Sistemas Distribuídos

Aula 8 – Sincronização em Sistemas Distribuídos

Prof. Ana Carolina Sokolonski

Bacharelado em Sistemas de Informação
Instituto Federal da Bahia – Campus Feira de Santana

2026

Agenda

O Problema do Tempo

Sincronização de Relógios Físicos

Relógios Lógicos de Lamport

Relógios Vetoriais

Exclusão Mútua Distribuída

Eleição de Líder

Transações Distribuídas

Deadlock Distribuído

Comparação e Resumo

Referências

O Problema do Tempo

Por quê sincronizar em SDs?

Por quê Sincronização é Difícil?

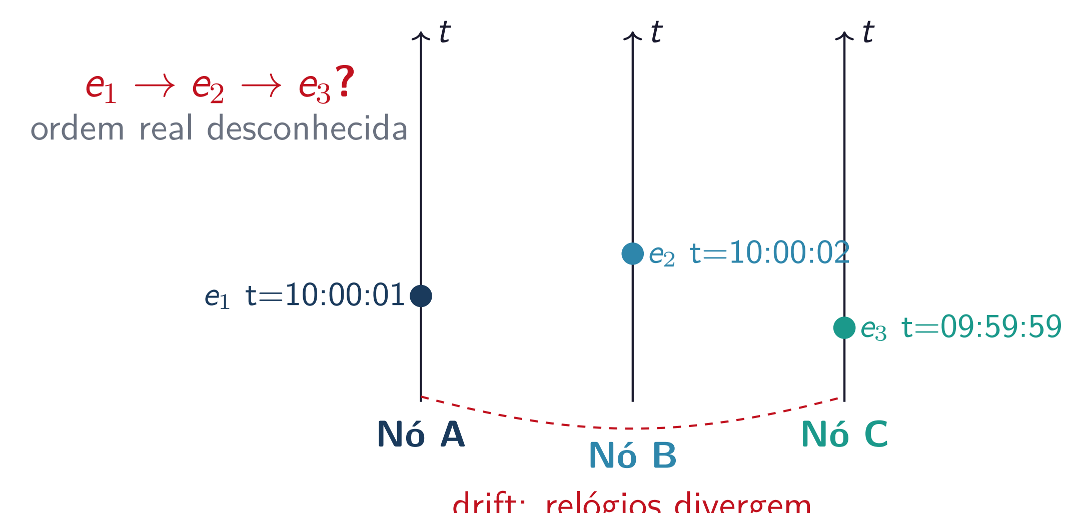
▪ Não existe relógio global

Em um SD, cada nó possui seu **próprio relógio físico** (*quartz oscillator*). Esses relógios derivam ao longo do tempo (*clock drift*) e **nunca ficam perfeitamente sincronizados** via rede.

▪ Consequências práticas

- Qual evento aconteceu primeiro?
- Como ordenar transações em bancos de dados distribuídos?
- Como garantir consistência entre réplicas?
- Como detectar deadlocks distribuídos?

Por que Sincronização é Difícil?



Sincronização de Relógios Físicos

NTP e algoritmos de ajuste

NTP — Network Time Protocol

Como funciona o NTP?

Hierarquia de servidores (**estratos**) que sincronizam relógios com fontes de tempo atômico (GPS, cesium). Cada nó consulta servidores NTP periodicamente e **ajusta gradualmente** seu relógio local.

Fórmula de estimativa do offset

Dado RTT $= T_4 - T_1$:

$$\theta = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}$$

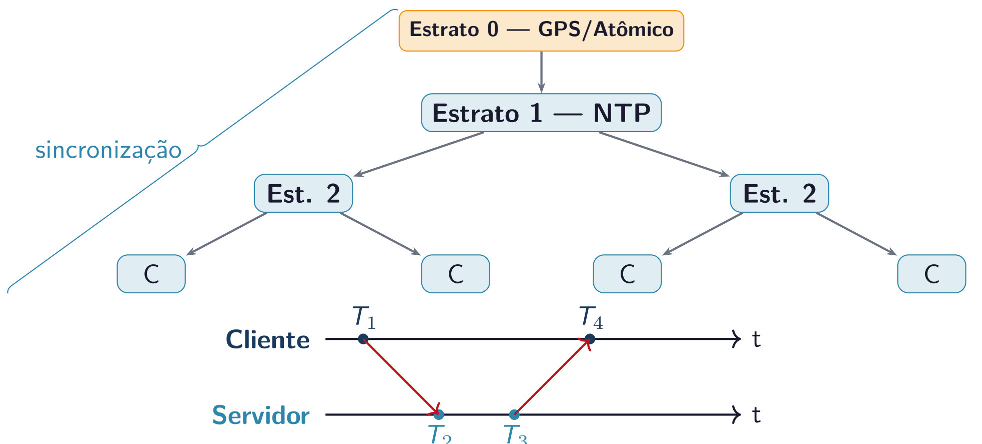
$$\delta = (T_4 - T_1) - (T_3 - T_2)$$

θ: offset estimado $\quad \delta$: RTT

Limitação

Precisão típica: **1–10 ms** em LAN, até **100 ms** na Internet. Não é suficiente para ordenação precisa de eventos.

NTP — Network Time Protocol



Relógios Lógicos de Lamport

Ordem sem relógio físico

A relação Happens-Before ($\rightarrow$)

Lamport (1978) definiu a relação **happens-before** sem usar relógios físicos:

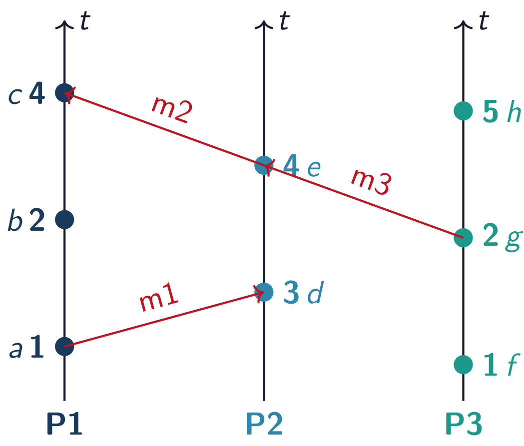
1. Se a e b estão no mesmo processo e a ocorre antes de b: $a \rightarrow b$
2. Se a é o envio de mensagem e b é o recebimento: $a \rightarrow b$
3. Transitividade: se $a \rightarrow b$ e $b \rightarrow c$, então $a \rightarrow c$

Eventos sem relação happens-before são **concorrentes**: $a \parallel b$

Regras do Relógio Lógico

- Antes de cada evento: $C \leftarrow C + 1$
- Ao enviar msg: inclui timestamp C na mensagem
- Ao receber msg com timestamp t: $C \leftarrow \max(C, t) + 1$

Relógios de Lamport — *Happens-Before*



Relógios Vetoriais

Capturando causalidade completa

Limitação do Relógio de Lamport

O relógio de Lamport garante: $a \to b \Rightarrow C(a) < C(b)$
Mas **NÃO** a inversa: $C(a) < C(b) \not\Rightarrow a \to b$
Não consegue distinguir eventos concorrentes de eventos causalmente relacionados.

Relógio Vetorial — Regras

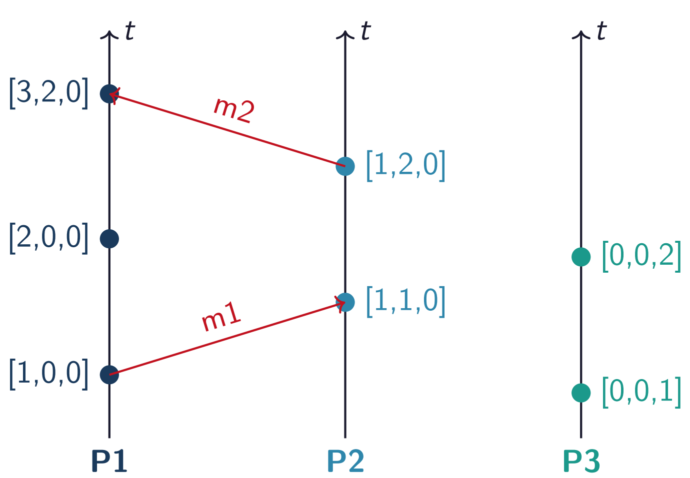
Cada processo P_i mantém vetor $V_i[1..n]$:

- **Evento local**: $V_i[i] \mathrel{+}= 1$
- **Envio**: inclui V_i na mensagem
- **Recebimento** de msg com vetor V_m: $V_i[j] = \max(V_i[j], V_m[j])$ para todo j, depois $V_i[i] \mathrel{+}= 1$

Propriedade chave

$V(a) < V(b) \iff a \to b$ (bicondicional!)
Permite detectar **concorrência real**: $a \parallel b$
sse $V(a) \not\leq V(b)$ e $V(b) \not\leq V(a)$

Relógios Vetoriais — Vector Clocks



Exclusão Mútua Distribuída

Acesso seguro a recursos compartilhados

Exclusão Mútua em SDs — O Problema

Por que é difícil em SDs?

Em sistemas centralizados, mutexes e semáforos resolvem o problema. Em SDs **não há memória compartilhada** e **não há árbitro central confiável**.

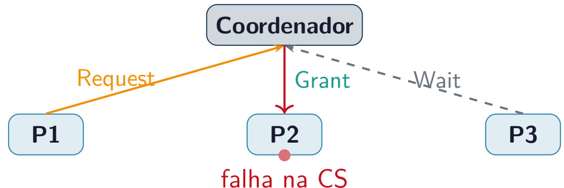
Qualquer solução precisa ser baseada em **troca de mensagens** e tolerante a falhas.

Requisitos de uma solução

- **Segurança**: no máximo um processo na seção crítica
- **Vivacidade**: todo processo que pede acesso eventualmente o obtém
- **Ordem**: respeitar a ordem de requisições (FIFO)

▪ Algoritmo Centralizado

Um **coordenador** concede permissão. Simples, mas ponto único de falha. Cada CS requer 3 mensagens: Request, Grant, Release.



▪ Ricart-Agrawala (1981)

Totalmente distribuído. Para entrar na CS:

1. Envia **REQUEST** com timestamp para todos
2. Aguarda **OK** de **todos** os outros processos
3. Entra na CS; ao sair, envia OK para quem estava esperando

Um processo responde OK imediatamente se não quer a CS ou se seu timestamp é maior. Caso contrário, enfileira o pedido.

Complexidade: $2(n-1)$ mensagens por CS.
Problema: tolerância a falhas baixa.

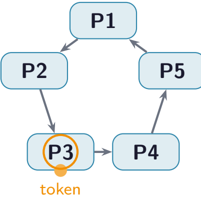
Algoritmo do Anel (Token Ring)

Processos organizados em **anel lógico**. Um **token** circula continuamente.

Para entrar na CS: aguarda o token chegar. Ao receber: entra na CS (se quiser), depois passa o token para o próximo.

Vantagem: sem starvation, sem deadlock.
Desvantagem: se um processo falha, o token é perdido.

Complexidade: 1 a n mensagens por CS.



Eleição de Líder

Escolhendo um coordenador

▪ Algoritmo Bully (Garcia-Molina, 1982)

O processo com maior ID vence a eleição.

Quando um processo P percebe que o coordenador falhou:

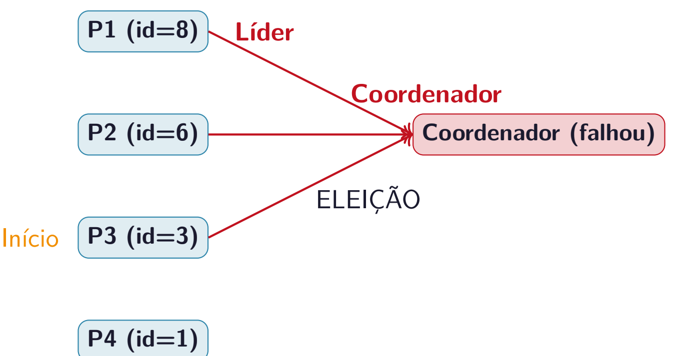
1. P envia ELECTION para todos com ID maior
2. Se ninguém responde: P se torna coordenador
3. Se alguém responde: aquele assume e repete o processo
4. O vencedor envia COORDINATOR para todos

Complexidade: $O(n^2)$ mensagens.
Problema: o maior processo pode estar lento ou sobrecarregado.

Algoritmo do Anel (Eleição)

Processos em anel lógico. O processo que inicia a eleição envia seu ID. Cada processo repassa o maior ID que viu. Quando o iniciador recebe seu próprio ID de volta: ele é o líder e anuncia.

Complexidade: $O(n)$ mensagens.



Transações Distribuídas

Atomicidade em múltiplos nós

Two-Phase Commit (2PC)

▪ Propriedades ACID em SDs

Atomicidade	Tudo ou nada (mesmo em n nós)
Consistência	Estado válido antes e depois
Isolamento	Transações não interferem
Durabilidade	Efeitos persistem após falha

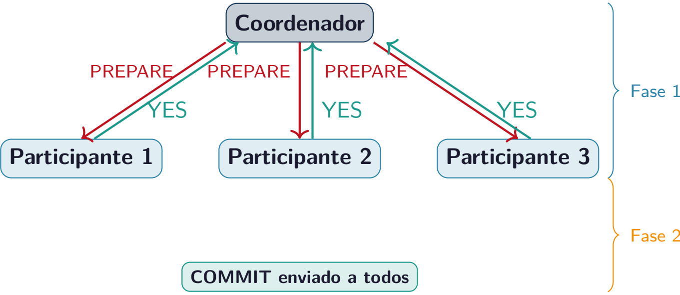
▪ Two-Phase Commit (2PC)

Fase 1 — Preparação: Coordenador envia PREPARE; cada participante vota YES (pode commitar) ou NO (deve abortar).

Fase 2 — Decisão: Se todos votaram YES: coordenador envia COMMIT. Se qualquer um votou NO: envia ABORT.

Problema: se o coordenador falha entre as fases, participantes ficam **bloqueados** (*blocking protocol*).

Two-Phase Commit (2PC)



Deadlock Distribuído

Detecção e prevenção

Deadlock em Sistemas Distribuídos

▪ Deadlock Distribuído

Ocorre quando um conjunto de processos em diferentes nós ficam **esperando indefinidamente** por recursos que estão sendo usados por outros processos do conjunto.

Mais difícil de detectar do que em sistemas centralizados: **não existe uma visão global do estado do sistema.**

▪ Grafo de Espera (Wait-for Graph)

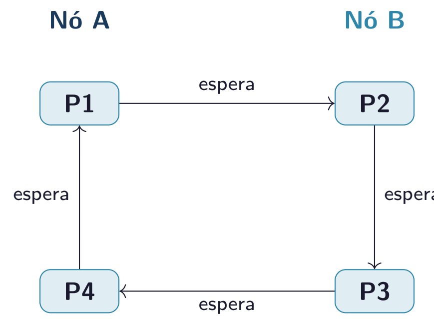
Nó $P_i \rightarrow P_j$ significa P_i está esperando recurso de P_j. **Deadlock** $\equiv$ **ciclo no grafo**.

Em SDs: grafo distribuído entre nós precisa de um algoritmo para detectar ciclos globais.

Deadlock em Sistemas Distribuídos

▪ Estratégias

- **Detecção**: Chandy-Misra-Haas
- **Prevenção**: timestamps (wait-die, wound-wait)
- **Evasão**: ordenação de recursos



Deadlock (ciclo fechado)

Problema	Solução	Trade-off
Sincronização de tempo	NTP, PTP, GPS (Spanner)	Precisão vs. custo
Ordem de eventos	Relógio de Lamport	Ordem parcial, sem concorrência
Causalidade	Relógio Vetorial	$O(n)$ overhead por mensagem
Exclusão mútua	Ricart-Agrawala, Token Ring	Msgs vs. tolerância a falhas
Eleição de líder	Bully, Anel	$O(n^2)$ vs. $O(n)$ mensagens
Transação distribuída	2PC, 3PC	Atomicidade vs. bloqueio
Deadlock	Detecção (CHM), Prevenção	Custo de detecção vs. starvation

Referências I

- TANENBAUM, A. S.; VAN STEEN, M. *Distributed Systems: Principles and Paradigms*. 3. ed. Pearson, 2017. Cap. 6.
- LAMPORT, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, v. 21, n. 7, pp. 558–565, 1978.
- RICART, G.; AGRAWALA, A. K. An Optimal Algorithm for Mutual Exclusion in Computer Networks. *Communications of the ACM*, v. 24, n. 1, 1981.
- GARCIA-MOLINA, H. Elections in a Distributed Computing System. *IEEE Transactions on Computers*, v. 31, n. 1, 1982.
- COULOURIS, G. et al. *Distributed Systems: Concepts and Design*. 5. ed. Addison-Wesley, 2013. Cap. 14–15.