GBC074 – Sistemas Distribuídos

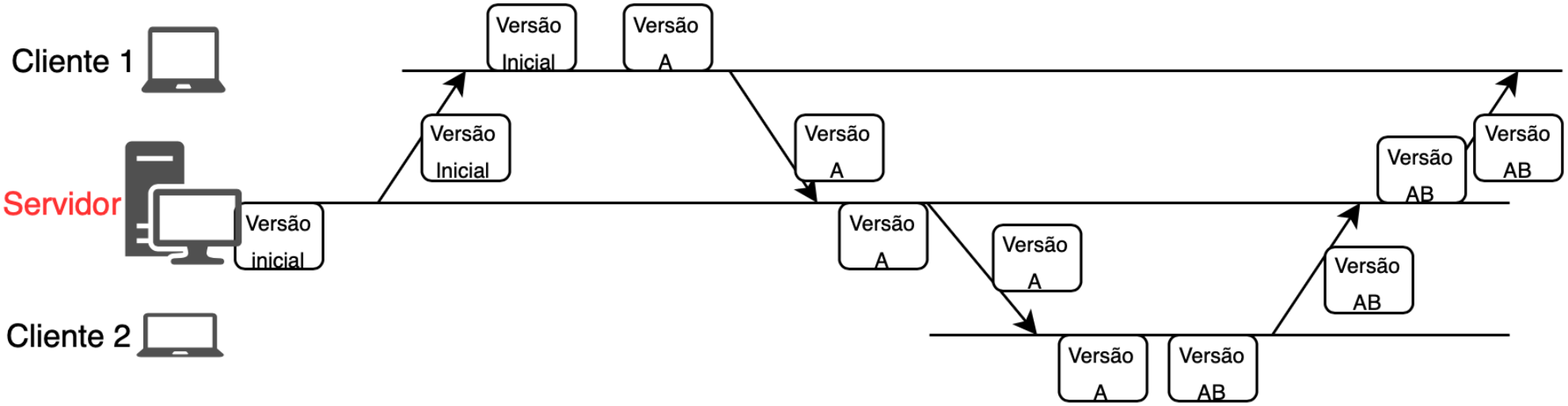
Tempo

Introdução

- Importância de tempo e relógio para sistemas distribuídos
- Exemplo:
  - **armazenamento de arquivos na nuvem (cloud-drive)**, sincronizado com o sistema de arquivos local
  - Similar a Dropbox, Box, Google Drive e OneDrive

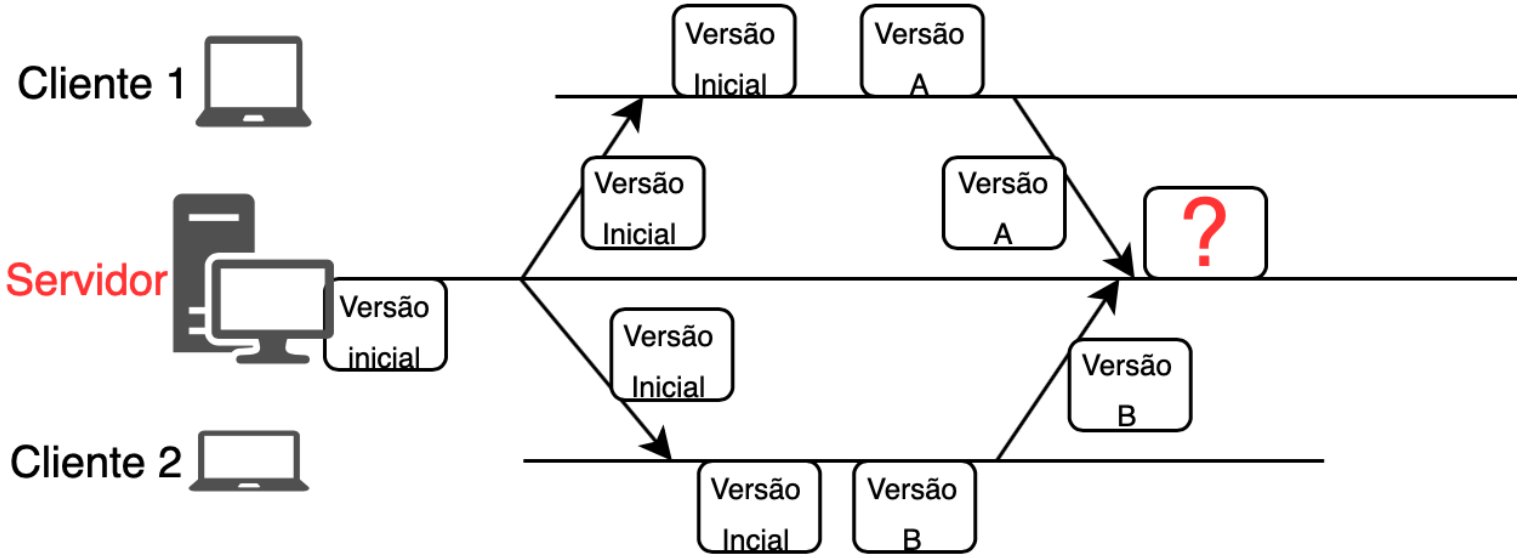
Introdução

- Cloud-drive:
  - Considera a sequência de eventos a seguir:



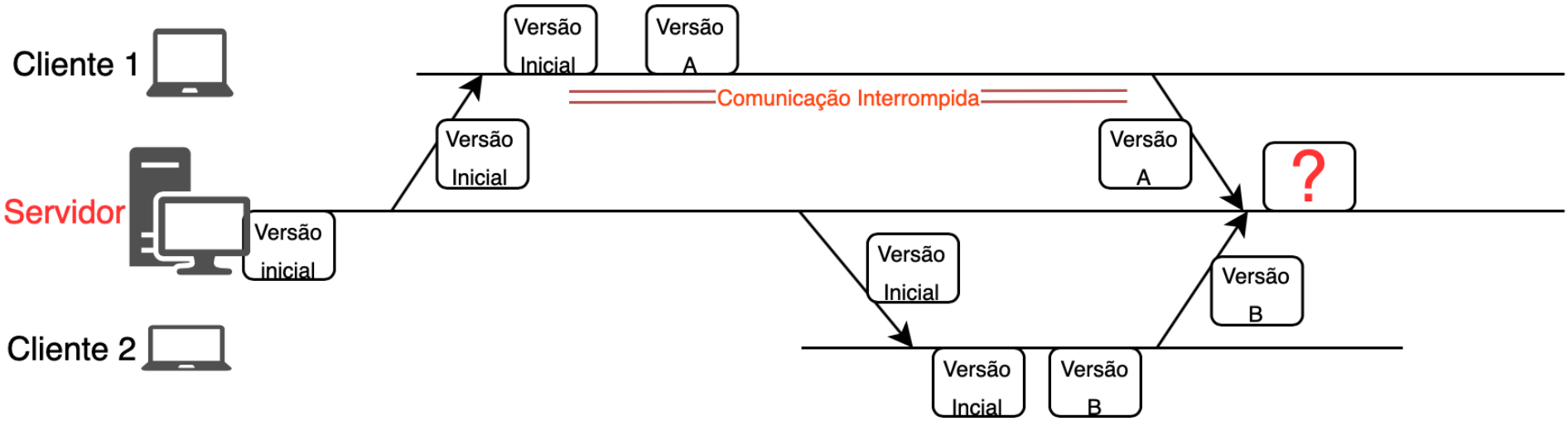
Introdução

- Cloud-drive:
  - Mesmo arquivo modificado em duas máquinas diferentes, enquanto desconectadas:
    - O que acontece quando elas se reconectam à Internet?
    - Qual versão deve ser armazenada e qual deve ser descartada?



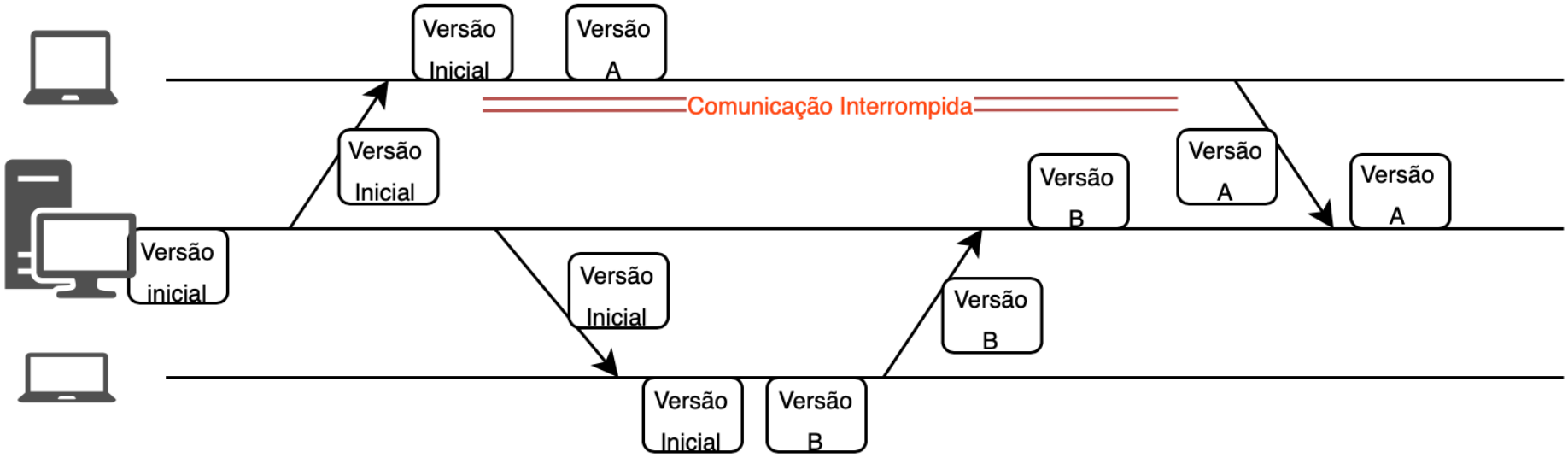
Introdução

- Cloud-drive:
  - Mesmo arquivo modificado em duas máquinas diferentes, enquanto a primeira esteve desconectada:
    - O que acontece quando elas se reconectam à Internet?
    - Qual versão deve ser armazenada e qual deve ser descartada?



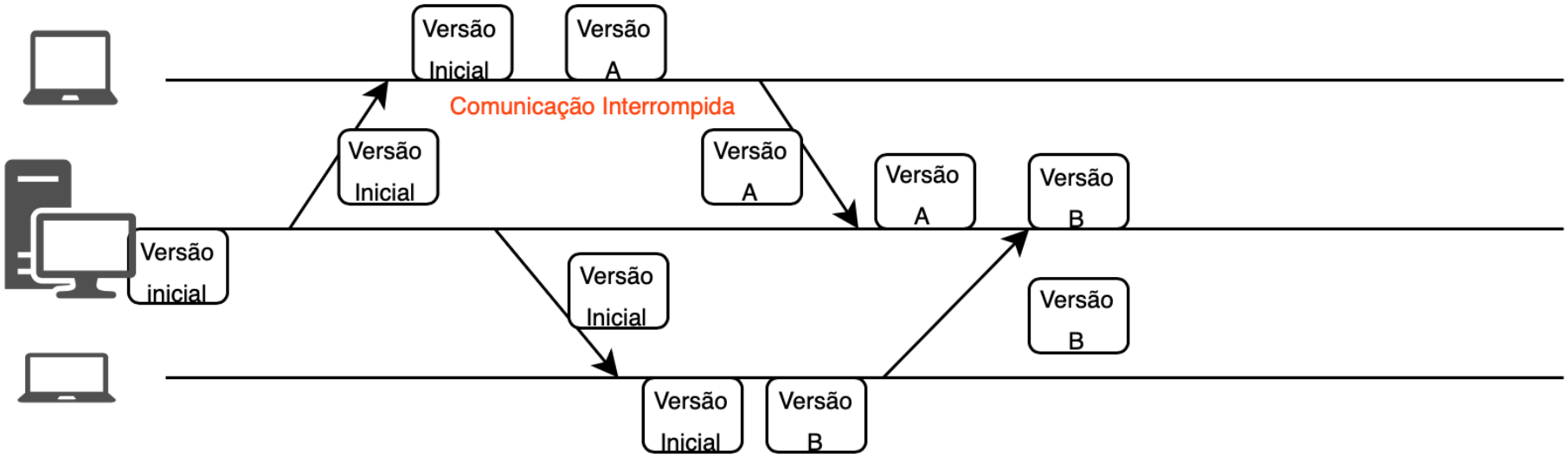
Introdução

- Cloud-drive:
  - Mesmo arquivo modificado em duas máquinas diferentes
  - Solução 1: tratar cada versão recebida como uma nova edição



Introdução

- Cloud-drive:
  - Mesmo arquivo modificado em duas máquinas diferentes
  - **Solução 1**: tratar cada versão recebida como uma nova edição



Introdução

- Cloud-drive:
  - Mesmo arquivo modificado em duas máquinas diferentes
  - **Solução 1**: tratar cada versão recebida como uma nova edição

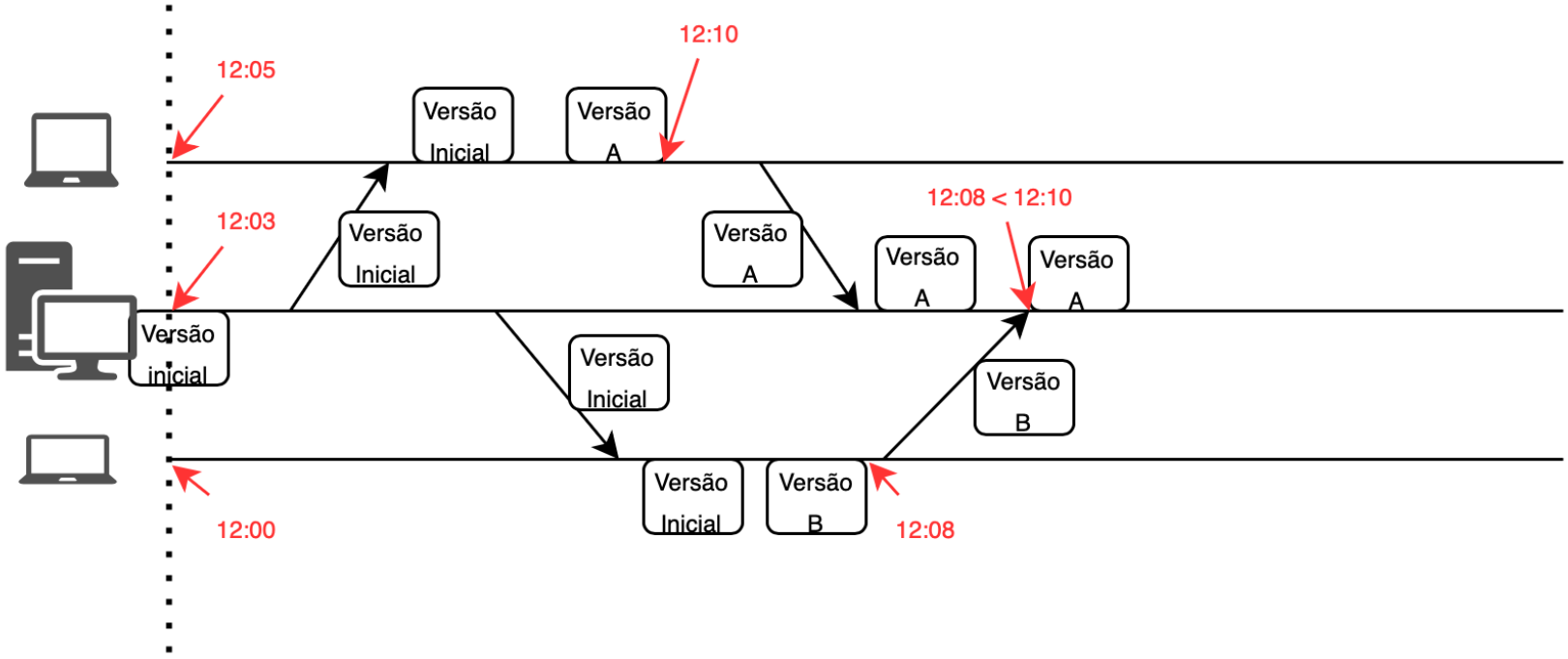
Problema: depende mais da rede do que das edições

Introdução

- Cloud-drive:
  - Mesmo arquivo modificado em duas máquinas diferentes
  - **Solução 2**: utilizar horário de **criação** e **modificação** do arquivo
    - Permanece no servidor o arquivo com data de modificação mais recente

Introdução

- Cloud-drive:
  - Mesmo arquivo modificado em duas máquinas diferentes
  - **Solução 2**: utilizar horário de **criação** e **modificação** do arquivo
    - Permanece no servidor o arquivo com data de modificação mais



Introdução

- Cloud-drive:
  - Mesmo arquivo modificado em duas máquinas diferentes
  - **Solução 2**: utilizar horário de **criação** e **modificação** do arquivo
    - Permanece no servidor o arquivo com data de modificação mais recente

Problema: horários são relativos!

Introdução

- Generalizando o problema:

Como ordenar operações de clientes?

- Exemplo:
  - CassandraDB usa ***last write wins*** ou ***latest version wins***, onde ***last*** é definido em termos do relógio do cliente
  - **Pergunta**
    - Como determinar qual foi enviada primeiro, em um sistema assíncrono?

Como sincronizar relógios em um sistema distribuído?

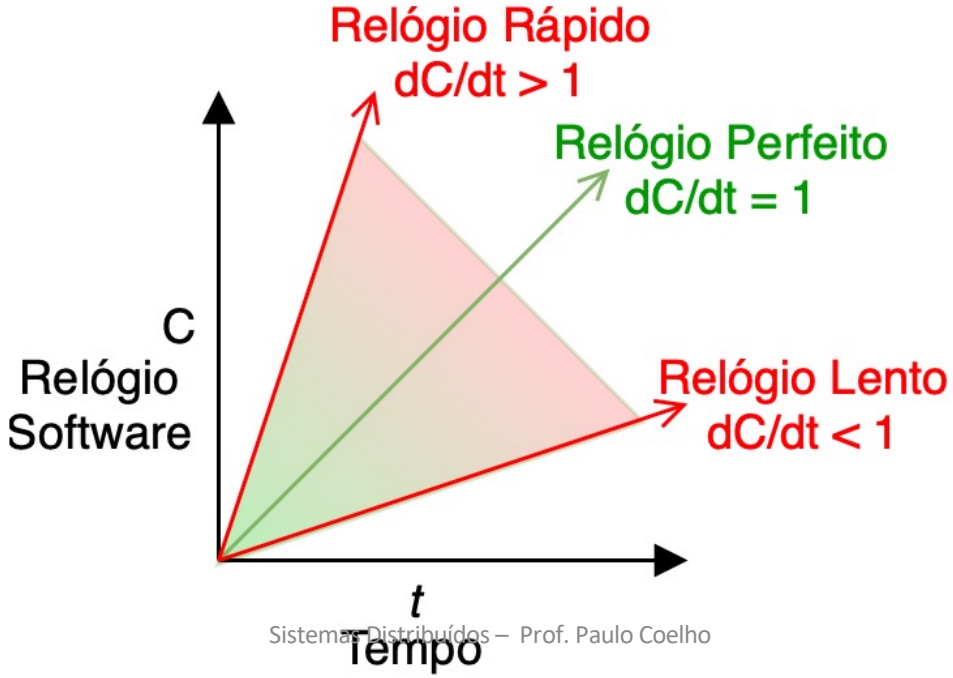
- Necessidade de **fonte de tempo confiável e distribuída:**
  - Protocolos de **sincronização de relógios físicos**.

Tempo físico

- Relógios em uma única máquina: ***Quartzo***
  - Diapazão de quartzo cortado a laser
  - Efeito Piezoelétrico vibra a $32768=2^{15}$Hz
  - Erro máximo: ½s por dia (faixa de 5 a 35 graus Celsius)
    - Erro varia com idade, corrente e imperfeições
- Incrementos são capturados em um contador
  - Gera **interrupções em intervalos programados**
    - Exemplo: Linux >2.6 usa 250Hz por padrão; máximo 1000Hz)
  - Interrupções causam ajustes em um **relógio em software**, um contador indireto C.

Tempo físico

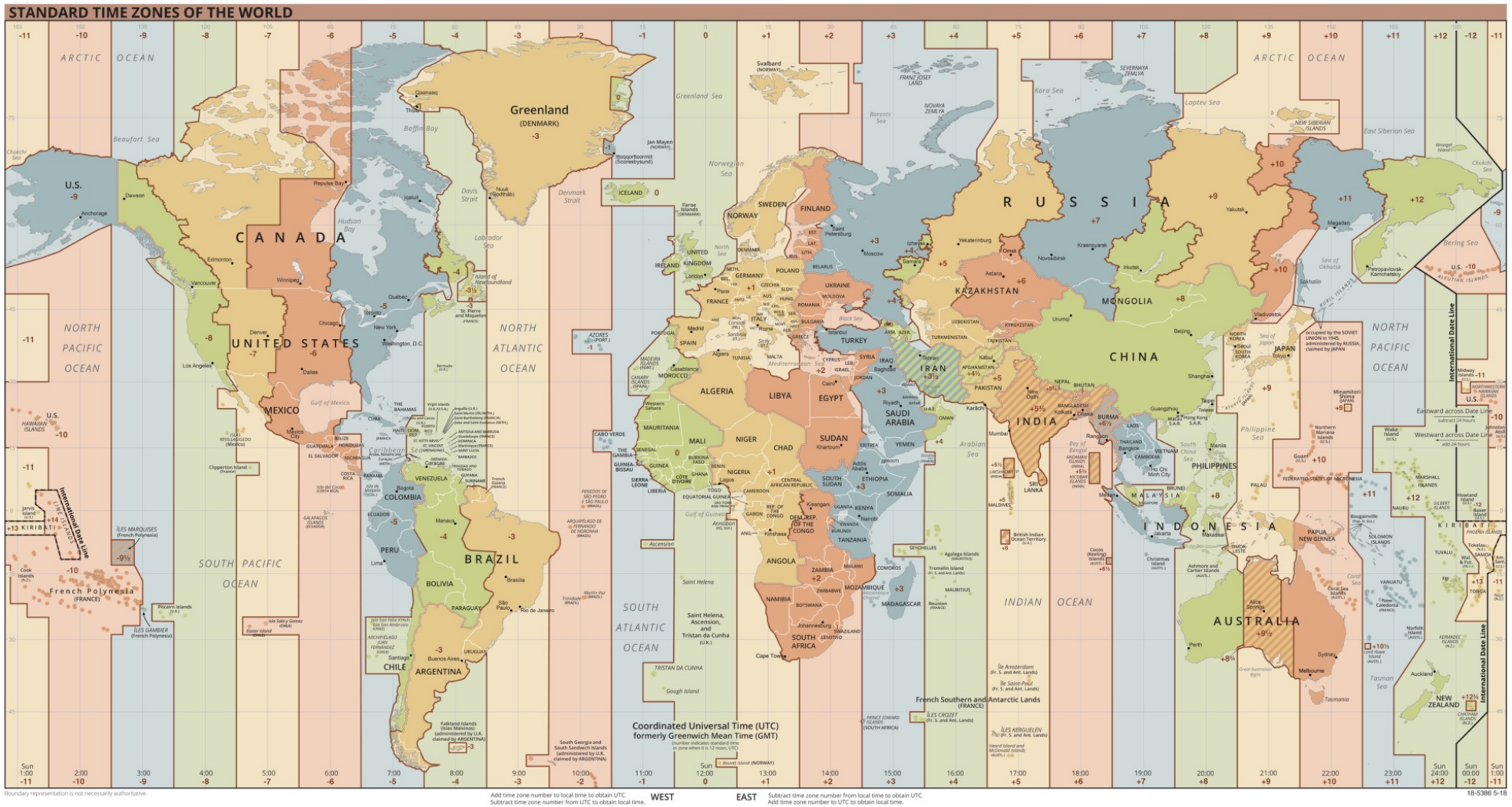
- Relógio em software (**C**):
  - Gera erro para mais ou para menos
  - Erro é limitado probabilisticamente
  - Taxa de erro (ρ) é denominada ***drift***
- Assumindo um relógio perfeito **t**, temos:
  - $1-\rho \leq dC/dt \leq 1+\rho$



Tempo físico

- Relógios atômicos: Erro < 1s / milhões de anos
- Usados para definir o O UTC (**Tempo Universal Coordenado)**
  - Baseado no TAI, Tempo Atômico Internacional
    - Leva em consideração o fato do dia não ter exatamente 24 horas
  - Sol está a pino às 12:00 na latitude 0, ou a no máximo 1s deste instante;
  - Ao redor da latitude 0 grau estabelece-se uma faixa em que todos os pontos tem o mesmo horário;
  - Outras 23 faixas como esta com deslocamentos consecutivos de +-1 hora;
  - Faixas sofrem ajustes por fatores políticos:
    - China, por exemplo, está toda dentro de um mesmo horário, "correto" para Beijing.

Tempo físico - UTC



Sincronização de relógios

- Com UTC resolvemos os problemas apresentados?

Sincronização de relógios

- Com UTC resolvemos os problemas apresentados?
- Não:
  - **Relógios podem se distanciar uns dos outros** na marcação do tempo
- É possível corrigir este distanciamento?

Sincronização de relógios

- Com UTC resolvemos os problemas apresentados?
- Não:
  - **Relógios podem se distanciar uns dos outros** na marcação do tempo
- É possível corrigir este distanciamento?
  - Também não!
- Sincronização frequente pode diminuir distanciamento
  - Qual a frequência de sincronização?

Sincronização de relógios

- Exemplo:

- ρ=0,1 (10%)
- δ = 1s
- Após 10s: falta de sincronia de até 1s w.r.t. UTC.
- Logo, a sincronização deve ser feita a cada 10s:
  - δ/ρ = 1s/0,1 = 10s

Sincronização de relógios

- Exemplo:

- ρ=0,1 (10%)
- δ = 1s

- E se quisermos sincronizar dois relógios com UTC?
  - Cada um com erro ρ
  - Não devem distanciar mais que δ unidades

Sincronização de relógios

- Exemplo:

- ρ=0,1 (10%)
- δ = 1s

- E se quisermos sincronizar dois relógios com UTC?
  - Cada um com erro ρ
  - Não devem distanciar mais que δ unidades

  - Erro máximo será de 2δ após 10 segundos
  - Basta **dobrar** frequência de sincronização

Sincronização de relógios

- GPS (*Global Positioning System*)
  - **Trilateração**: determinar a distância do receptor em termos dos eixos x, y e z em relação a cada um dos satélites.
  - Relógios dos satélites e receptores precisam estar sincronizados
  - sincronizar os relógios é exatamente o problema que estamos tentando resolver!!!!
  - Para contornar esta restrição, usa-se um quarto satélite, para determinar a distância na **dimensão do tempo**.

Sincronização de relógios

- GPS (*Global Positioning System*)
  - **Solução 3**: Colocar receptor GPS em cada nó do sistema
    - Erro de 0,1ns a 1ms do UTC
    - Preços de receptores impedem adoção

Sincronização de relógios

- GPS (*Global Positioning System*)
  - **Solução 4**: usar a redes de computadores e sincronizar com outra máquina, que fez o investimento necessário para manter o erro baixo.

  - **Receita**:
    - Pergunte que horas são
    - Use a resposta para ajustar o relógio local
    - Considere o erro introduzido pela latência variável da rede

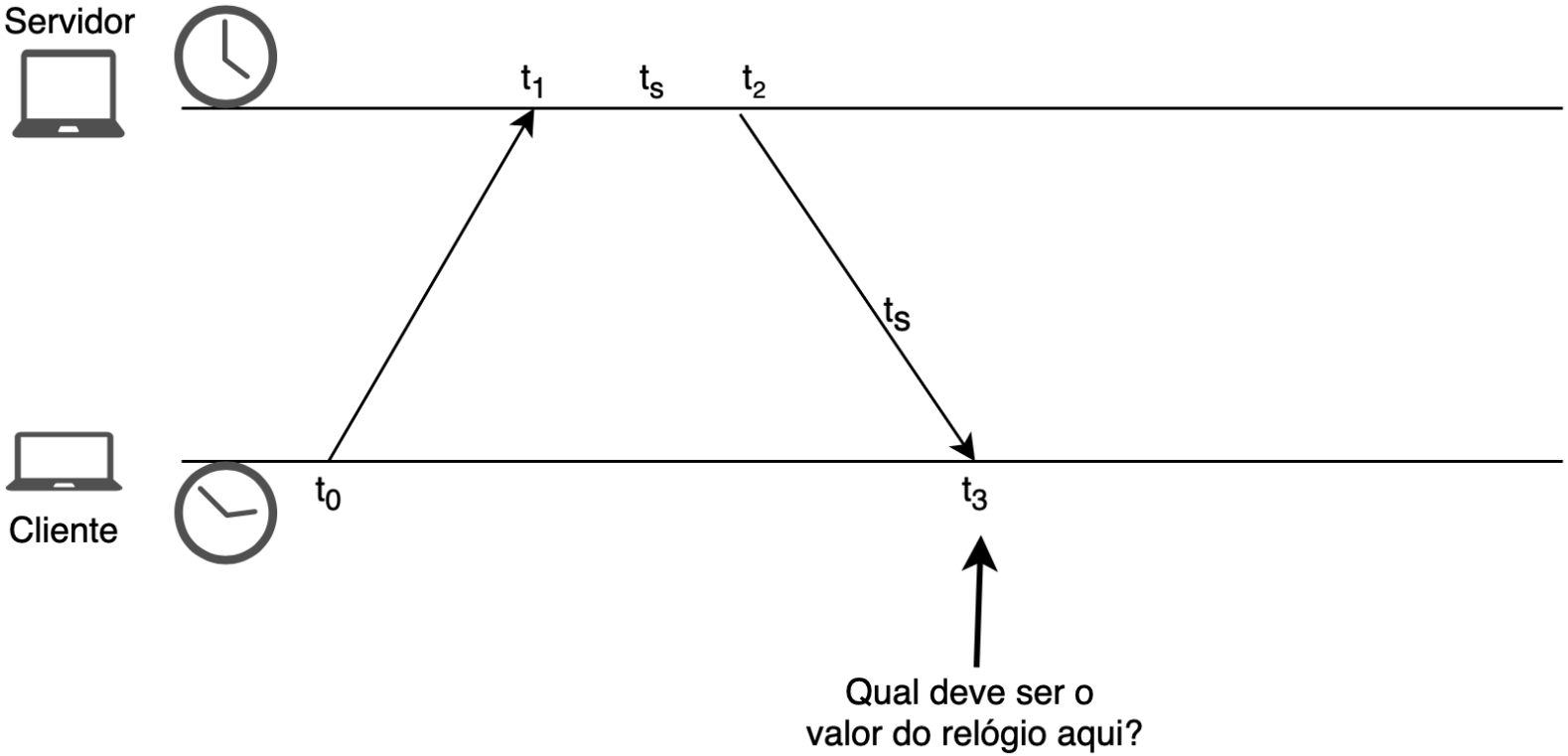
Sincronização de relógios

- GPS (*Global Positioning System*)
  - **Solução 4**: usar a redes de computadores e sincronizar com outra máquina, que fez o investimento necessário para manter o erro baixo.

  - Sendo mais específico:
    - Cliente pergunta "que horas são?" - t0
    - Servidor recebe pergunta - t1
    - Servidor anota o valor do relógio - ts
    - Servidor envia resposta - t2
    - Cliente recebe resposta - t3

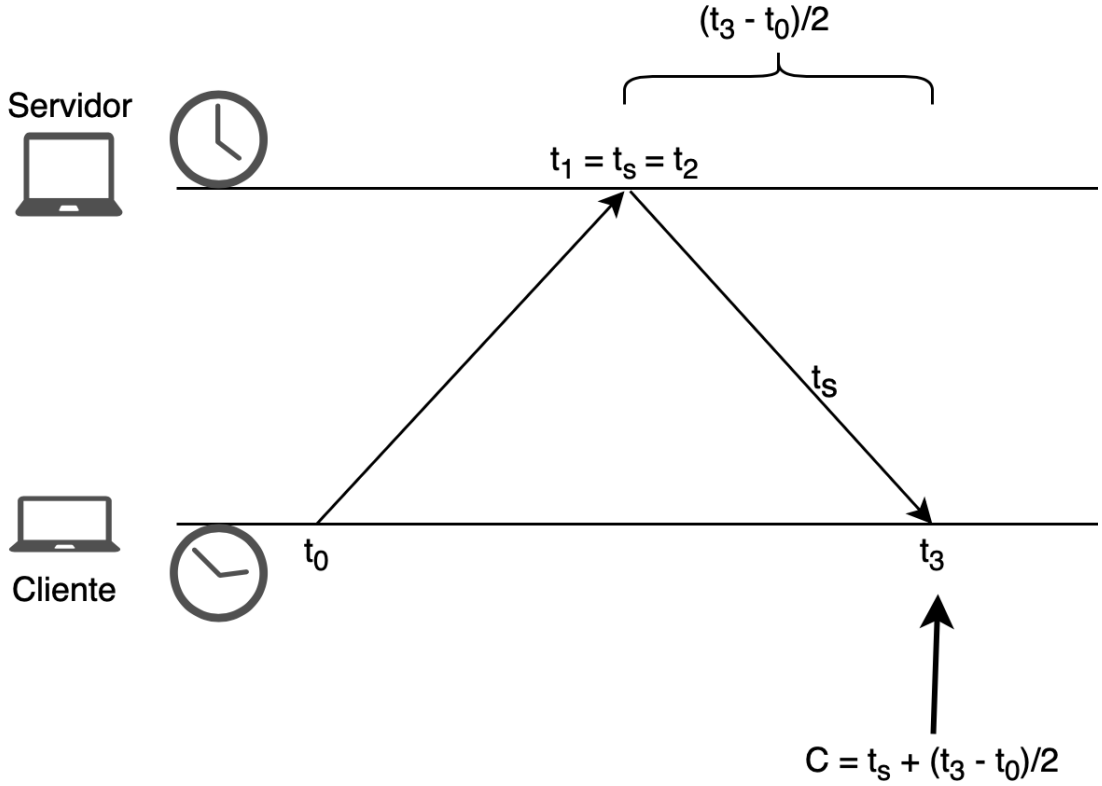
Sincronização de relógios

- GPS (*Global Positioning System*)
  - **Solução 4**: usar a redes de computadores e sincronizar com outra máquina, que fez o investimento necessário para manter o erro baixo



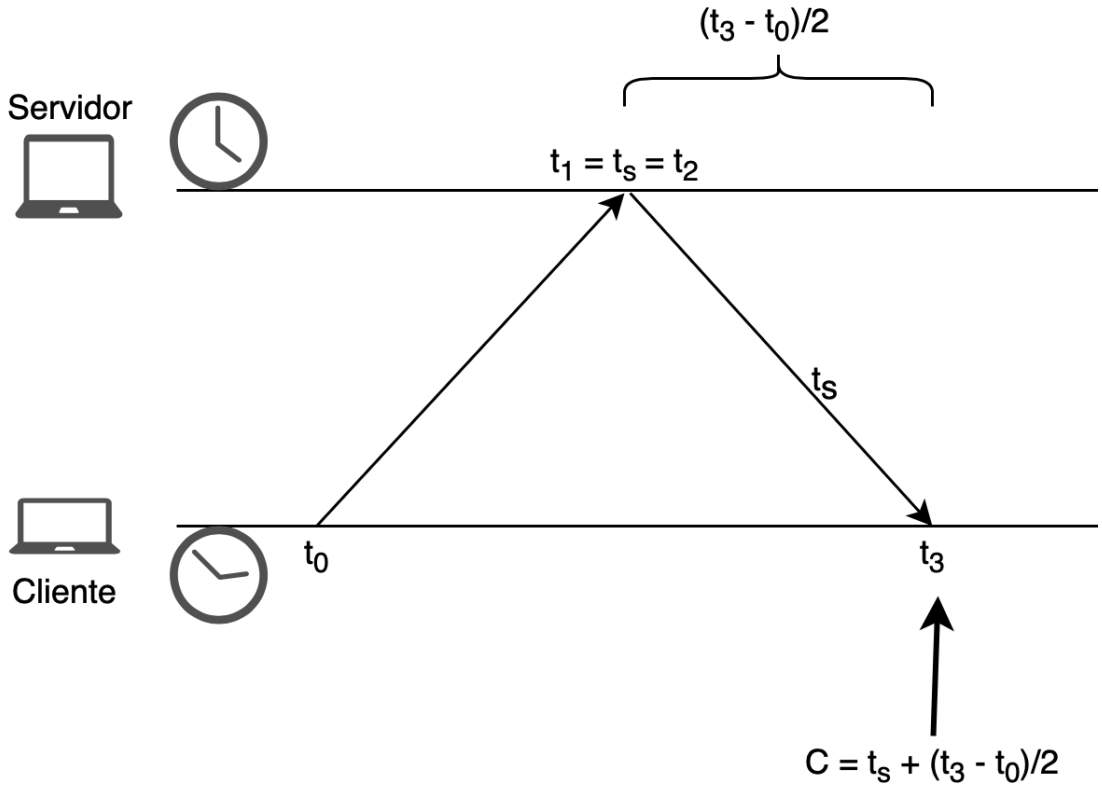
Sincronização de relógios

- Extensão da solução 4 - **Algoritmo de Cristian**
  - Assuma t1 = ts = t2
  - Assuma (t3−t0)/2 como o tempo de transmissão da resposta (média da ida e da volta)
  - Cliente ajusta relógio para: C = ts + (t3-t0)/2



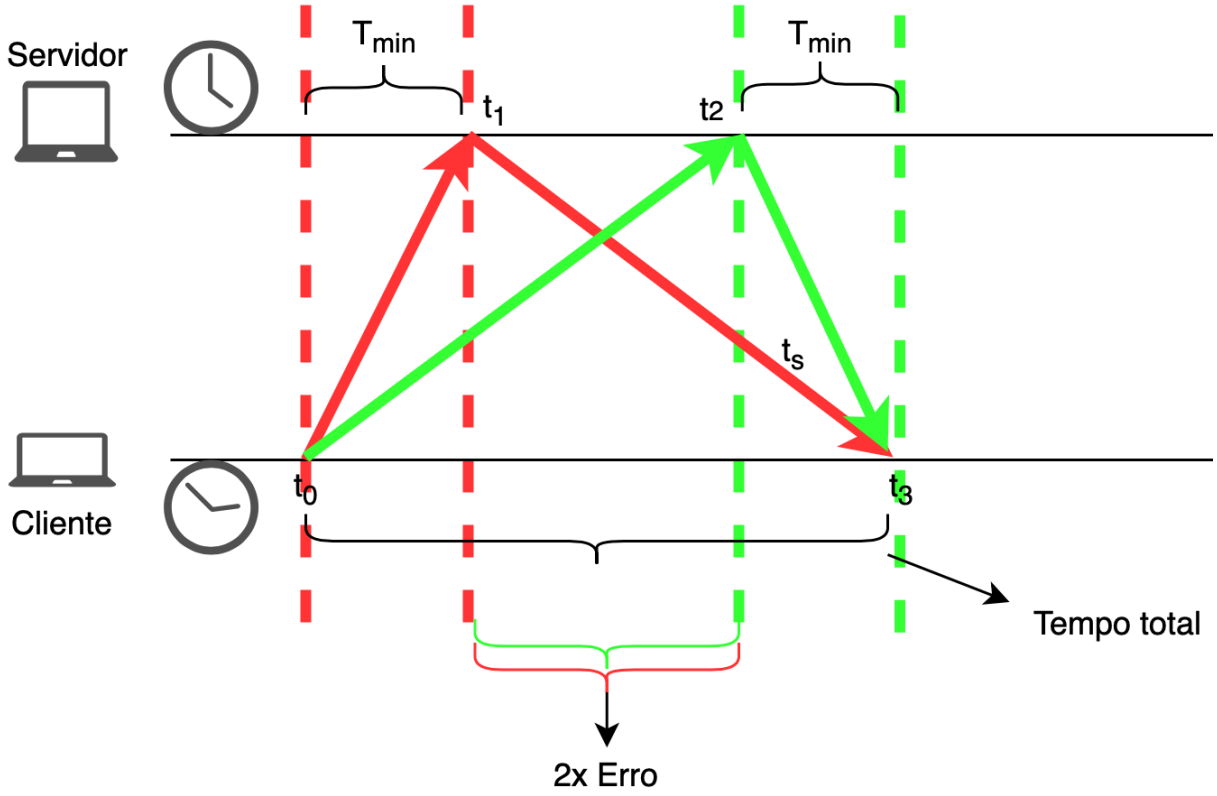
Sincronização de relógios

- Extensão da solução 4 - **Algoritmo de Cristian**
  - Aproximação (t3−t0)/2, é boa?
    - Latência de requisição e resposta podem ser diferentes
    - Problema é que medir a latência em uma única direção demandaria relógios sincronizados (!!!)



Sincronização de relógios

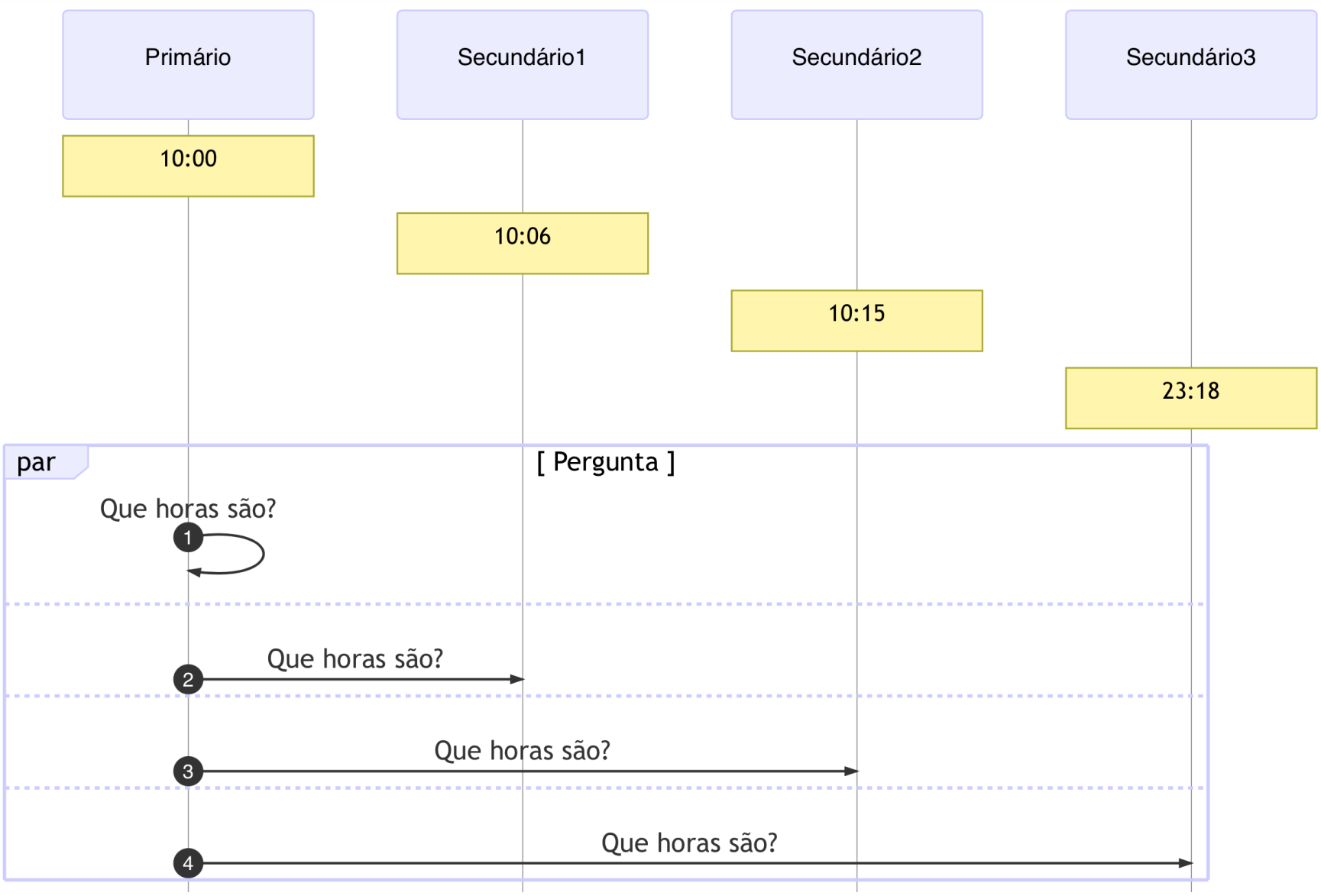
- Extensão da solução 4 - **Algoritmo de Cristian**
  - Aproximação (t3−t0)/2, é boa?
    - Medindo o erro:
      - Necessidade estimativa de tempo mínimo: Tmin
      - E = (t2-t1)/2 = (t3-t0)/2 - Tmin



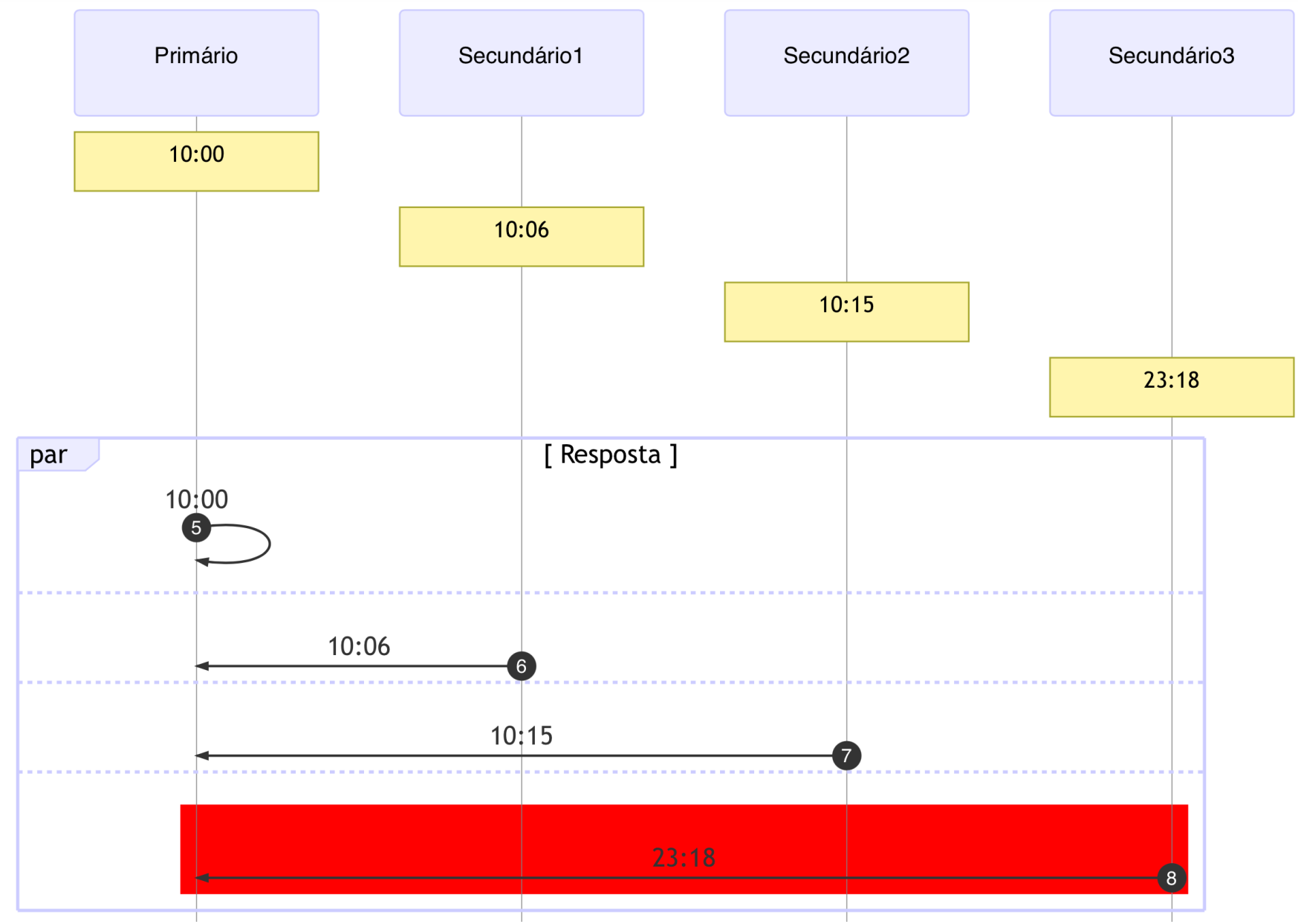
Sincronização de relógios

- **Algoritmo de Berkeley**
- **Ideia básica:**
  - Todo nó executa um "*daemon*" de sincronização
  - 2 papeis: **primário** e **secundário**
  - O papel do primário pode ser rotacionado
- **Algoritmo**:
  - Primário pergunta "que horas são?" para cada secundário
  - Secundário responde com valor atual do relógio
  - Primário ajusta as respostas de acordo com o algoritmo de Cristian, para minimizar erros.
  - Primário computa média dos valores recebidos, ignorando *outliers*
  - Primário envia **ajustes** para secundários
  - Secundário executa ajuste sugerido pelo primário.

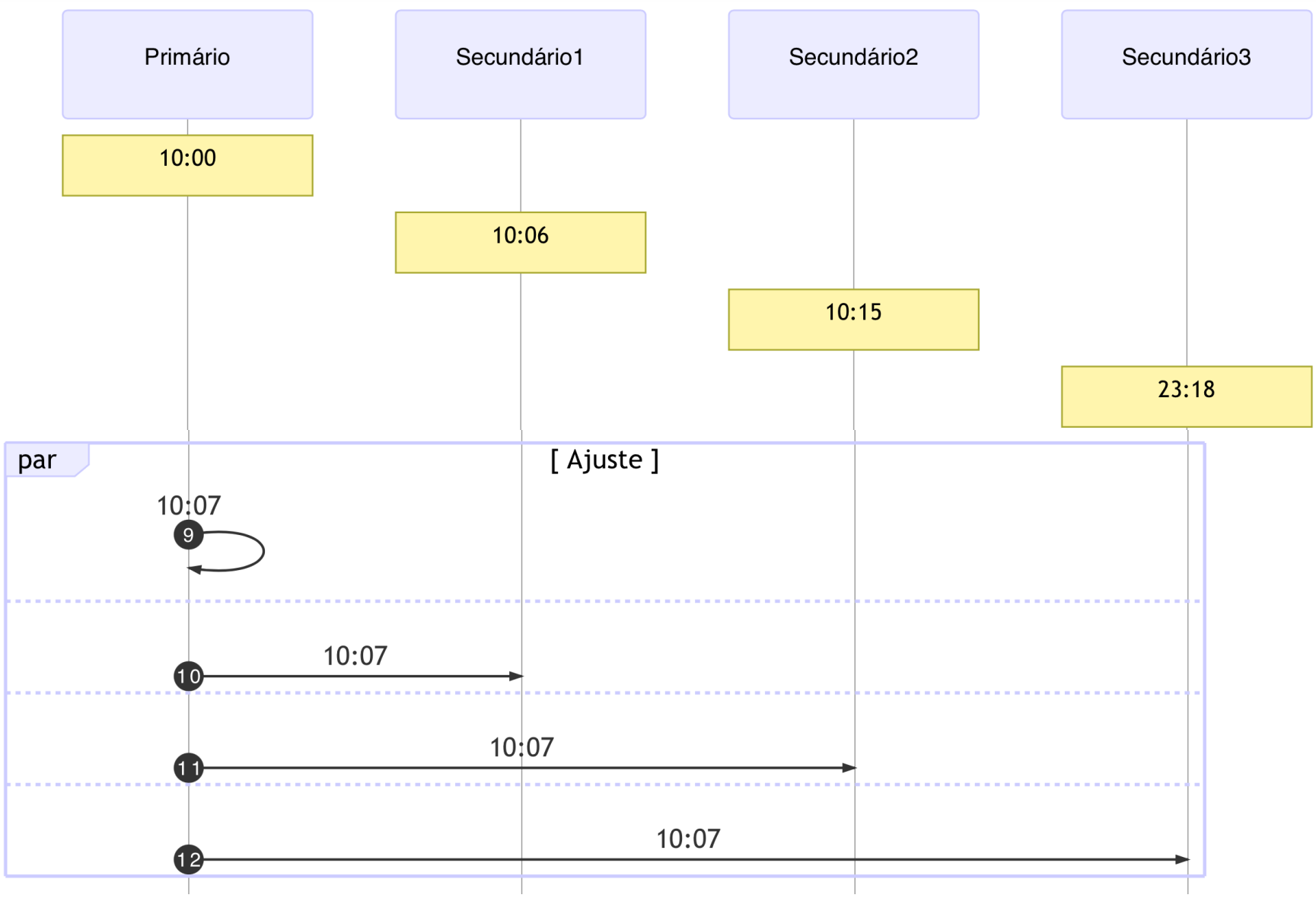
Sincronização de relógios: Algoritmo de Berkeley



Sincronização de relógios: Algoritmo de Berkeley



Sincronização de relógios: Algoritmo de Berkeley



Sincronização de relógios

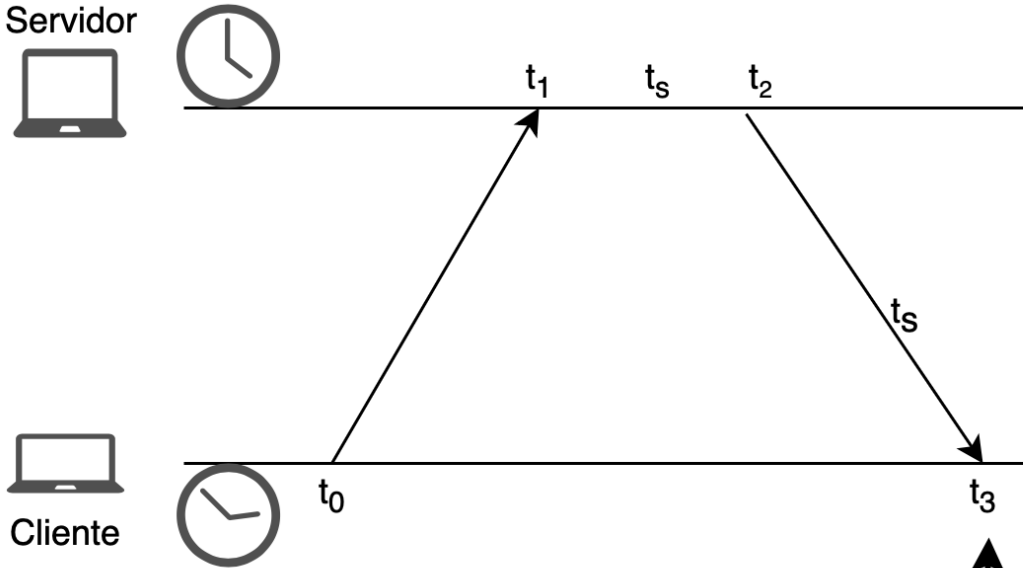
- **NTP (*Network Time Protocol*)**
  - RFCs 1305, 5905-5908 (IPv6, erro <10µs)
- Componentes organizados em camadas (*estrata*)
  - Informação do tempo flui da camada 0 (*stratum 0*) até a camada 15 (*stratum* 15)
- Componentes não estão presos a camadas
  - Novos caminhos são encontrados usando-se o algoritmo de árvore geradora mínima Bellman-Ford, além de caminhos redundantes que conferem propriedades de **tolerância a falhas** à topologia
- Comunicação pode ser **autenticada**
- Múltiplas formas de execução:
  - Modo multicast: propaga tempo em rede local
  - RPC: algoritmo de Cristian
  - Simétrico: parecido com Berkeley

Sincronização de relógios

- **NTP - Organização hierárquica**
  - Permite escalar o uso do protocolo para níveis globais
  - Stratum 0: relógios atômicos/receptores GPS
  - Stratum 1: ms to stratum 0
  - Stratum 2: contata múltiplos stratum 1 e pares
  - Strata 3...15
  - Stratum 16: dessincronizado

Sincronização de relógios

- **SNTP (*Simple Network Time Protocol*)**
  - Adequada aos nós nas folhas
  - Essencialmente o algoritmo de Cristian:
    - t=[(t1−t0)+(t2−t3)]/2 (média do tempo para enviar e receber)
    - tc=t3+t
  - Exemplo:
    - t0=1100, t3=1200, t1=800, t2=850
    - t=((800−1100)+(850−1200))/2=(−300−350)/=−325
    - tc=875



Sincronização de relógios

- **Nunca volte no tempo**

- Pode levar a situações estranhas como um dado ter data de edição anterior a data de criação
- Para evitar estas situações:
  - **Ajustes graduais** nos relógios:
    - **Ajustar frequência de interrupção para atrasar/adiantar relógio**
      OU
    - **Ajustar dos incrementos com cada interrupção**

Uso de relógios sincronizados

- Há uma série de problemas interessantes que podem ser resolvidos:
  - **Autenticação**, **terminação de transações**, **alocação de *leases***
- Exemplo: Ordenação de eventos em BD
  - Sistema Bancário replicado
  - 2 cópias em lados opostos de uma rede de larga escala Clientes disparam operações como saques, depósitos e transferências, por meio de mensagens para as duas cópias
  - Mensagens para a cópia próxima do cliente (em verde) são entregues rapidamente, enquanto mensagens para a cópia distante (em vermelho), demoram mais para ser entregues

Uso de relógios sincronizados

- Exemplo: Sistema Bancário replicado
  - U1 envia C1: atualizar saldo da conta para USD 10
  - U2 envia C2: atualizar saldo da conta para USD 20

Uso de relógios sincronizados

- Exemplo: Sistema Bancário replicado
  - Ao final da execução:
    - R1 terá executado C1 seguido de C2
    - R2 terá executado C2 seguido de C1
  - Como resolver?

Uso de relógios sincronizados

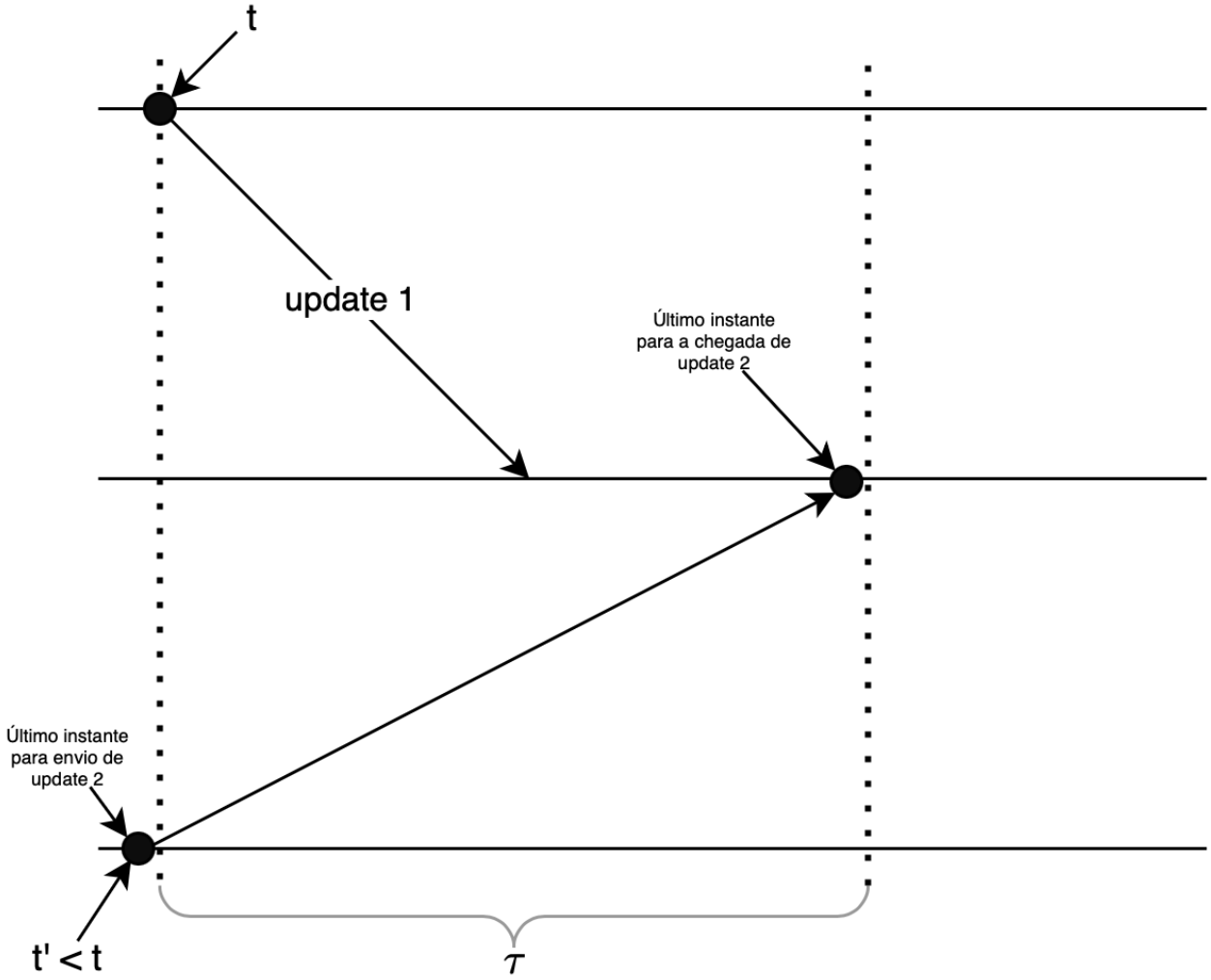
- Exemplo: Sistema Bancário replicado
  - E se as réplicas processarem mensagens na ordem que foram enviadas, como identificado pelos seus *timestamps*?
    - Resolve parciamente

Uso de relógios sincronizados

- Exemplo: Sistema Bancário replicado
  - E se as réplicas processarem mensagens na ordem que foram enviadas, como identificado pelos seus *timestamps*?
  - Como identificar que nenhuma outra mensagem ainda por ser entregue com timestamp menor foi enviada antes?
  - Devemos estender:
    - **τ:** Tempo de propagação máximo de uma mensagem
    - Finito e conhecido

Uso de relógios sincronizados

- Exemplo: Sistema Bancário replicado
  - Ordenação de mensagens por *timestamp*



Uso de relógios sincronizados

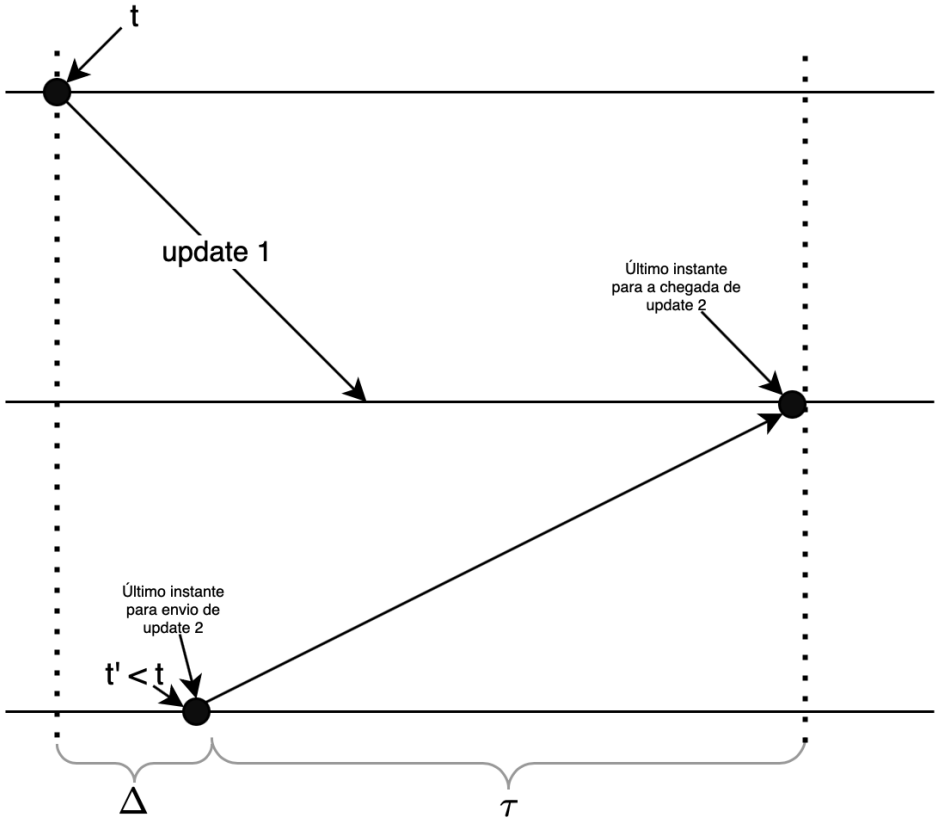
- Exemplo: Sistema Bancário replicado

Ordenação de Mensagens por Timestamp

- Quando enviar uma mensagem, aumente-a com o valor atual do relógio.
- Quando receber uma mensagem, coloque-a em uma fila ordenada por *timestamp*.
- Quando o relógio marcar um tempo maior que $t + \tau$, onde t é o timestamp da mensagem na cabeça da fila, retire a mensagem da cabeça da fila e execute o comando correspondente.

Uso de relógios sincronizados

- Exemplo: Sistema Bancário replicado
  - Ordenação de mensagens por *timestamp*
  - E se os relógios se dessincronizarem?
    - Basta definir um limite Δ e sincronizar os relógios a cada Δ/2ρ
  - Como fica a espera neste caso?
    - t+τ+Δ



Uso de relógios sincronizados

- Nó pode potencialmente esperar por **muito** tempo antes de usar um recurso
- E se ele aprendesse que os outros nós não farão requisições?
- E se houvesse um relógio que avançasse não com o tempo, mas com eventos interessantes do sistema?
- Esta é a ideia dos **relógios lógicos**

Tempo Lógico

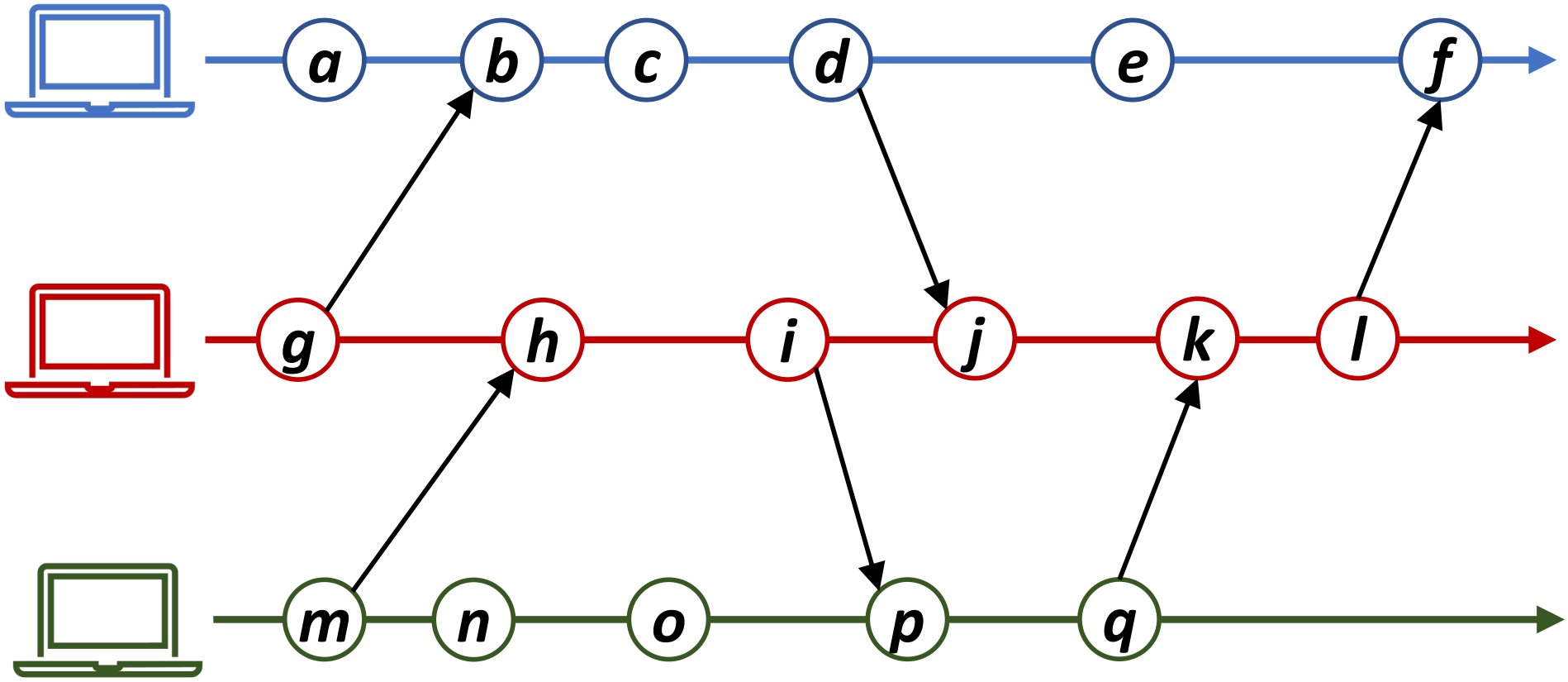
- O que importa são eventos e não a passagem do tempo, uma vez que tempo físico é relativo aos processos
- Conceito apresentando por Leslie Lamport:
  - Time, Clocks and the Ordering of Events in a Distributed System. July 5, 1978
  - Captura relação de **causalidade** entre eventos (happened-before)

Tempo Lógico

- **Causalidade**
  - Se um evento *e* aconteceu antes (*happened-before*) um evento *e'*, então *e* (potencialmente) causou *e'*
  - Diz-se ainda que *e* precede *e'*
  - Notação: $e \rightarrow e'$
- Formalização:
  - Evento *a* aconteceu antes de evento *b'* ($a \rightarrow b$) se uma das condições a seguir é válida:
    1. Se *a* e *b* são eventos em um mesmo processo e *a* foi executado antes de *b*;
    2. Se *a* e *b* são eventos de processos distintos e *a* é o envio de uma mensagem e *b* é a sua recepção;
    3. Se há transitividade, isto é, se existe um evento *c*, tal que $a \rightarrow c$ e $c \rightarrow b$

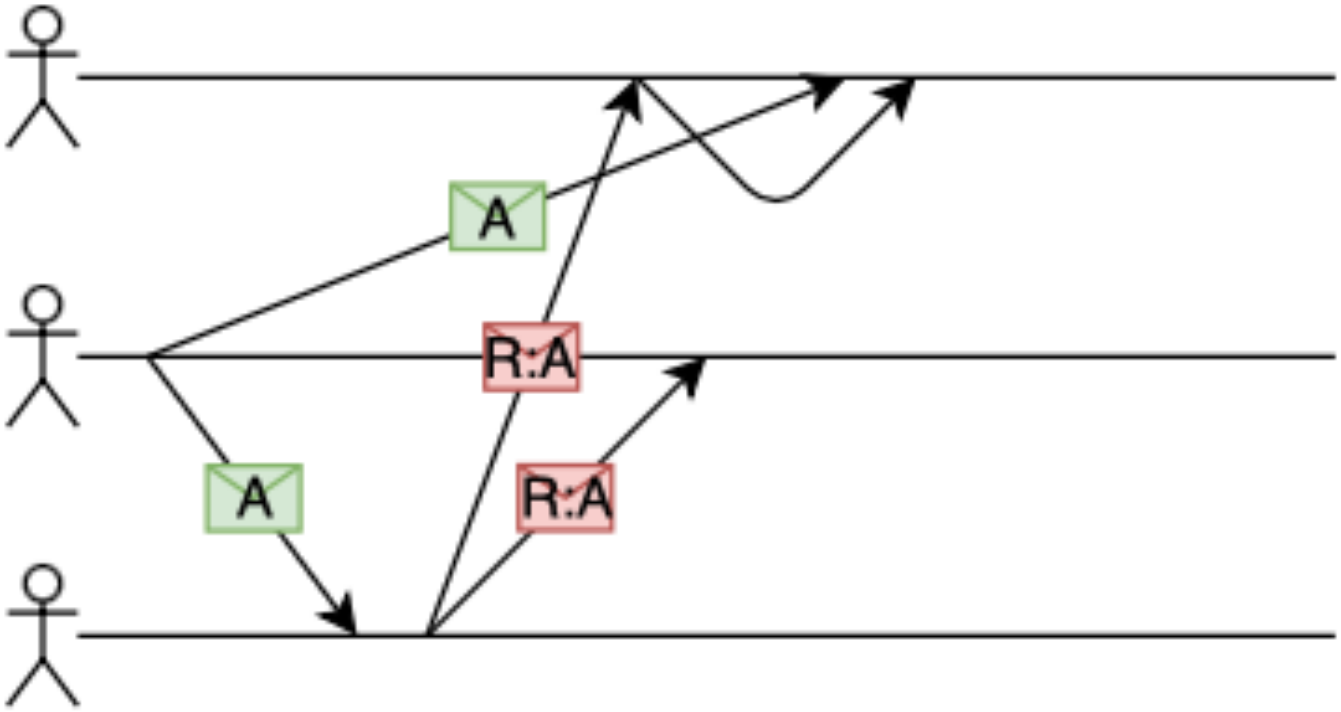
Tempo Lógico

- **Causalidade e concorrência**



Tempo Lógico

- Mas para que serve a causalidade?
  - Exemplo: leitura de mensagem
    - A → R:A
    - Cliente de cima não precisa saber **quando** msgs foram enviadas
    - Apenas a **ordem** das mensagens é suficiente
    - Como representar esta causalidade?



Relógios Lógicos

- Como capturar causalidade?
- Proposta de Lamport – **Relógio lógico**
  - A.k.a Relógio de Lamport
  - Permite associar um *timestamp* a eventos
  - Garante a seguinte propriedade:
    - seja *e* um evento
    - seja *C(e)* o valor do relógio lógico associado a *e*
    - se $e \rightarrow e'$ então $C(e) < C(e')$
- Mas como definir a função C?

Relógios Lógicos

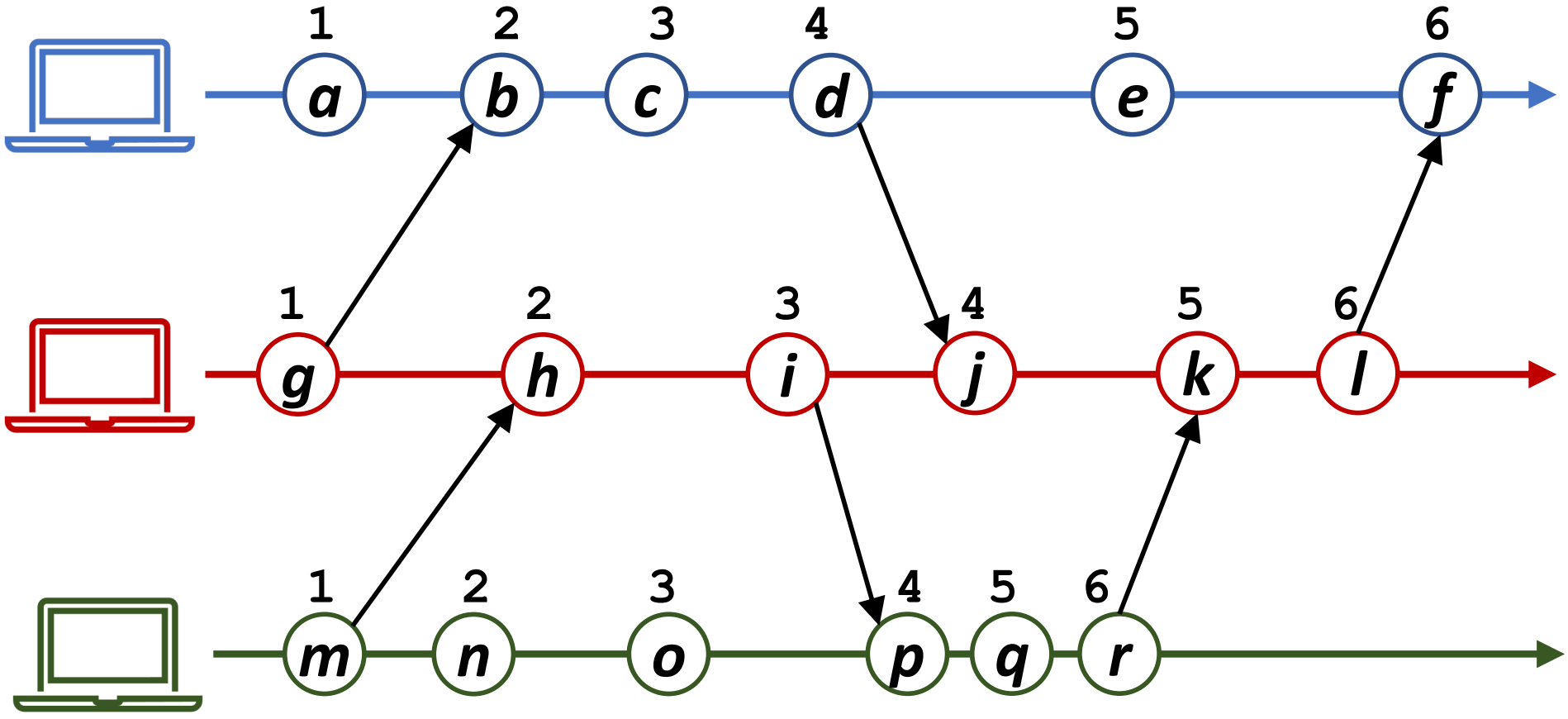
- Mas como definir a função C?
  - Considere a seguinte definição:

Quase Relógio de Lamport

- Seja c_p um contador em p com valor inicialmente igual a 0.
- $C(e) = ++c_p$ no momento em que e ocorreu.
- Usamos como $<$ a relação normal de inteiros.

Tempo Lógico

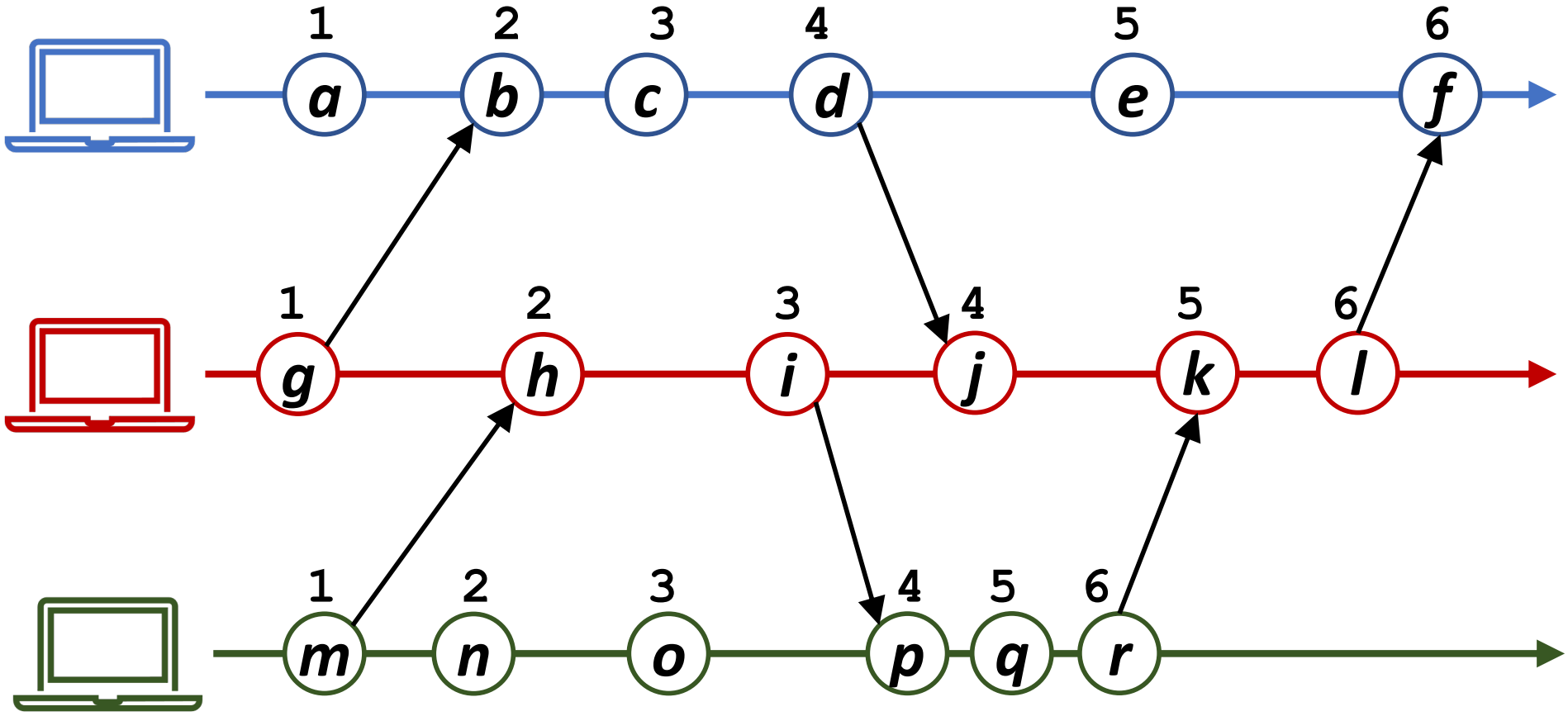
- **Quase-relógio de Lamport**



Problema?

Tempo Lógico

- **Quase-relógio de Lamport**



Sim: $r \rightarrow k$, mas $C(r) > C(k)$

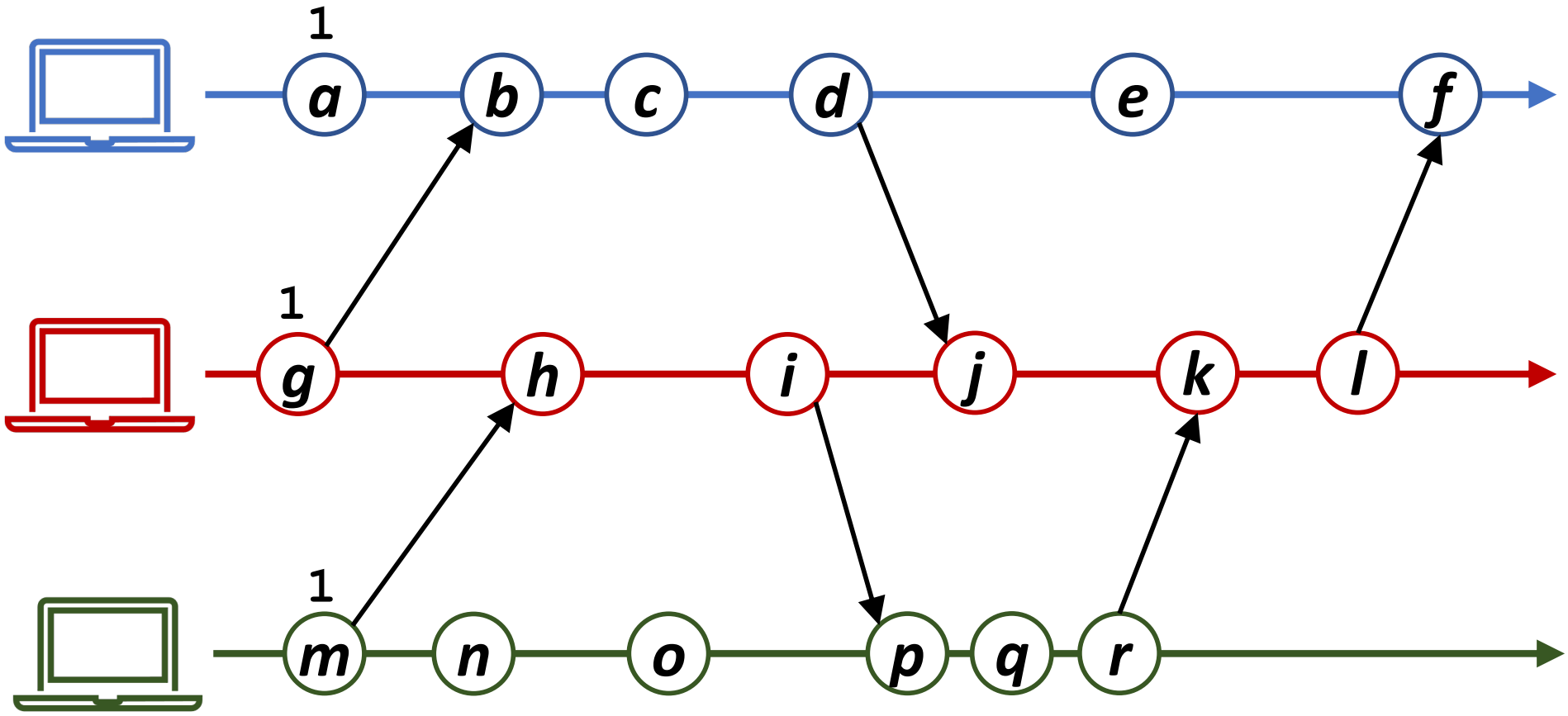
Tempo Lógico – Relógio de Lamport

- Mas como definir a função C?
  - Nova definição:

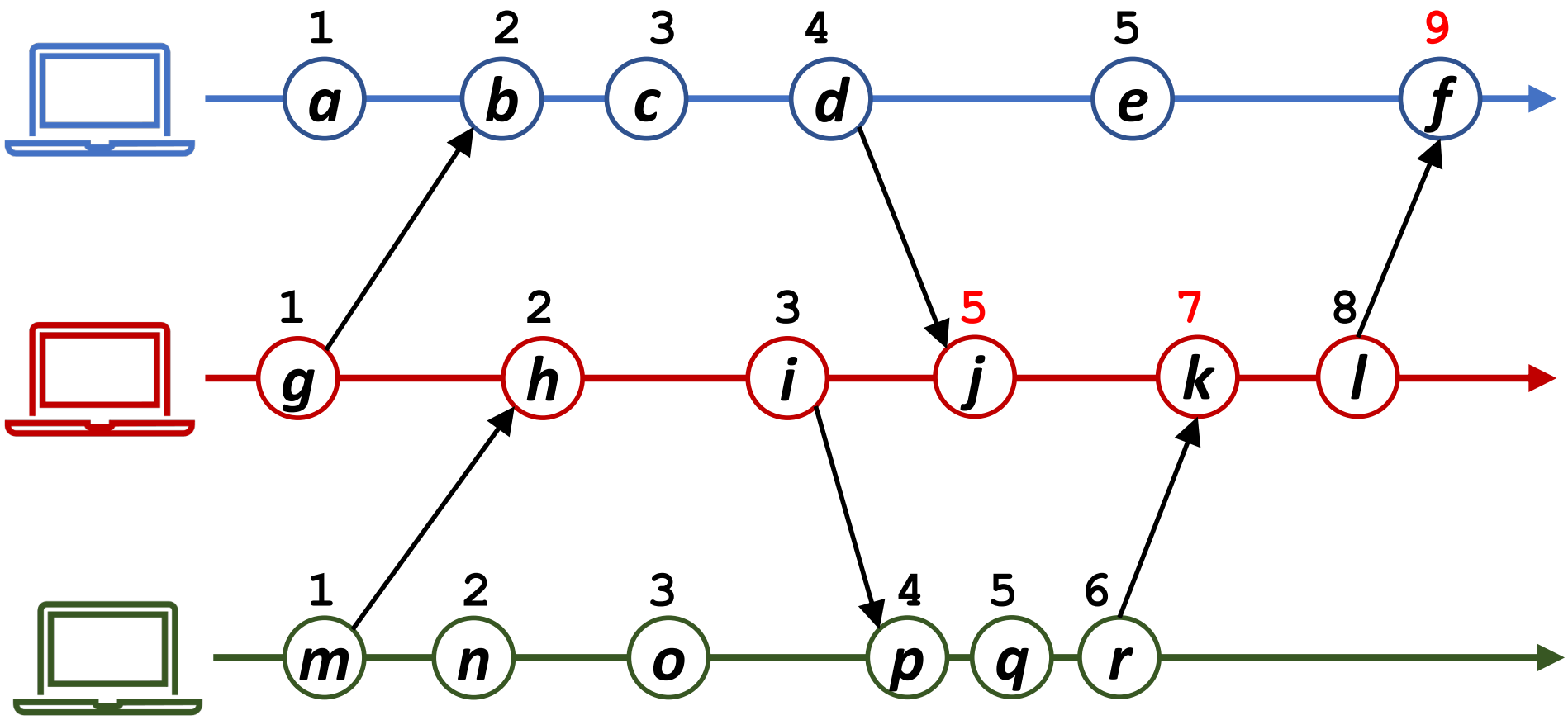
Relógio de Lamport

- Seja c_p um contador em p com valor inicialmente igual a 0.
- Se o evento e é uma operação local
  - $c_p \leftarrow c_p + 1$
  - $C(e) \leftarrow c_p$
- Se o evento e é o envio de uma mensagem
  - $c_p \leftarrow c_p + 1$
  - $C(e) \leftarrow c_p$
  - $C(e)$ é enviado com a mensagem como seu *timestamp*.
- Se o evento e é a recepção de uma mensagem com *timestamp* ts
  - $c_p \leftarrow max(c_p, ts) + 1$.
  - $C(e) \leftarrow c_p$

Tempo Lógico – Relógio de Lamport



Tempo Lógico – Relógio de Lamport



Para todo par de evento *e*, *e'*, temos:

$e \rightarrow e' \implies C(e) < C(e')$

Tempo Lógico – Relógio de Lamport

- **Relógio de Lamport**

$$a \rightarrow b \implies C(a) < C(b)$$

Contrário não é verdade:

$$C(a) < C(b) \implies a \rightarrow b \text{ [mentira!]}$$

- Como saber se dois eventos são concorrentes ou existe precedência apenas pelo valor do relógio?
- Utilizando **relógio vetorial**

Tempo Lógico – Relógio Vetorial

- Vetor de relógios lógicos:

Cada processo mantém próprio contador

+

"Visão" dos contadores dos outros processos

- "Visões" são atualizadas a cada recepção de mensagem
- Ex: cenário com n = 5 processos, p_0 a p_4
  - Possível relógio vetorial (*vector clock*) para processo p_2:
  - c_{p2} = 1,0,2,3,1

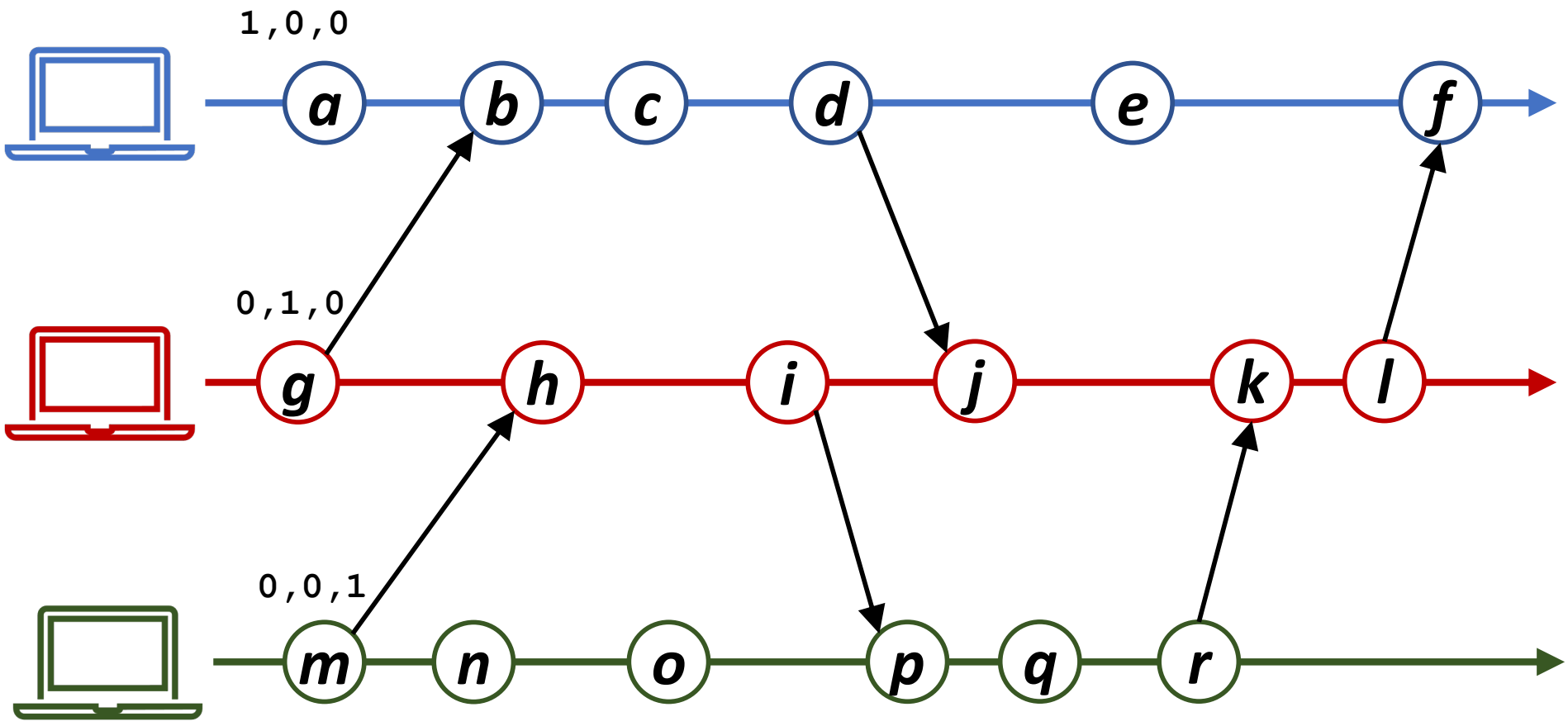
Tempo Lógico – Relógio Vetorial

Relógio Vetorial

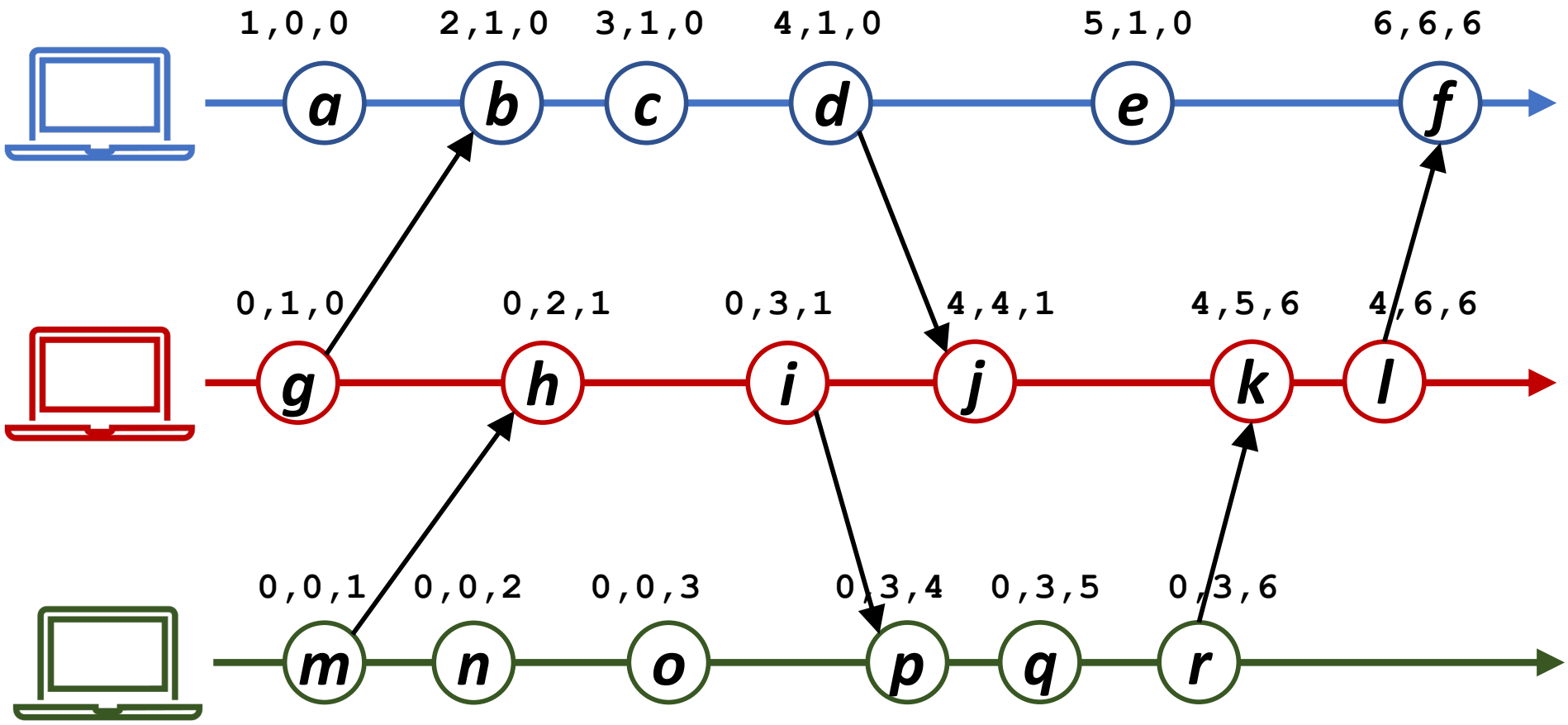
Considerando o ponto de vista do processo p

- Seja $c_p[i], 1 \leq i \leq n$ inicialmente igual a 0
- Seja um evento e
  - Se e é uma operação local
    - $c_p[p] \leftarrow c_p[p] + 1$
    - $V(e) \leftarrow c_p$
  - Se e é o envio de uma mensagem
    - $c_p[p] \leftarrow c_p[p] + 1$
    - $V(e) \leftarrow c_p$
    - $V(e)$ é enviado com a mensagem como seu timestamp.
  - Se e é a recepção de uma mensagem com timestamp ts de q, então
    - $c_p[p] \leftarrow c_p[p] + 1$
    - $c_p[i] \leftarrow max(c_p[i], ts[i]), \forall i \neq p$
    - $V(e) \leftarrow c_p$

Tempo Lógico – Relógio de Lamport



Tempo Lógico – Relógio de Lamport



Para todo par de evento *e, e'*, temos:

$e \rightarrow e' \Leftrightarrow C(e) < C(e')$

Tempo Lógico – Relógios Híbridos

- Relógios lógicos:
  - ignoram a passagem do tempo
  - Importam-se com ordem de eventos.
- Desvantagem quando eventos precisam ser associados a eventos externos ao sistema
  - Exemplo: depuração
    - Suponha que após uma atualização de um sistema, você note um problema nos dados e identifique o evento problemático nos *logs* do sistema, associado ao seu relógio lógico.
    - Como identificar se este evento problemático aconteceu antes ou depois da atualização?
- Relógios híbridos tentam resolver este problema combinando relógios físicos e lógicos

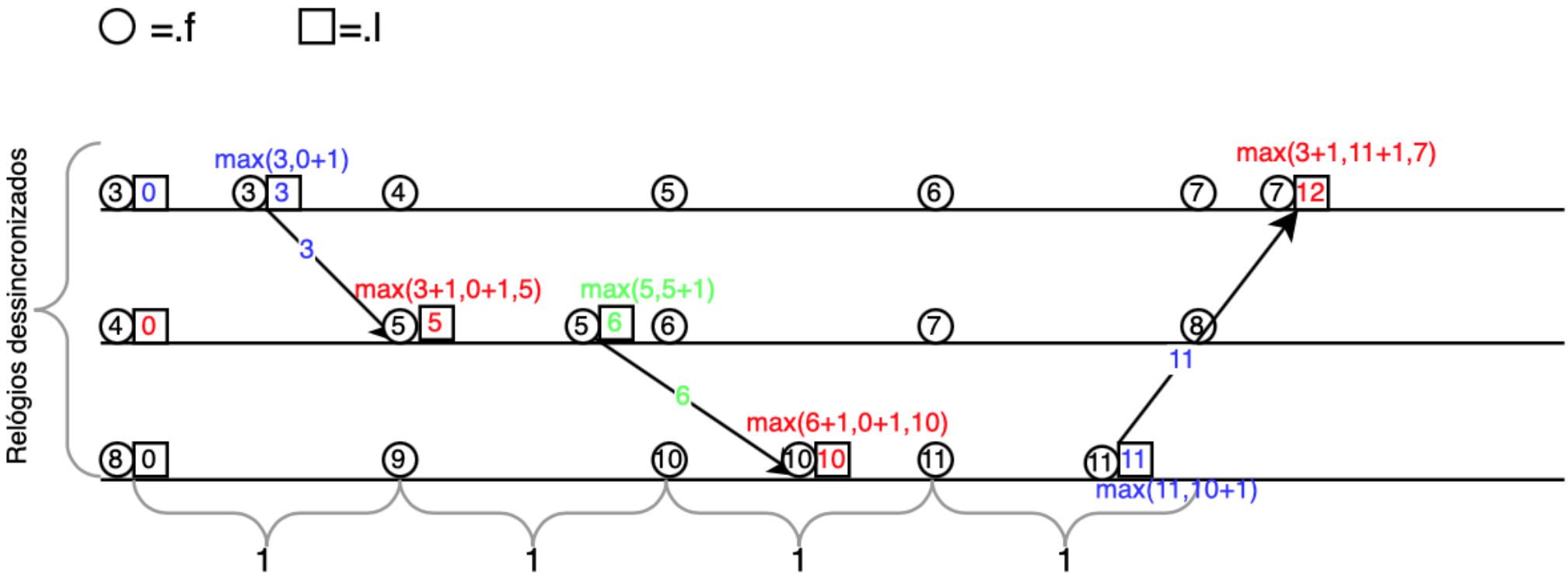
Tempo Lógico – Relógios Híbridos

Relógio Híbrido Simples

Considerando o ponto de vista do processo p

- $c_p.f$ é o relógio físico de p, incrementado automaticamente
- $c_p.l$ é o relógio lógico de p, inicialmente 0
- Seja um evento e
  - Se e é uma operação local
    - $c_p.l \leftarrow max(c_p.l + 1, c_p.f)$
    - $H(e) \leftarrow c_p.l$
  - Se e é o envio de uma mensagem
    - $c_p.l \leftarrow max(c_p.l + 1, c_p.f)$
    - $H(e) \leftarrow c_p.l$
    - $H(e)$ é enviado com a mensagem como seu timestamp.
  - Se e é a recepção de uma mensagem com timestamp ts de q, então
    - $c_p.l \leftarrow max(c_p.l + 1, ts + 1, c_p.f)$
    - $V(e) = c_p.l$

Tempo Lógico – Relógios Híbridos



Tempo Lógico – Relógios Híbridos

- Se “excesso” de eventos:
  - Valor do relógio lógico pode ser incrementado muito rapidamente
  - Perde-se relação com o relógio físico

- Versão melhorada do algoritmo (vide material):
  - distância entre os dois relógios é limitada

Comunicação em Grupo

- Um processo envia mensagens para um conjunto de processos
- **Difusão Totalmente Ordenada** (*Total Order Multicast*):
  - **Difusão**: mensagens são enviadas de 1 para *n*
  - **Totalmente Ordenada**: todos os processos entregam as mensagens na mesma **ordem**
- **Difusão Causalmente Ordenada**:
  - **Causalmente Ordenada**: uma mensagem só é entregue se todas as que causalmente a precedem já foram entregues
- Como resolver o problema do *cloud-drive* com estas abstrações?

Comunicação em Grupo

- **Replicação de Máquinas de Estado** (*State Machine Replication*):
  - Programa se comporta de forma determinística
  - O que acontece se tivermos **várias cópias** deste programa, executando em locais distintos, mas garantirmos que cada cópia veja exatamente a **mesma entrada de dados?**

Comunicação em Grupo

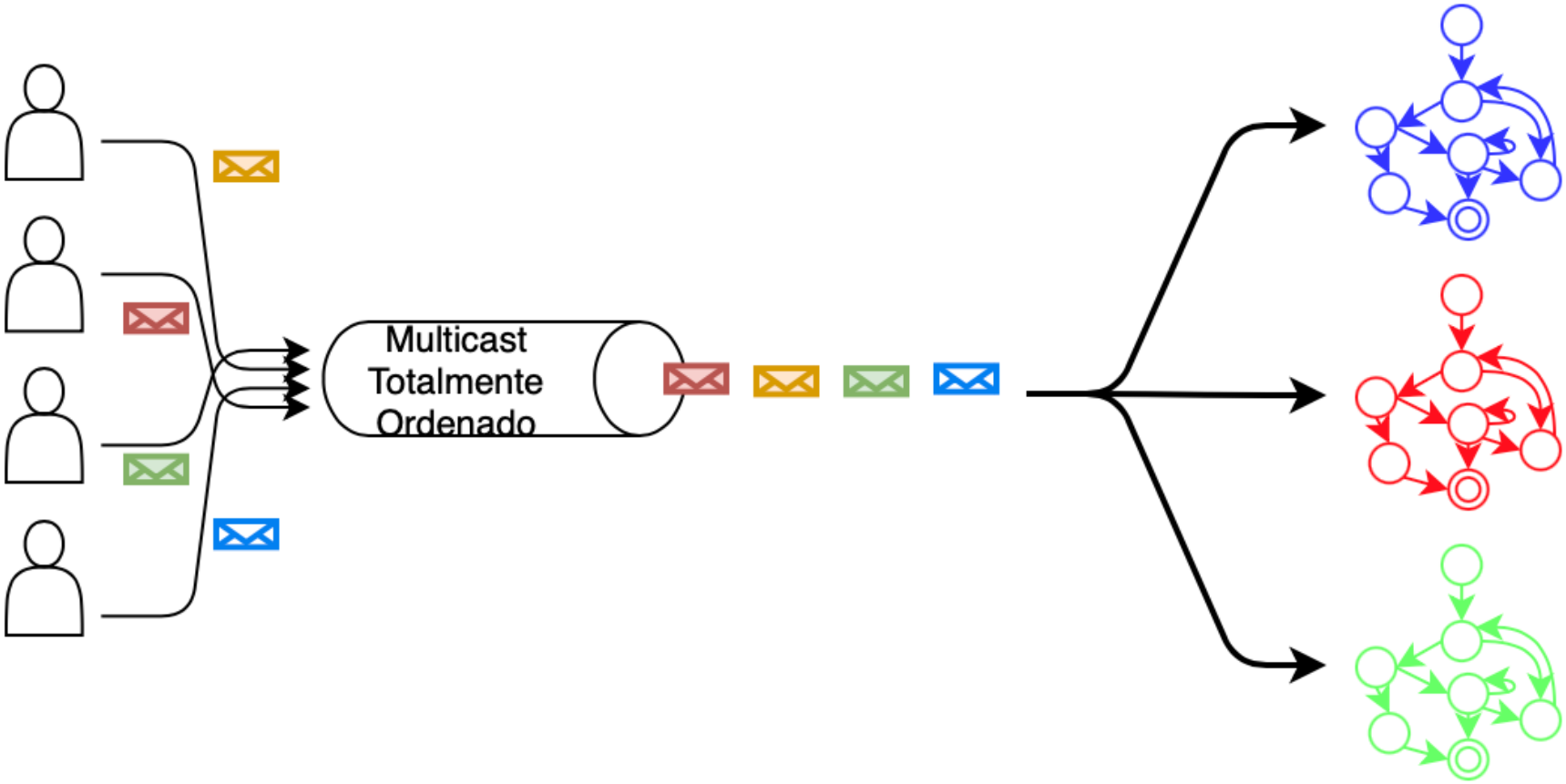
- **Replicação de Máquinas de Estado** (*State Machine Replication*):
  - Programa se comporta de forma determinística
  - O que acontece se tivermos **várias cópias** deste programa, executando em locais distintos, mas garantirmos que cada cópia veja exatamente a **mesma entrada de dados?**
  - Todas as cópias **transitarão pelos mesmos estados** e chegarão ao mesmo estado final
  - Como garantir esta ordem?

Comunicação em Grupo

- **Replicação de Máquinas de Estado** (*State Machine Replication*):
  - Programa se comporta de forma determinística
  - O que acontece se tivermos **várias cópias** deste programa, executando em locais distintos, mas garantirmos que cada cópia veja exatamente a **mesma entrada de dados?**
  - Todas as cópias **transitarão pelos mesmos estados** e chegarão ao mesmo estado final
  - Como garantir esta ordem?
    - Com a **difusão totalmente ordenada**

Comunicação em Grupo

- **Replicação de Máquinas de Estado** (*State Machine Replication*):



Difusão totalmente ordenada

- Canais devem ser FIFO e entrega garantida

Difusão Totalmente Ordenado

- Considerando o ponto de vista do processo p
- f_p é uma fila de mensagens ordenadas pelo seus *timestamps*, mantida em p
- Para difundir uma mensagem m
  - colocar m na fila
  - enviar m para todos os demais processos
- Quando uma mensagem m é recebida
  - colocar m na fila
  - se m não é um ack
    - enviar $mack$ de volta ao remetente de m (com *timestamp* maior que de m)
- Seja m a mensagem com *timestamp* ts na cabeça da fila
  - Se para cada processo q, há uma mensagem m' de q com *timestamp* ts' na fila de p tal que $ts < ts'$
    - entregar m para a aplicação

Difusão totalmente ordenada

- Canais devem ser FIFO e entrega garantida

Difusão Totalmente Ordenado

- Considerando o ponto de vista do processo p
- f_p é uma fila de mensagens ordenadas pelo seus *timestamps*, mantida em p
- Para difundir uma mensagem m
  - colocar m na fila
  - enviar m para todos os demais processos
- Quando uma mensagem m é recebida
  - colocar m na fila
  - se m não é um ack
    - enviar $mack$ de volta ao remetente de m (com *timestamp* maior que de m)
- Seja m a mensagem com *timestamp* ts na cabeça da fila
  - Se para cada processo q, há uma mensagem m' de q com *timestamp* ts' na fila de p tal que $ts < ts'$
    - entregar m para a aplicação

E se canais não forem FIFO?

Difusão causalmente ordenada

- Canais devem ser FIFO e entrega garantida

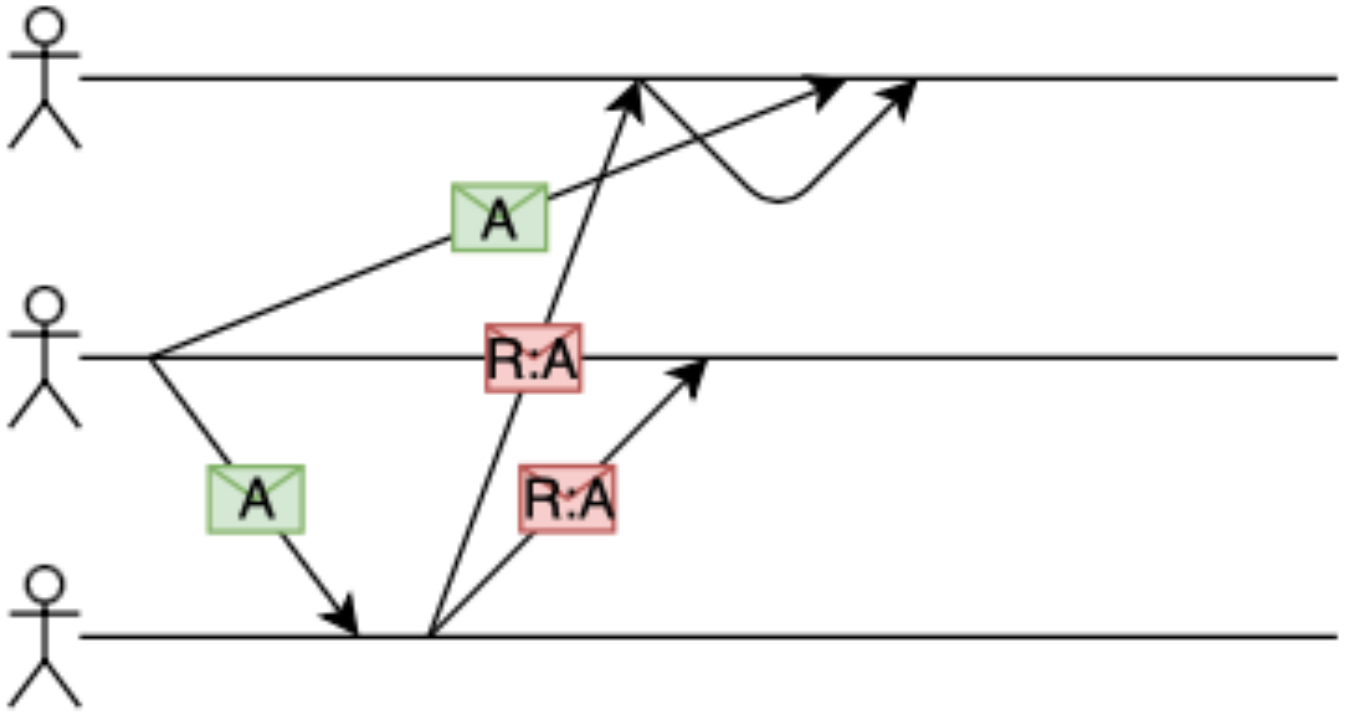
Difusão Causalmente Ordenada

- Considerando o ponto de vista do processo p
- p incrementa $c_p[p]$ somente no envio de mensagens.
- p só entrega uma mensagem recebida de q, com timestamp ts quando
  - $ts[q] = c_p[q] + 1$
  - $ts[k] \leq c_p[k], k \neq q$

Difusão causalmente ordenada

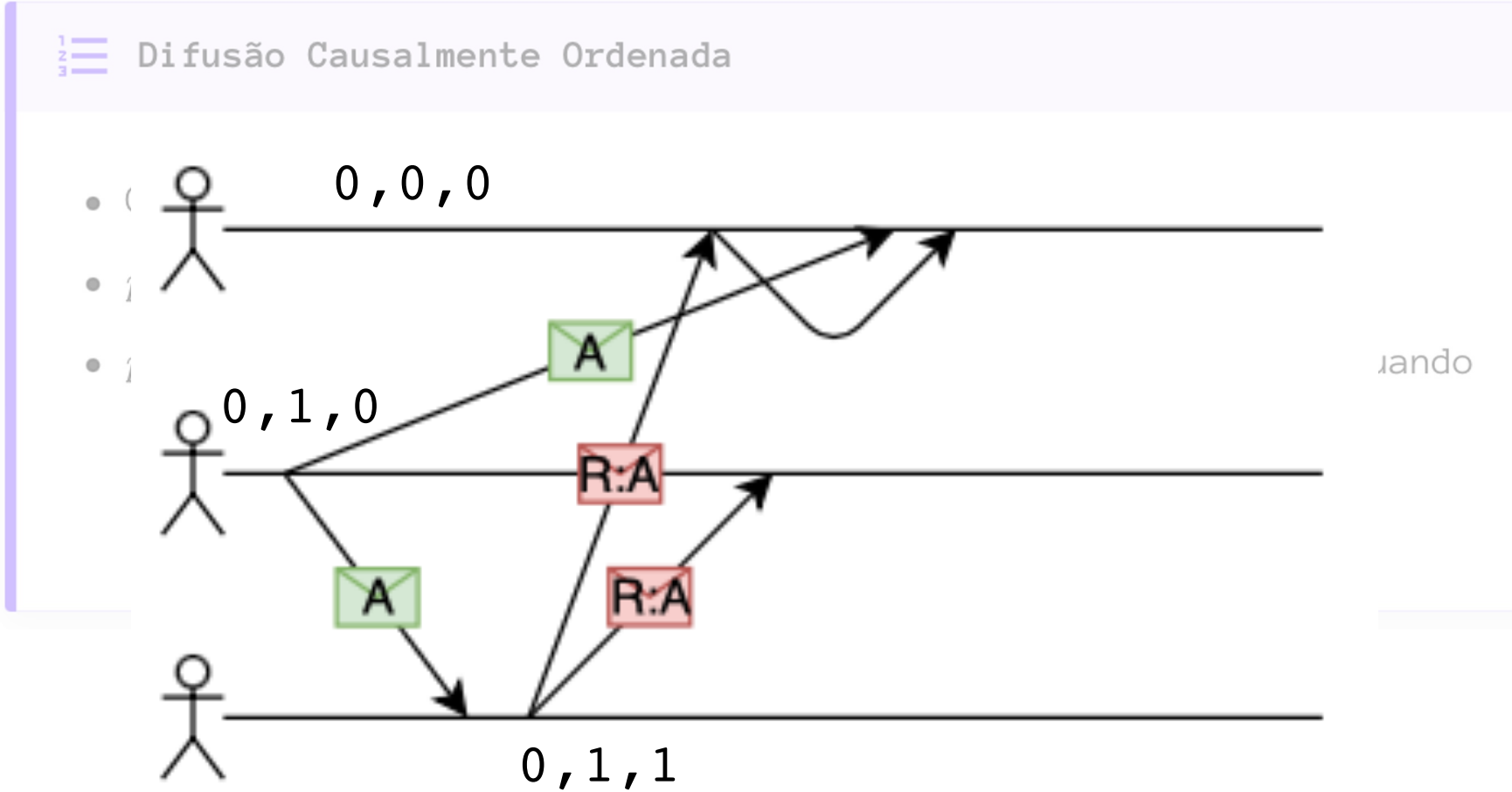
- Canais devem ser FIFO e entrega garantida

Difusão Causalmente Ordenada



Difusão causalmente ordenada

- Canais devem ser FIFO e entrega garantida



Comunicação em grupo - *Middleware*

- Implementação de forma transparente para a aplicação
- Clientes enviam requisições como faziam antes do serviço ser replicado
- Como então as mensagens tem seus relógios lógicos atualizados e usados para a geração de *timestamps*?
  - **Interceptadores** em uma camada de ***middleware***

Comunicação em grupo - *Middleware*