

# Coerência vs Disponibilidade

# Teorema CAP

## PODC 2000 Invited Talk

---

**Wednesday, July 19, 2000**  
**8:30 - 9:30**

**Towards Robust Distributed Systems**  
*Eric A. Brewer*  
*University of California, Berkeley & Inktomi*



Current distributed systems, even the ones that work, tend to be very fragile: they are hard to keep up, hard to manage, hard to grow, hard to evolve, and hard to program. In this talk, I look at several issues in an attempt to clean up the way we think about these systems. These issues include the fault model, high availability, graceful degradation, data consistency, evolution, composition, and autonomy.

These are not (yet) provable principles, but merely ways to think about the issues that simplify design in practice. They draw on experience at Berkeley and with giant-scale systems built at Inktomi, including the system that handles 50% of all web searches.

---

### Biographical Sketch

Eric Brewer is an Associate Professor of Computer Science at UC Berkeley, and the Co-Founder and Chief Scientist of Inktomi Corporation. He is a Global Leader for Tomorrow of the World Economic Forum, and is listed as an Internet leader by Forbes, MIT's Technology Review (TR100), Vanity Fair, Upside and others. Current research includes Internet systems, security and mobile computing.

# Teorema CAP

- É impossível ter simultaneamente:
  - Coerência (consistency)
  - Disponibilidade (availability)
  - E tolerar partições na rede (partition-tolerance)
- Só é possível ter duas destas coisas (à escolha)

# Teorema CAP



Trading Consistency for Availability in Distributed Systems\*

Ken Birman   Roy Friedman

Department of Computer Science  
Cornell University  
Ithaca, NY 14853.

April 8, 1996

# Propagação Epidémica

# Propagação Epidémica

- Vamos considerar um sistema que tenta replicar informação em vários processos da forma menos coordenada possível:
  - Cada processo, periodicamente, contacta outro processo e envia actualizações para os outros nós.
  - Este tipo de propagação de informação é designado por propagação epidémica ou por rumour (do inglês, gossip).
  - Tem a vantagem de ser totalmente descentralizado e de oferecer um bom balanceamento da carga.

# Propagação Epidémica

- Num sistema desta natureza, levantam-se dois tipos de questões:
  - Por que ordem é que se devem aplicar as actualizações que são recebidas de outros nós?
  - Se um cliente contacta uma replica R1 e posteriormente uma réplica R2 que garantias mínimas faz sentido oferecer?

# *Lazy Replication, a.k.a. Gossip architecture*

Exemplo de um sistema replicado otimista



# Providing High Availability Using Lazy Replication

RIVKA LADIN

Digital Equipment Corp.

and

BARBARA LISKOV

LIUBA SHRIRA

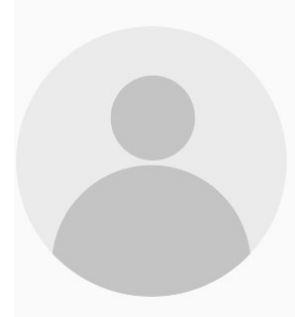
and

SANJAY GHEMAWAT

MIT Laboratory for Computer Science

---

To provide high availability for services such as mail or bulletin boards, data must be replicated. One way to guarantee consistency of replicated data is to force service operations to occur in the same order at all sites, but this approach is expensive. For some applications a weaker causal operation order can preserve consistency while providing better performance. This paper describes a new way of implementing causal operations. Our technique also supports two other kinds of operations: operations that are totally ordered with respect to one another and operations that are totally ordered with respect to all other operations. The method performs well in terms of response time, operation-processing capacity, amount of stored state, and number and size of messages; it does better than replication methods based on reliable multicast techniques.



# Funcionamento base

- Clientes enviam pedidos (leitura ou modificação) a **uma réplica próxima**
- Réplicas **propagam modificações de forma relaxada**, podendo ter vistas divergentes

1. Por que ordem é que se devem aplicar as actualizações que são recebidas de outros nós?

**R: Ordem causal (respeitando a relação “aconteceu-antes”)**

2. Se um cliente contacta uma replica R1 e posteriormente uma réplica R2 que garantias mínimas faz sentido oferecer?

**R: Os clientes observam sempre um estado que é coerente com a relação “aconteceu-antes”**

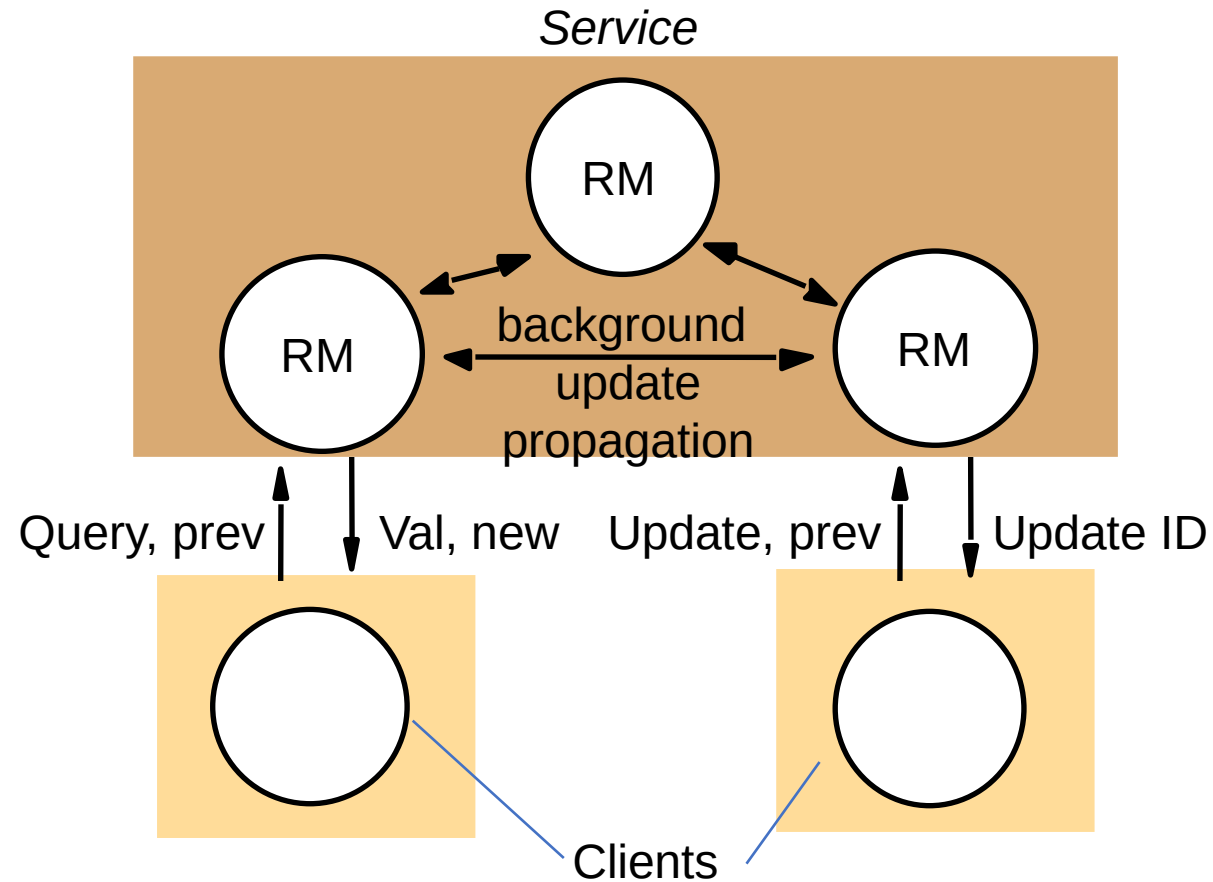
# Como oferecer estas garantias?

- Usando relógios lógicos
  - Possível
  - Mas não eficiente

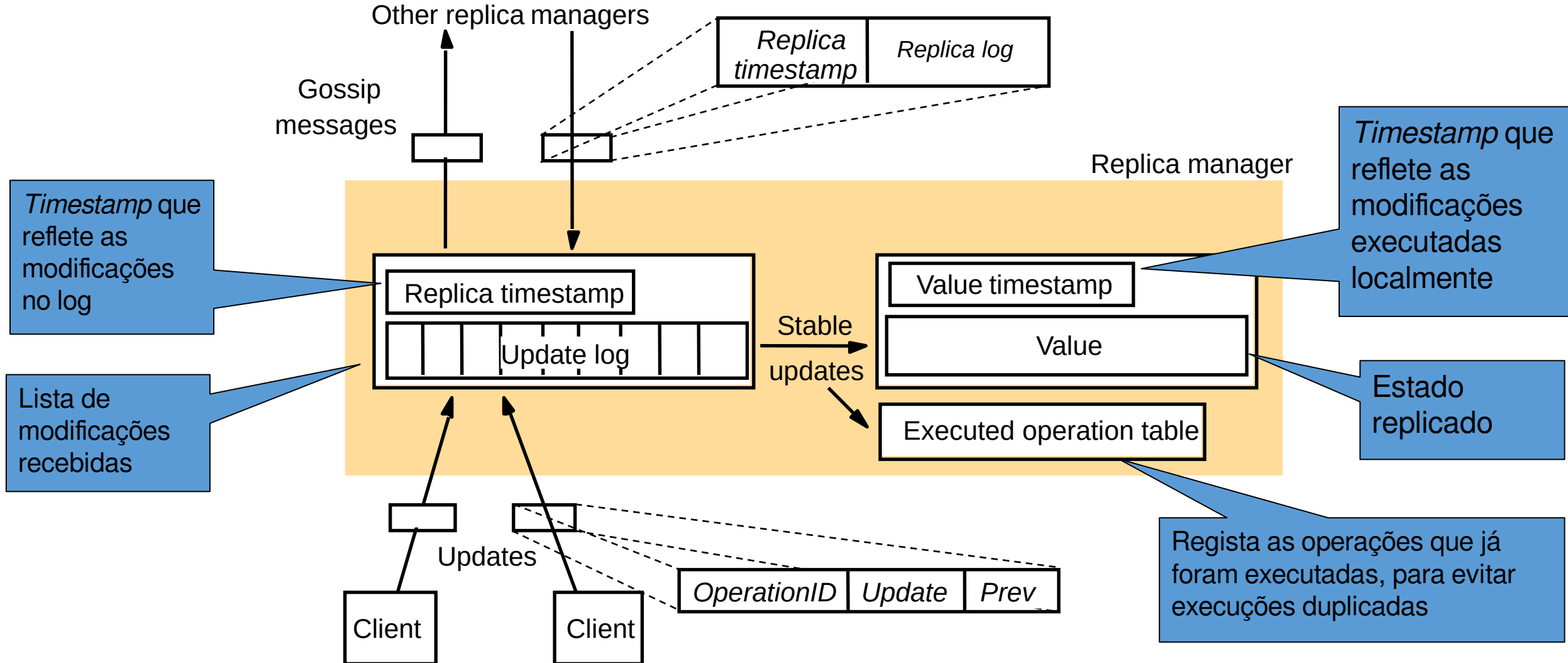
# Algoritmo: interação *cliente* – réplica

- Cada *cliente* mantém um **timestamp vetorial** chamado *prev*
  - Vetor de inteiros, um inteiro por cada réplica
  - Reflete a última versão acedida pelo *cliente*
- Em cada pedido a uma réplica, **cliente envia (pedido, *prev*)**
- Réplica responde com **(resposta, *new*)**
  - *new* é o timestamp vetorial que reflete o **estado da réplica**
  - Se réplica estiver atrasada espera até se atualizar
- Cliente **atualiza *prev* com *new***
  - Para cada entrada *i*, atualiza *prev[i]* se *new[i]* > *prev[i]*

# Algoritmo: interação *cliente* – réplica



# Estado mantido por uma réplica



# Razões para manter *update log*

- Gestor de réplica pode já ter recebido uma modificação mas não a poder executar porque ainda falta receber/executar **dependências causais**
  - Nesse caso, a modificação ainda não é “estável”, portanto está **pendente** no *log* mas ainda não foi executada
- Permite propagar modificações individuais às restantes réplicas
  - Mantêm-se o update no log até se receber confirmação de todas as réplicas

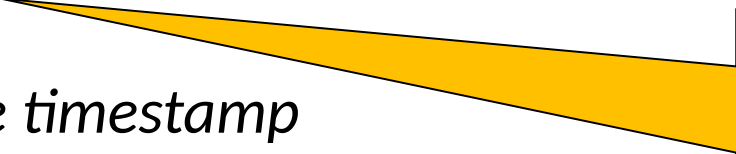


# Pedidos de leitura

- Réplica verifica se *pedido.prev*  $\leq$  *value timestamp*
  - Se sim, retorna o valor atual (junto com o *value timestamp*)
  - Se não, o pedido fica pendente

# Pedidos de modificação

- Quando réplica *i* recebe o pedido vindo do *cliente*:
  - Verifica se não o executou já. Se sim, descarta-o, **caso contrário**:
  - Incrementa a entrada *i* do seu *replica timestamp* em uma unidade
  - Atribui à modificação um novo *timestamp* calculado por:
    - *Timestamp pedido.prev* com a entrada *i* substituída pelo novo valor calculado acima (assim, este timestamp é **único** para este update)
  - Junta a modificação ao *log* e retorna o novo *timestamp* ao *cliente*
  - **Espera até  $\text{pedido.prev} \leq \text{value timestamp}$**  se verificar para executar o pedido localmente
  - Quando executar o pedido, atualiza o *value timestamp*
    - Para cada entrada *i*, atualiza  $\text{valueTS}[i]$  se  $\text{update.TS}[i] > \text{valueTS}[i]$



Garante que a execução respeita a ordem causal

# Propagação de modificações

- Periodicamente, cada gestor de réplica  $i$  contacta outro gestor de réplica  $j$
- $i$  envia a  $j$  as modificações do log de  $i$  **que estima  $j$  não ter**
- Modificações enviadas em ordem
- Para cada modificação que  $j$  recebe:
  - Se não for duplicada, acrescenta-a ao seu log
  - Atualiza o seu replicaTS
  - Assim que  **$prev \leq value timestamp$** , executa a modificação

# Exemplo da Gossip Architecture

# Enquadramento

- Sistema de 2 réplicas,  $R_0$  e  $R_1$ , em que ambas mantêm saldos de contas bancárias
- As contas da Alice e Bob começaram com saldo nulo e depois receberam transferências de uma conta  $S$  (\*)
- Cada operação foi aceite por uma réplica diferente:
  -

(\*) Assuma-se que, no instante inicial, a conta  $S$  começa com saldo suficiente para ambas as transferências

# Exemplo: pedido de escrita

## Réplica $R_0$

- Valores: Alice 0, Bob 0
- valueTS:  $\langle 0, 0 \rangle$
- replicaTS:  $\langle 0, 0 \rangle$
- Update log: [vazio]

## Réplica $R_0$

- Valores: Alice 0, Bob 0
- valueTS:  $\langle 0, 0 \rangle$
- replicaTS:  $\langle 1, 0 \rangle$
- Update log:  $op_{S>Alice}$

## Réplica $R_0$

- Valores: Alice 10, Bob 0
- valueTS:  $\langle 1, 0 \rangle$
- replicaTS:  $\langle 1, 0 \rangle$
- Update log:  $op_{S>Alice}$

Junta a operação ao *update log*, atualizando o TS respectivo, e **responde ao cliente**

Logo depois, observa se  $op.prevTS \leq valueTS$ . Neste caso, a condição verifica-se, logo executa a operação sobre os valores locais. Caso contrário, a operação ficaria pendente (*unstable*).

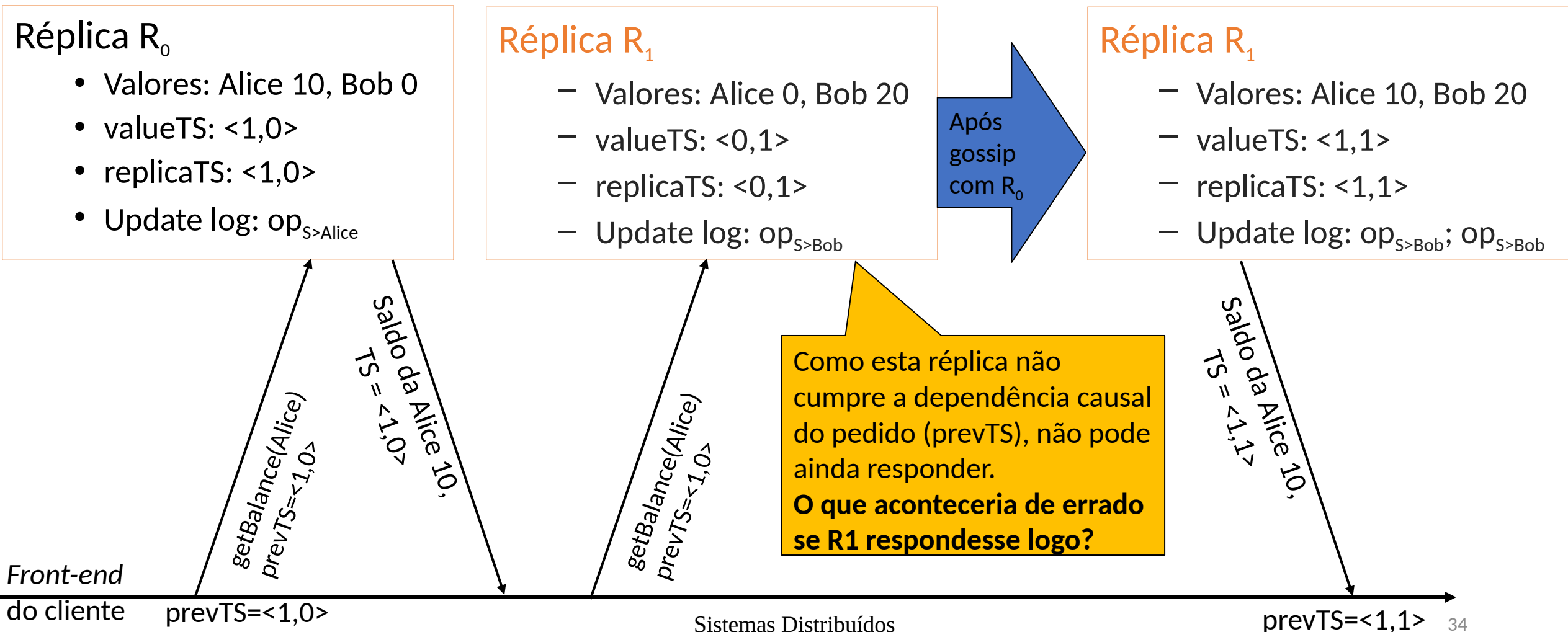
Front-end  
do cliente

prevTS= $\langle 0, 0 \rangle$

Sistemas Distribuídos

prevTS= $\langle 1, 0 \rangle$

# Exemplo: pedidos de leitura a diferentes réplicas



# Lazy Replication

- O artigo também descreve algoritmos para executar operações que necessitam de modelos de coerência mais fortes
  - **Forced** operations
  - **Immediate** operations
- Nesta cadeira não descrevemos estes algoritmos



# Bayou



## **Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System**

Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers,  
Mike J. Spreitzer and Carl H. Hauser

Computer Science Laboratory  
Xerox Palo Alto Research Center  
Palo Alto, California 94304 U.S.A.

# Bayou

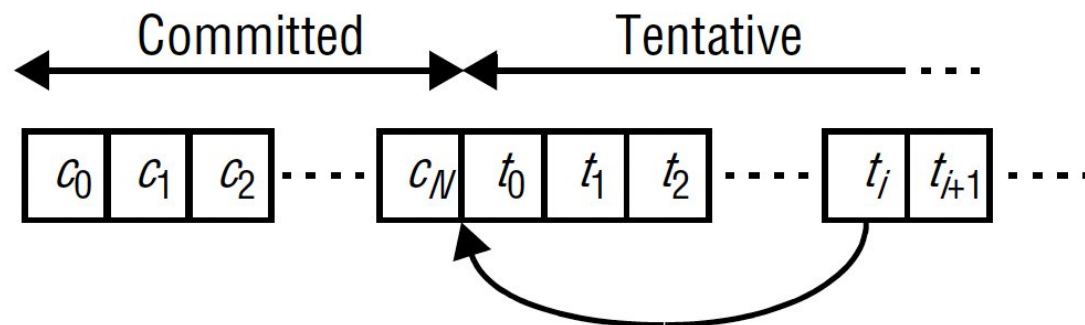
- No sistema de "Lazy Replication" assume que as operações concorrentes são comutativas:
  - Por exemplo, "pintar círculo" e "alterar espessura da linha" são operações que podem ser aplicadas por qualquer ordem
- Se as operações não forem comutativas, devem ser usadas as primitivas mais fortes (forced/immediate).

# Método de detecção e resolução de conflitos

- Na arquitetura Gossip usam-se **timestamps vetoriais** para garantir **causalidade** entre determinadas operações.
- No Bayou os conflitos são resolvidos de acordo com uma política específica **definida pela aplicação** (solução “domain-specific”)
  - **todas** as escritas são aceites na réplica onde chegam
  - Quando as escritas são propagadas para as outras réplicas usa-se um qualquer método de resolução dos conflitos definido pela aplicação
    - Por exemplo, “escritas feitas pelo *professor* são preferidas às feitas pelo *aluno*”
  - A consequência é uma **transformação operacional**
    - Certas escritas feitas anteriormente são efetivamente desfeitas

# Updates: tentativo e commit

- Quando se faz um *update* ele é marcado como **tentativo**
  - Nesta fase podem ser ainda alterados
- Quando o update é **committed**, a sua ordem já não pode mudar
  - Quem decide a ordem é tipicamente uma réplica especial, o **primário**
  - Quando um *update* é committed as outras réplicas podem por isso ter de o reordenar



# Exemplo de operação em Bayou

```
Bayou_Write(  
  update = {insert, Meetings, 12/18/95, 1:30pm, 60min, "Budget Meeting"},  
  dependency_check = {  
    query = "SELECT key FROM Meetings WHERE day = 12/18/95  
            AND start < 2:30pm AND end > 1:30pm",  
    expected_result = EMPTY},  
  mergeproc = {  
    alternates = {{12/18/95, 3:00pm}, {12/19/95, 9:30am}};  
    newupdate = {};  
    FOREACH a IN alternates {  
      # check if there would be a conflict  
      IF (NOT EMPTY (  
        SELECT key FROM Meetings WHERE day = a.date  
        AND start < a.time + 60min AND end > a.time))  
        CONTINUE;  
      # no conflict, can schedule meeting at that time  
      newupdate = {insert, Meetings, a.date, a.time, 60min, "Budget Meeting"};  
      BREAK;  
    }  
    IF (newupdate = {}) # no alternate is acceptable  
      newupdate = {insert, ErrorLog, 12/18/95, 1:30pm, 60min, "Budget Meeting"};  
    RETURN newupdate;  
  )  
)
```

# Bibliografia recomendada

- [Coulouris et al]
  - Secções 18.4.1, 18.4.2 e 14.4
- W. Vogles, “[Eventually Consistent](#)”, Communications of the ACM, 2009

