INSTITUTO DE INFORMÁTICA
Universidade Federal de Goiás

Sincronizando Eventos em Sistemas Distribuídos

Relógios Lógicos de Lamport

Aluno: Artur Rocha Lapot (15011640)

INSTITUTO DE INFORMÁTICA
UFG

UFG
UNIVERSIDADE FEDERAL DE GOIÁS

O Problema do Tempo em Sistemas Distribuídos

- **Tempo em Sistemas Centralizados:** Inequívoco. Um processo consulta o SO e obtém o tempo. Se o processo A consulta o tempo e depois o processo B, o tempo de B será maior ou igual ao de A.

- **Tempo em Sistemas Distribuídos:**
  - Alcançar concordância sobre o tempo é complexo.
  - Cada máquina possui seu próprio relógio, e esses relógios podem (e irão) divergir (fenômeno conhecido como *clock skew*).

O Problema do Tempo em Sistemas Distribuídos

- **Exemplo de Problema (Programa make do Unix):**
  - O make examina os timestamps de modificação dos arquivos fonte e objeto para decidir se um arquivo precisa ser recompilado.
  - Se o arquivo-fonte input.c tem timestamp 2151 e o objeto input.o tem 2150, input.c é recompilado.
  - Em um sistema distribuído, se o relógio da máquina onde output.c foi modificado estiver atrasado, ele pode receber um timestamp (ex: 2143) anterior ao de output.o (ex: 2144), mesmo que a modificação tenha sido posterior.
  - Resultado: O make não recompilaria output.c, levando a um executável com código inconsistente e possíveis falhas.

- **O que Realmente Importa:** Muitas vezes, não é o tempo absoluto, mas a **ordem** em que os eventos ocorrem.

A Relação "Acontece Antes" (Happens-Before)

Para sincronizar relógios lógicos, Lamport (1978) definiu a relação "acontece antes", denotada por $a \rightarrow b$.

- **Observação Direta da Relação:**
  1. **No Mesmo Processo:** Se a e b são eventos no mesmo processo, e a ocorre antes de b (na ordem do programa), então $a \rightarrow b$ é verdadeiro.
  2. **Troca de Mensagens:** Se a é o evento de envio de uma mensagem por um processo, e b é o evento de recebimento dessa mensagem por outro processo, então $a \rightarrow b$ também é verdadeiro. Uma mensagem não pode ser recebida antes de ser enviada.

- **Propriedade da Transitividade:** Se $a \rightarrow b$ e $b \rightarrow c$, então $a \rightarrow c$.

- **Eventos Concorrentes:**
  - Se dois eventos, x e y, ocorrem em processos diferentes que não trocam mensagens (nem mesmo indiretamente), então $x \rightarrow y$ não é verdadeiro, nem $y \rightarrow x$.
  - Esses eventos são ditos **concorrentes**. Nada pode ser dito (ou precisa ser dito) sobre qual evento ocorreu primeiro.

Relógios Lógicos de Lamport: O Conceito Fundamental

- **Objetivo:** Atribuir um valor de tempo (timestamp) C(a) a cada evento a, de forma que todos os processos no sistema distribuído concordem com esses valores.

- **Propriedade Fundamental (Condição do Relógio):**
  - Se $a \rightarrow b$, então $C(a) < C(b)$.

- **Implicações da Propriedade Fundamental:**
  - Se a e b são dois eventos dentro do mesmo processo e a ocorre antes de b, então $C(a) < C(b)$.
  - Se a é o envio de uma mensagem por um processo e b é o recebimento dessa mensagem por outro processo, então $C(a) < C(b)$.

- **Propriedade Adicional do Relógio:**
  - O tempo do relógio, C, deve sempre avançar (aumentar), nunca retroceder.
  - Correções ao tempo do relógio são feitas adicionando um valor positivo, nunca subtraindo.

O Algoritmo de Lamport para Atribuição de Tempos

O algoritmo de Lamport é tipicamente implementado na camada de middleware. Cada processo Pi mantém um contador local Ci (seu relógio lógico).

- **Regras de Atualização dos Contadores:**
  1. **Antes de executar um evento** (seja ele o envio de uma mensagem, a entrega de uma mensagem para a aplicação, ou algum outro evento interno relevante para a ordenação):
     - Pi incrementa seu contador local: Ci ⟵ Ci + 1.
  2. **Quando o processo Pi envia uma mensagem m para o processo Pj:**
     - Pi define o timestamp da mensagem ts(m) como o valor de Ci *após* ter executado o incremento do passo 1.
  3. **Ao receber uma mensagem m do processo Pi, o processo Pj:**
     - Ajusta seu próprio contador local: Cj ⟵ max{Cj, ts(m)}.
     - Em seguida, Pj executa o passo 1 (incrementa Cj ⟵ Cj + 1).
     - Finalmente, entrega a mensagem à aplicação.

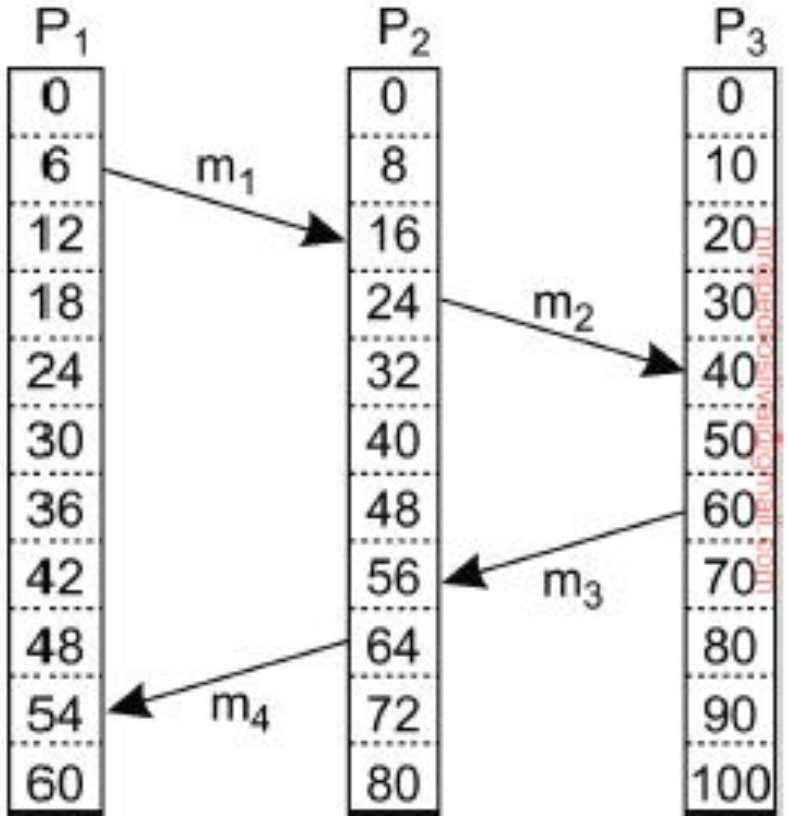
O Algoritmo de Lamport para Atribuição de Tempos

- **Desempate (Garantindo Unicidade Total):**
  - Se for desejável que não haja dois eventos ocorrendo exatamente no mesmo "tempo" lógico, pode-se anexar o identificador único do processo ao timestamp.
  - Exemplo: Um evento no tempo lógico 40 no processo Pi é timestamped como ⟨40, i⟩. Se i < j, então ⟨40, i⟩ é considerado anterior a ⟨40, j⟩.

Exemplo Visual do Algoritmo

- **Cenário Inicial:**
  - Três processos (P1, P2, P3) com relógios lógicos que avançam em taxas diferentes (ex: P1 incrementa de 6, P2 de 8, P3 de 10).

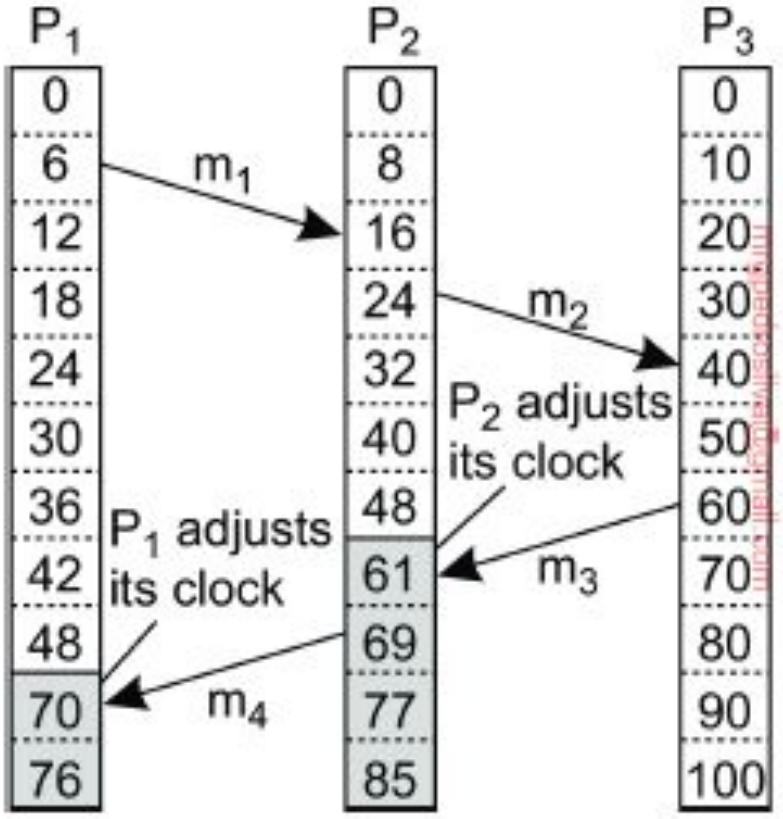
- **Problema de Inconsistência Temporal:**
  - Mensagem m3 sai de P3 no tempo lógico 60.
  - Chega em P2 quando o relógio de P2 marca 56. Isso viola C(envio) < C(recebimento).



Exemplo Visual do Algoritmo

- **Correção com o Algoritmo de Lamport:**
  - P2, ao receber m3 com ts(m3) = 60, percebe que seu relógio (56) está atrasado.
  - P2 ajusta seu relógio: C2 ← max{56, 60} = 60.
  - P2 então incrementa seu relógio (passo 1 do algoritmo): C2 ← 60 + 1 = 61.
  - A mensagem m3 é considerada como chegada em P2 no tempo lógico 61.
  - Similarmente, a mensagem m4 chega em P1 no tempo 70 (pois C1 é ajustado para max{54,69} + 1 = 70 após receber m4 com ts(m4) = 69 de P2, que tinha C2 = 69 ao enviar).

- **Resultado:** A ordenação "acontece antes" é preservada para todas as trocas de mensagens.



Exemplo de Aplicação - Multicast Totalmente Ordenado

Uma aplicação importante dos relógios de Lamport é a implementação de **multicast totalmente ordenado** de forma distribuída.

- **Problema a ser Resolvido:**
  - Em sistemas com dados replicados (ex: banco de dados replicado em Nova York e São Francisco), as atualizações devem ser aplicadas na mesma ordem em todas as réplicas para manter a consistência.
  - Se a atualização A1 (+ $100) e A2 (+1% de juros) chegam em ordens diferentes nas réplicas, os saldos finais podem divergir (ex: $1111 vs $1110 na Figura 5.9).

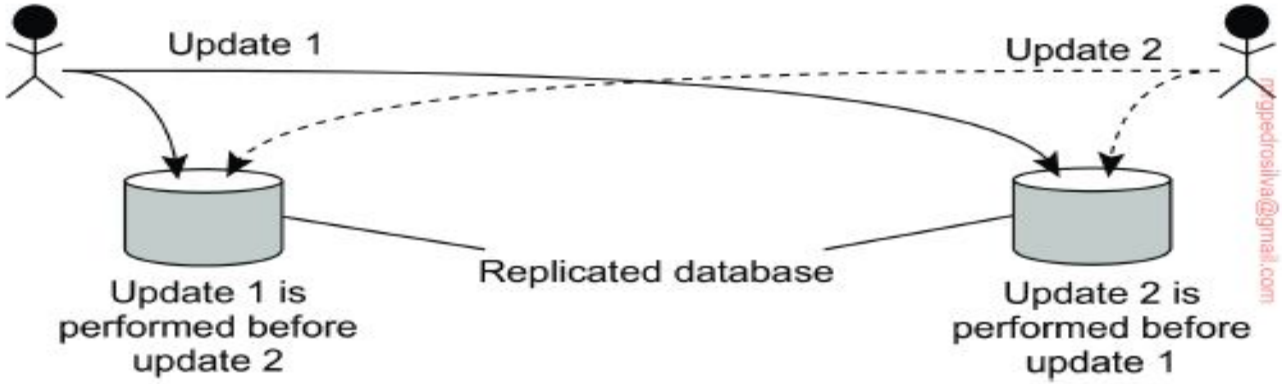


Figure 5.9: Updating a replicated database and leaving it in an inconsistent state.

Exemplo de Aplicação - Multicast Totalmente Ordenado

- **O que é Multicast Totalmente Ordenado?**
  - Uma operação de multicast onde todas as mensagens são entregues **na mesma ordem** para cada processo receptor no grupo.

- **Objetivo:** Garantir que, mesmo com atrasos de comunicação, todas as cópias de um banco de dados (ou qualquer estado replicado) processem as atualizações na mesma sequência.

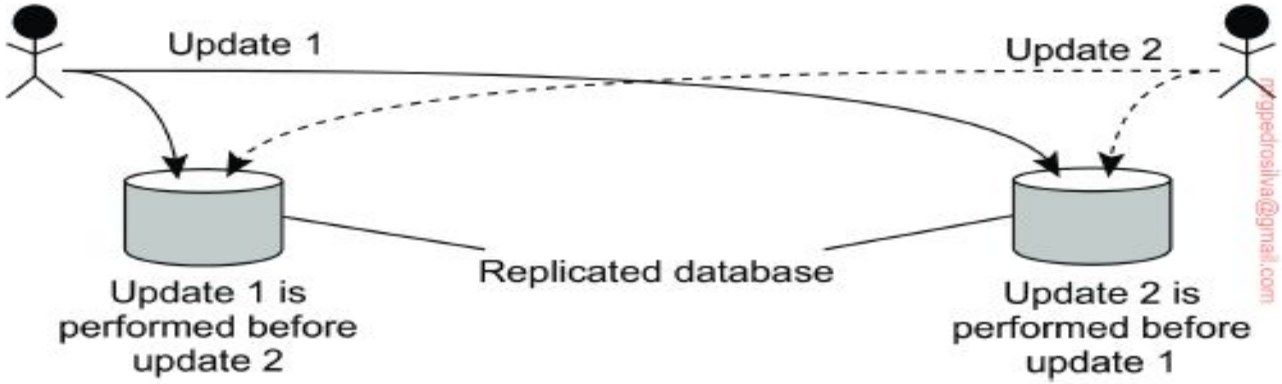


Figure 5.9: Updating a replicated database and leaving it in an inconsistent state.

Funcionamento do Multicast Totalmente Ordenado com Relógios de Lamport

- **Premissas:**
  - Mensagens são enviadas em multicast para todos os membros do grupo (incluindo o remetente, conceitualmente).
  - Mensagens do mesmo remetente são recebidas na ordem em que foram enviadas.
  - Nenhuma mensagem é perdida (confiabilidade na camada inferior).

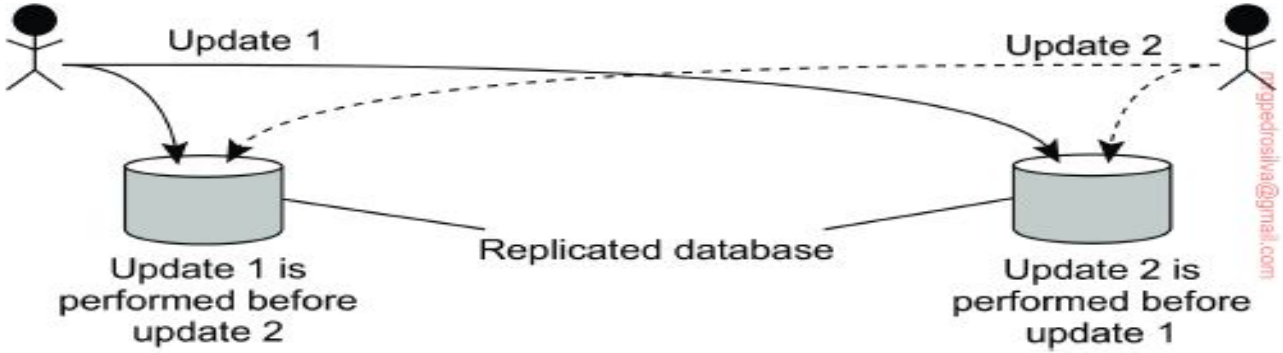


Figure 5.9: Updating a replicated database and leaving it in an inconsistent state.

Funcionamento do Multicast Totalmente Ordenado com Relógios de Lamport

- **Mecanismo:**
  1. **Timestamping:** Cada mensagem multicast é timestamped com o tempo lógico (Lamport) atual do remetente.

  2. **Fila Local:** Ao receber uma mensagem, cada processo a insere em uma fila local, ordenada de acordo com seu timestamp. Mensagens com o mesmo timestamp são desempatadas usando o ID do processo remetente.

  3. **Acknowledgments (ACKs):** O receptor envia um multicast de ACK para os outros processos. O timestamp do ACK será maior que o da mensagem original (devido às regras de Lamport).

Funcionamento do Multicast Totalmente Ordenado com Relógios de Lamport

4. **Condição de Entrega à Aplicação:** Um processo só pode entregar uma mensagem m (que está na cabeça de sua fila) para a aplicação se:
   - m está na cabeça da fila local (é a próxima na ordem de timestamps).
   - m foi reconhecida (ACKed) por todos os outros processos no grupo. (Isso significa que todos os outros processos também viram m e concordam com sua posição na ordem global).

5. **Remoção da Fila:** Após a entrega, a mensagem é removida da fila e os ACKs associados podem ser descartados.

- **Resultado:** Como todos os processos mantêm a mesma cópia da fila (eventualmente) e usam a mesma regra de entrega, todas as mensagens são entregues na mesma ordem global para todas as aplicações. Isso é conhecido como **replicação de máquina de estados** (state machine replication).

Limitações dos Relógios Lógicos de Lamport

- **Não Capturam Causalidade Totalmente:**
  - A principal limitação é que, embora os relógios de Lamport garantam que se $a \rightarrow b$ então $C(a) < C(b)$, o inverso não é necessariamente verdadeiro.
  - Ou seja, se $C(a) < C(b)$, isso **não implica** que o evento a necessariamente "aconteceu antes" do evento b no sentido de uma dependência causal.
  - Os eventos a e b podem ser concorrentes, mas devido à forma como os timestamps são atribuídos (incluindo IDs de processo para desempate), um pode acabar com um timestamp menor que o outro.

Limitações dos Relógios Lógicos de Lamport

- **Exemplo de Ambiguidade:**
  - Na Figura 5.11, podemos ter Trcv(m1) < Tsnd(m2). No entanto, o envio de m2 pode não ter tido nenhuma relação causal com o recebimento de m1.

  - Os relógios de Lamport não conseguem distinguir esta situação de uma onde haveria uma dependência causal.

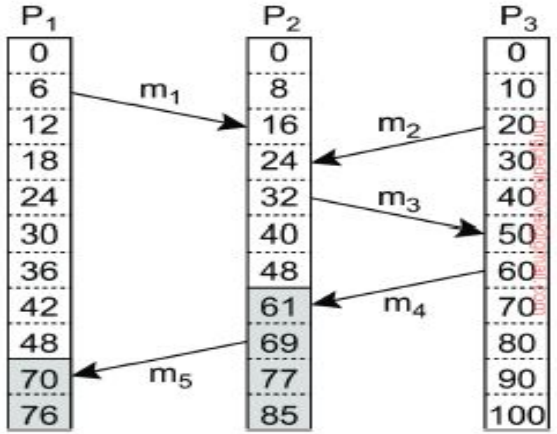


Figure 5.11: Concurrent message transmission using logical clocks.

Limitações dos Relógios Lógicos de Lamport

- **Necessidade de Algo Mais para Causalidade:**
  - Para capturar explicitamente as relações causais (ou seja, para que C(a) < C(b) implique a ⟶ b), são necessários mecanismos mais fortes, como os **Relógios Vetoriais**.

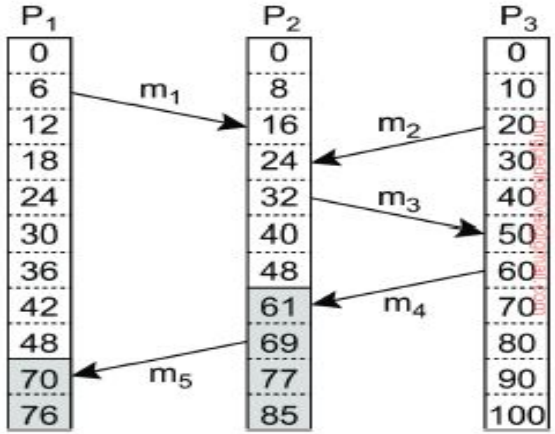


Figure 5.11: Concurrent message transmission using logical clocks.

Conclusão

- **Ordenação sem Tempo Absoluto:** Relógios Lógicos de Lamport fornecem um mecanismo fundamental e elegante para impor uma ordem total a eventos em sistemas distribuídos, sem a necessidade de relógios físicos perfeitamente sincronizados.

- **Base para Consistência:** São cruciais para a construção de algoritmos distribuídos que requerem um acordo sobre a ordem dos eventos, como é o caso do multicast totalmente ordenado, essencial para a replicação de máquinas de estado.

- **Simplicidade e Ampla Aplicabilidade:** A técnica é baseada na relação intuitiva "acontece antes" e, apesar de suas limitações em capturar toda a causalidade, sua simplicidade e eficácia em prover uma ordenação total a tornam uma ferramenta valiosa em sistemas distribuídos.

- **Próximos Passos:** Para uma noção mais forte de causalidade, onde a ordem dos timestamps reflete diretamente as dependências causais, outras técnicas como os Relógios Vetoriais são exploradas.