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 simultanemente:
  - Coerência (consistency)
  - Disponibilidade (availavility)
  - E tolerar particções 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 propagaçao 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 balancemaneto 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.

Aplicar actualizações

- Considerem-se três réplicas, R1, R2, e R3 e uma aplicação para preparar slides de forma cooperativa.
  - Na réplica R1 um cliente executa “criar círculo”, e esta actualização é propagada para R2
  - Na réplica R2 um cliente executa a operação “pintar círculo” que é propagada para R3
  - A réplica R3 recebe “pintar círculo” antes de ter recebido a operação “criar círculo”. Isto pode gerar um erro.

Aplicar actualizações

- Considerem-se três réplicas, R1, R2, e R3 e uma aplicação para preparar slides de forma cooperativa.
  - Na réplica R1 um cliente executa “label=distribíudos” e propaga esta operação para R2
  - Na replica R2, um cliente corrige a gralha, executando “label=distribuídos”
  - A réplica R3 recebe primeiro a operação de R2
  - A réplica R3 recebe posteriormente a operação de R1
  - Se a réplica R3 aplicar as actualizaçoes por esta ordem, a correcção feita na réplica R2 vai ser apagada

Aplicar actualizações

- A grande maioria dos sistemas de propagação epidémica aplica as actualizações de forma a respeitar a relação “aconteceu-antes”.
  - Ou seja, todos as actualizações são entregues por ordem causal.

Suportar clientes móveis

- O cliente muda a sua palavra chave na réplica R1
- O cliente muda de réplica e contacta R2 antes da alteração ser propagada
- O cliente não consegue usar a sua nova palavra passe!

Suportar clientes móveis

- Vários sistemas garantem que os clientes observam sempre um estado que é coerente com a relação “aconteceu-antes”

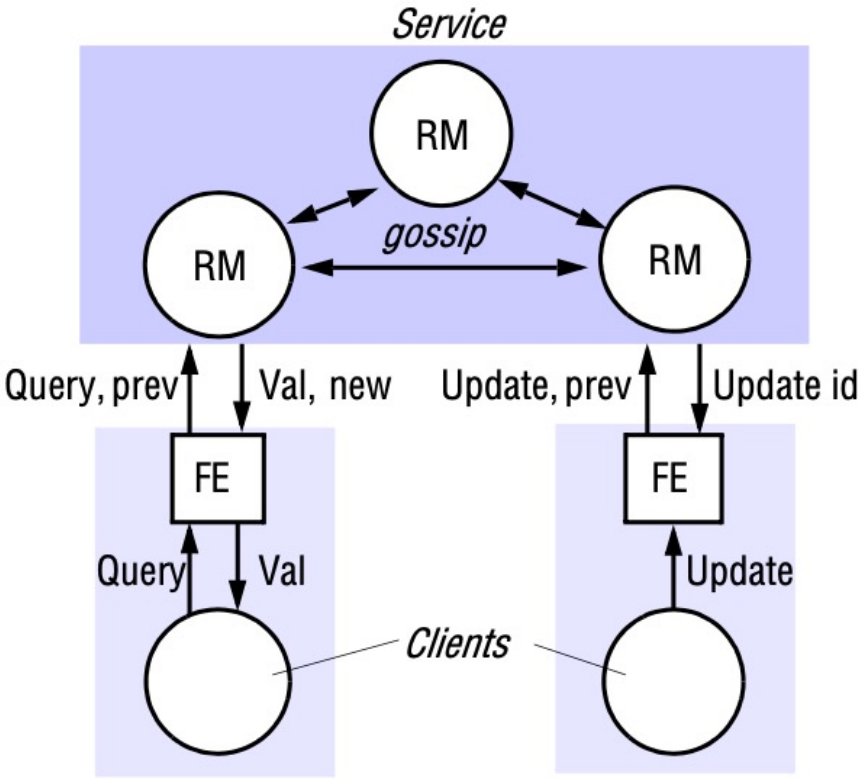
Suportar clientes móveis

- Alguns sistemas sugerem a utilização de garantias ainda mais fracas
  - Read-your writes
  - Monotonic reads
  - Writes Follow Reads
  - Monotonic writes

Como oferecer estas garantias?

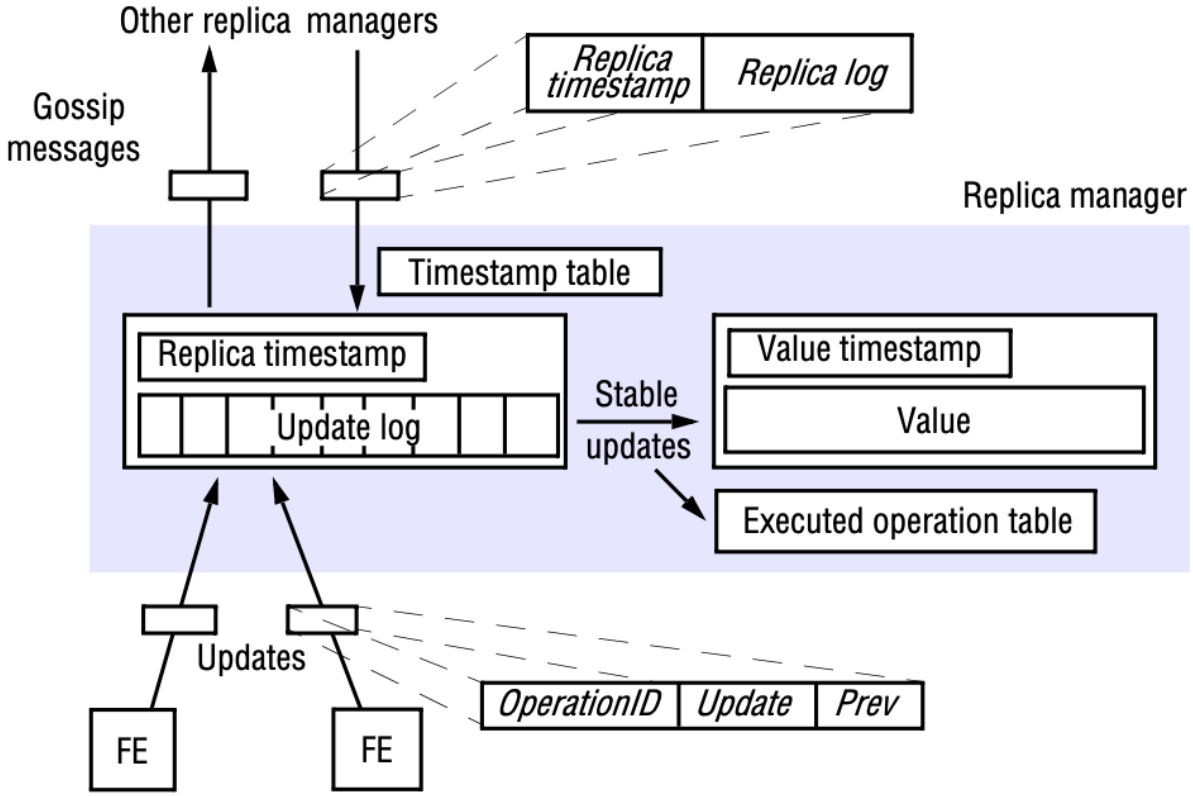
- Usando relógios lógicos
  - Possível
  - Mas não eficiente

Lazy Replication



Retirado da página 783 do livro da cadeira!

Lazy Replication



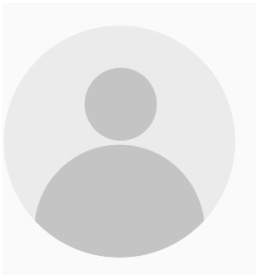
Retirado da página 787 do livro da cadeira!

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.

Gossip architecture

- Proposta académica de 1992, que inspirou muitos sistemas fracamente coerentes de hoje
- Objetivo:
  - Oferecer sempre **acesso rápido** aos clientes
  - Mesmo em situações de **partições**
  - **Sacrificando a coerência**
- Nome ***gossip*** vem do facto das réplicas propagarem as novas modificações entre si periodicamente, em *background*
  - Como se fosse o espalhar de um boato

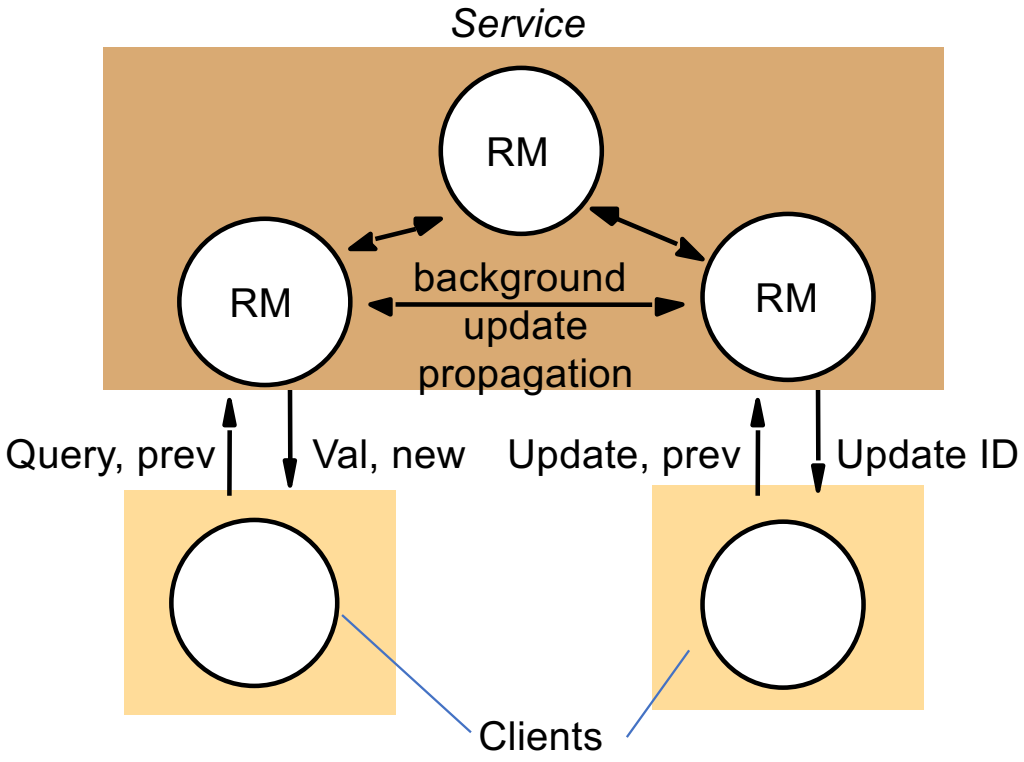
Coerência fraca mas com duas garantias

- Assegura *monotonic reads*
  - Mesmo que o **cliente** aceda a diferentes réplicas
- Estado de uma réplica respeita sempre a **ordem causal** entre modificações
  - Se uma modificação *m2* depende de outra *m1*, réplica nunca executa *m2* sem ter antes executado *m1*

Algoritmo: interação *cliente* – réplica

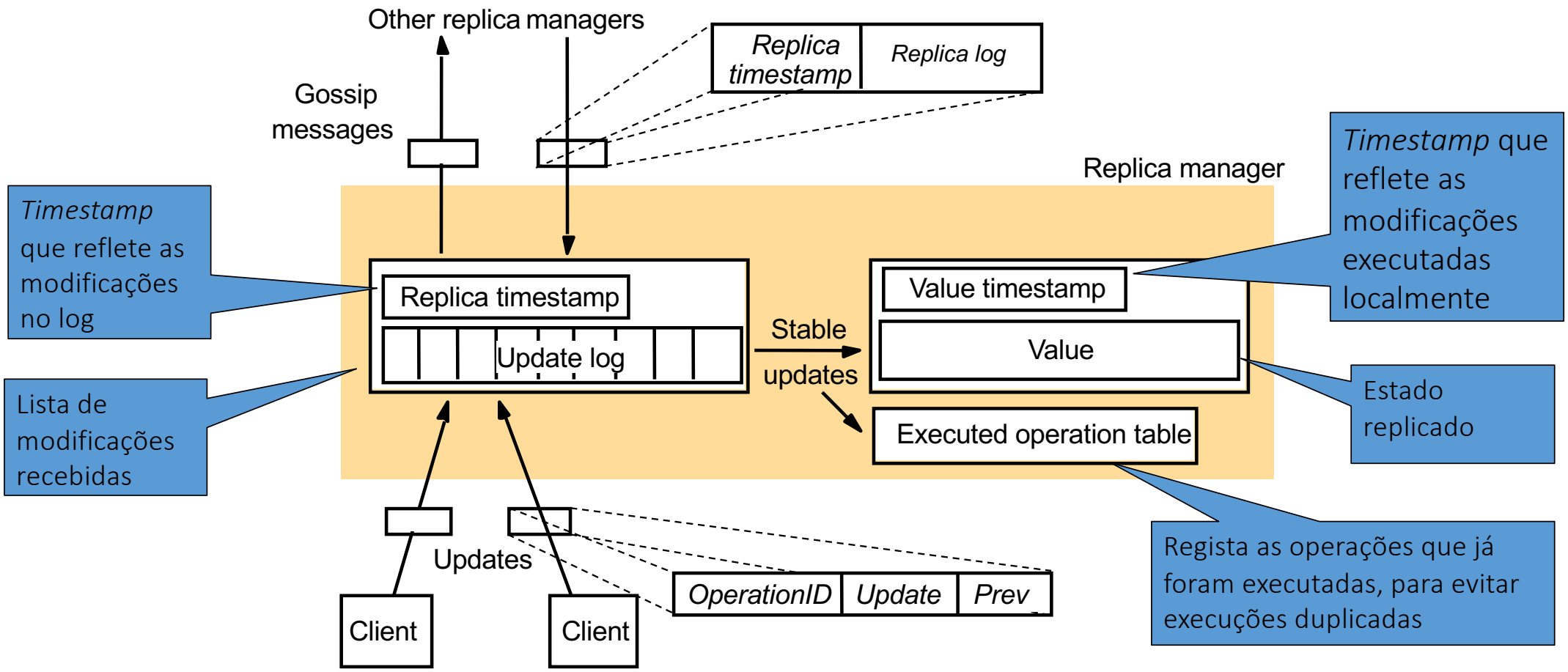
- Cada *cliente* mantém um ***timestamp* vetorial** chamado *prev*
  - Vetor de inteiros, um inteiro por cada réplica
  - Reflete a última versão acedida pelo *cliente*
- Em cada pedido a uma réplica, ***cliente* envia (pedido, *prev*)**
- Réplica responde com **(resposta, *new*)**
  - *new* é o timestamp 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



Retirado da página 783 do livro da cadeira!

Estado mantido por uma réplica



Retirado da página 787 do livro da cadeira!

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 ≤ 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 *pedido.prev ≤ value timestamp*** se verificar para executar o pedido localmente
  - Quando executar o pedido, atualiza o *value timestamp*
    - Para cada entrada i, atualiza valueTS[i] se *replicaTS*[i] > *valueTS*[i]

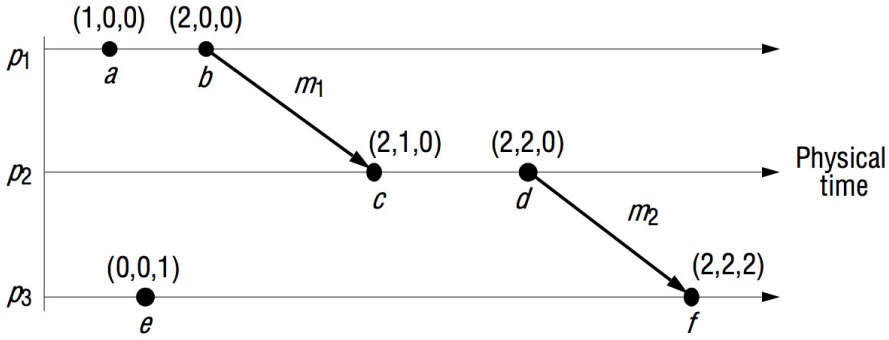
Garante que a execução respeita a ordem causal

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 ≤ value timestamp***, executa a modificação

Happens-before versus protocolo *gossip*

- Quando estudámos a relação *happens-before* vimos que 3 tipos de eventos atualizavam as *timestamps* vetoriais*:*
  - Eventos locais genéricos
  - Envio de uma mensagem
  - Receção de uma mensagem
- No protocolo *gossip* só um tipo de eventos atualiza as *timestamps* vetoriais*:*
  - *Updates*



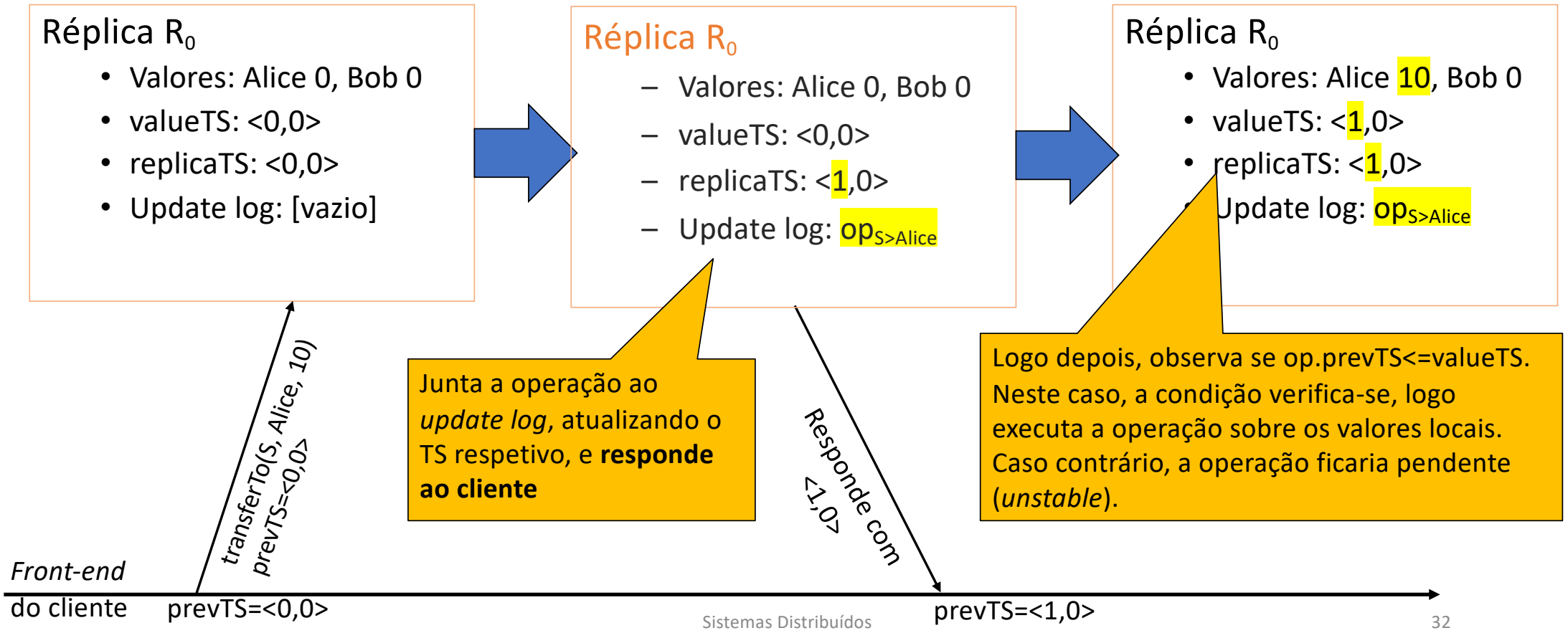
Exemplo da Gossip Architecture

Enquadramento

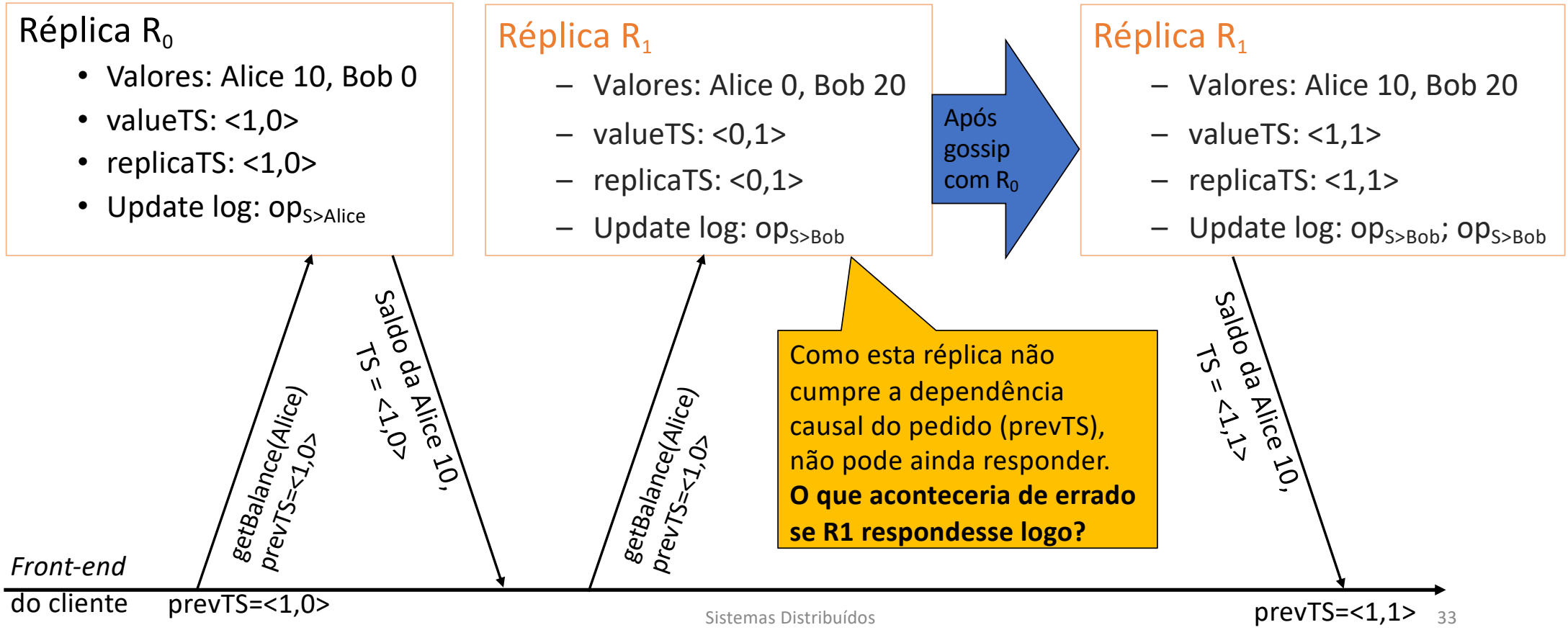
- Sistema de 2 réplicas, R_0 e R_1, em que ambas mantêm saldos de contas bancárias
- As contas da Alice e Bob começaram com saldo nulo e depois receberam transferências de uma conta S (*)
- Cada operação foi aceite por uma réplica diferente:
  - $op_{S>Alice}$: transferTo(S, Alice, 10)
    Aceite por R_0 com TS=<0,1> e prevTS=<0,0>
  - $op_{S>Bob}$: transferTo(S, Bob, 20)
    Aceite por R_1 com TS=<1,0> e prevTS=<0,0>

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

Exemplo: pedido de escrita



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

Garantias de sessão

Session Guarantees for Weakly Consistent Replicated Data

Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer,
and Brent B. Welch

Computer Science Laboratory
Xerox Palo Alto Research Center
Palo Alto, California 94304

Abstract

Four per-session guarantees are proposed to aid users and applications of weakly consistent replicated data: Read Your Writes, Monotonic Reads, Writes Follow Reads, and Monotonic Writes. The intent is to present individual applications with a view of the database that is consistent with their own actions, even if they read and write from various, potentially inconsistent servers. The guarantees

may want to read and update data copied onto their portable computers even if they did not have the foresight to lock it before either a voluntary or an involuntary disconnection occurred. Also, the presence of slow or expensive communications links in the system can make maintaining closely synchronized copies of data difficult or uneconomical.

Unfortunately, the lack of guarantees concerning the ordering of read and write operations in weakly consistent

Garantias de sessão

Guarantee	session state updated on	session state checked on
Read Your Writes	Write	Read
Monotonic Reads	Read	Read
Writes Follow Reads	Read	Write
Monotonic Writes	Write	Write

Retirado do artigo

Garantias de sessão

```
Write(W,S) = {
    if WFR then
        check S.vector dominates read-vector
    if MW then
        check S.vector dominates write-vector
    wid := write W to S
    write-vector[S] := wid.clock
    }
```

Retirado do artigo

Garantias de sessão

```
Read(R,S) = {
    if MR then
        check S.vector dominates read-vector
    if RYW then
        check S.vector dominates write-vector
    [result, relevant-write-vector] := read R from S
    read-vector := MAX(read-vector,
        relevant-write-vector)
    return result
    }
```

Retirado do artigo

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

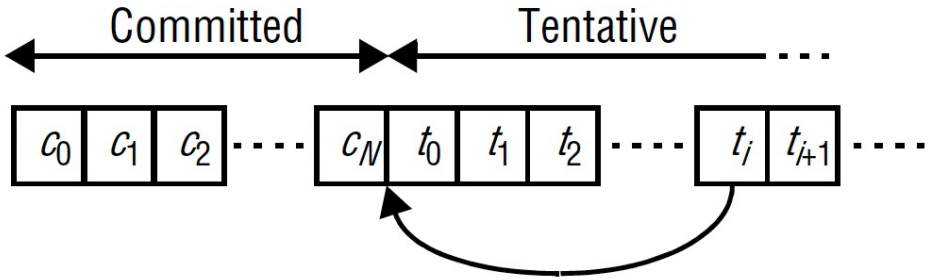
- No sistema de ”Lazy Replication” assume que as operações concorrentes são comutativas:
  - Por exemplo, “pintar círculo” e “alterar espessura da linha” são operações que podem ser aplicadas por qualquer ordem
- Se as operações não forem comutativas, devem ser usadas as primitivas mais fortes (forced/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

Bayou

- O Bayou introduz a ideia de reconciliar de forma automática réplicas divergentes:
  - Ordenar as operações concorrentes por uma ordem total
  - Cancelar as operações que foram executas por uma ordem diferente
  - Re-aplicar estas operações pela ordem total, aplicando uma função de reconciliação que é específica da aplicação
  - As operações podem ser codificadas de forma a facilitar a reconciliação automática

Updates: tentativo e *commit*

- Quando se faz um *update* ele é marcado como **tentativo**
  - Nesta fase podem ser ainda alterados
- Quando o update é **commited**, a sua ordem já não pode mudar
  - Quem decide a ordem é tipicamente uma réplica especial, o **primário**
  - Quando um *update é* commited 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;}
)
```

Retirado do artigo

Bibliografia recomendada

- [Coulouris et al]
  - Secções 18.4.1, 18.4.2 e 14.4

- W. Vogles, "Eventually Consistent", Communications of the ACM, 2009