transmitida antes de m_2 por P_i implica que todo P_j recebe m_1 antes de m_2
- **Processar transação**
 - transação da cabeça da fila é executada após o recebimento da confirmação de todos os processos
 - incluindo o próprio processo que iniciou a transação
- Filas vão ser idênticas em todos os processos
 - ordenadas por tempo lógico das transações

Garantindo Ordem Total

- **Problema:** tempo lógico de dois eventos em processos diferentes pode ser o mesmo
- **Solução:** adicionar número do processo como parte do relógio lógico: “L(e).identificador”
 - apenas para quebrar empate na fila
 - não afeta funcionamento do relógio lógico
 - assumir que identificador de processo é único

Exemplo de Ordem Total



- u_1, u_2 : transações a serem executadas
- $r_i(u_j)$: recebimento da transação j pelo processo i (vai para fila de transações, gera confirmação)
- $a_i(u_j)$: recebimento da confirmação da transação j pelo processo i
- No início, fila de transações em R1: [u_1 (1.1)], fila em R2: [u_2 (1.2)]
- Depois de $r_1(u_2)$, fila em R1: [u_1 (1.1), u_2 (1.2)]
- Depois de $r_2(u_1)$, fila em R2: [u_1 (1.1), u_2 (1.2)]
- Depois de $a_2(u_1)$ R2 pode executar u_1 : cabeça da fila, e confirmada por todos
- Depois de $a_1(u_2)$ R1 **não** pode executar u_1 , pois precisa aguardar confirmação de u_1 vindo de r_2 , $a_1(u_1)$
- Depois de $a_2(u_2)$ R2 pode executar u_2 : cabeça da fila e confirmada por todos
- Depois de $a_1(u_1)$ R1 pode executar u_1 , e em seguida u_2

Totally Ordered Multicast

- Algoritmo garante ordenação total das transações de forma distribuída
 - sem o uso de um servidor central
- Transações locais são transmitidas para todos processos do sistema
 - requer a confirmação do recebimento de cada transação para todos os processos do sistema
- Alta complexidade de mensagens
 - $1 + n$ mensagens de multicast por transação
- Atrasa execução dos eventos
 - execução apenas após chegada das confirmações
- Ainda é utilizado em alguns contextos
 - principalmente como primitiva para outros mecanismos