UNIVERSIDADE DE ÉVORA

Sistemas Distribuídos

Replicação e *Sharding*

Introdução

Motivação

- num SD espera-se que os serviços tem **alta disponibilidade**, o **melhor desempenho possível** e que o efeito das **falhas seja mínimo**

Replicação

- fundamental para SDs, na medida em que contribui para desempenho, alta disponibilidade e tolerância a falhas

Exemplos de Replicação

- cache em servidores Web
- cópias de dados no BackEnd de um serviço
- redundância de módulos (software+hardware)

Replicação: as possíveis Vantagens

ganho de desempenho

- cache de dados mais perto do cliente (browser, proxy) permite melhores tempos de acesso, diminui a latência
- divisão e distribuição de grandes coleções de dados
- replicação de servidores para distribuição de carga/processamento permite melhores tempos de resposta
  - ainda evita sobrecarga em servidores e rede

Replicação: as possíveis Vantagens

aumento da disponibilidade

- disponibilidade do serviço: quantidade de tempo em que o serviço está acessível/disponível/funcional e com tempos de resposta razoáveis
  - expressa numa taxa: 0 a 100%. Deseja-se que seja próxima de 100%.

- Factores relevantes para a alta disponibilidade
  - falhas no(s) servidor(es)
  - falhas nos dados necessários para o serviço
  - problemas de rede, problemas de comunicação

- Com Replicação dos dados em vários servidores, em caso de falha de um servidor a aplicação cliente pode ainda ser atendida por um servidor alternativo.
- O serviço continua disponível!

Replicação: as possíveis Vantagens

aumento da disponibilidade

1- Se um serviço tem réplicas em **N** servidores independentes e com probabilidade **p** de falharem ou ficarem incontactáveis,

então, a disponibilidade do serviço é

- 1 – probabilidade(todos os servidores falharem ou ficarem inacessíveis)
- $1 - p^N$

2- se um servidor tem uma

- probabilidade de falhar de 5% durante um determinado período, e
- existem dois desses servidores com réplicas de um serviço
- então a disponibilidade será $1 - 0.05^2$, ou 0.9975, ou ainda 99.75%

Diferença entre replicação de servidores e Cache

- cache não tem obrigatoriamente conjuntos de objetos completos (ficheiros inteiros). Cache é uma forma de replicação parcial.
- cache está mais próxima do destino (exemplo: aplicação cliente, como um browser...)

Replicação: as possíveis Vantagens

tolerância a falhas

- disponibilidade não significa correção ou consistência

Um serviço tolerante a faltas garante sempre um comportamento correto, mesmo após um certo nº e de falhas de determinado tipo

Problemas

- crash de N servidores
  - resolver com replicação: **N+1** servidores
- N falhas bizantinas

  (existe resposta, mas com um valor errado. difíceis de detectar)
  - resolver com replicação: **2N+1** servidores
  - a resposta válida é a da maioria dos servidores, pelo menos N+1

Replicação e Requisitos

Requisitos na replicação dos dados:

1) transparência
   - os clientes devem ser poupados aos detalhes da replicação.
   - não devem ter de funcionar em função do nº de copias “físicas” dos dados.
   - do ponto de vista do cliente, tudo deve funcionar como se existisse uma única réplica.

2) consistência
   - existe consistência quando as operações efetuadas sobre um conjunto de objetos replicados obtém resultados que obedecem a critérios de correção

Modelo Geral de Replicação

assume-se sistema assíncrono

Replica Manager (**RM**)

- módulos ou servidores que contém as réplicas em cada computador
- atuam diretamente sobre as réplicas
- comunicam entre si, arquitetura cliente-servidor
- cada RM tem uma réplica de tudo ou parte dos dados
- PRESSUPOSTOS:
  - as operações que o RM efetua podem ser desfeitas (permitir rollback)
  - o estado da réplica é função determinística do estado inicial e sequência de operações efetuadas

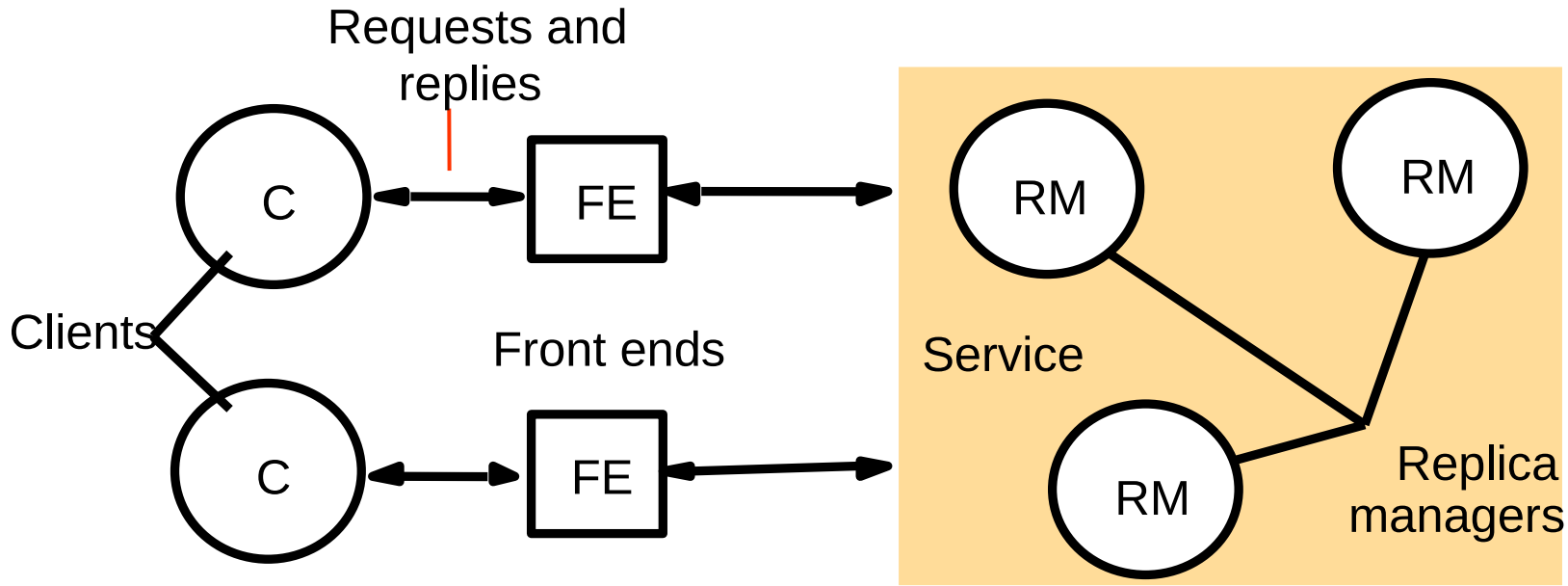
Operações dos Clientes

- *read-only request*: apenas consultas, não há escrita
- *update requests*: alteram um objeto

Arquitetura para o Modelo Geral de Replicação

FrontEnd: módulo mais exterior do serviço

- intermediário entre os Clientes do serviço e os RM
- torna a replicação transparente para o cliente



Modelo Geral: **Fases** para atendimento do pedido do cliente

1. FrontEnd envia o pedido para um ou mais RM
   a) FE comunica apenas com um RM, que depois se liga aos restantes
   b) FE envia multicast a todos os RM
2. Coordenação: RMs coordenam-se para executarem a operação de forma consistente (decidem se o pedido é aplicado e a ordem do pedido relativamente a outros)
3. Execução: execução do pedido nos RM (pelo menos tentada)
4. Acordo (*Agreement*): RMs entendem-se relativamente ao resultado e em função disso fazem rollback ou commit
5. Resposta: um ou mais RM respondem ao FE
   - em certos casos, o FE coleciona as respostas dos RM e seleciona a resposta a passar ao cliente
     - o resultado com pelo menos N+1 respostas em 2N+1 servidores
     - permite ultrapassar falhas bizantinas

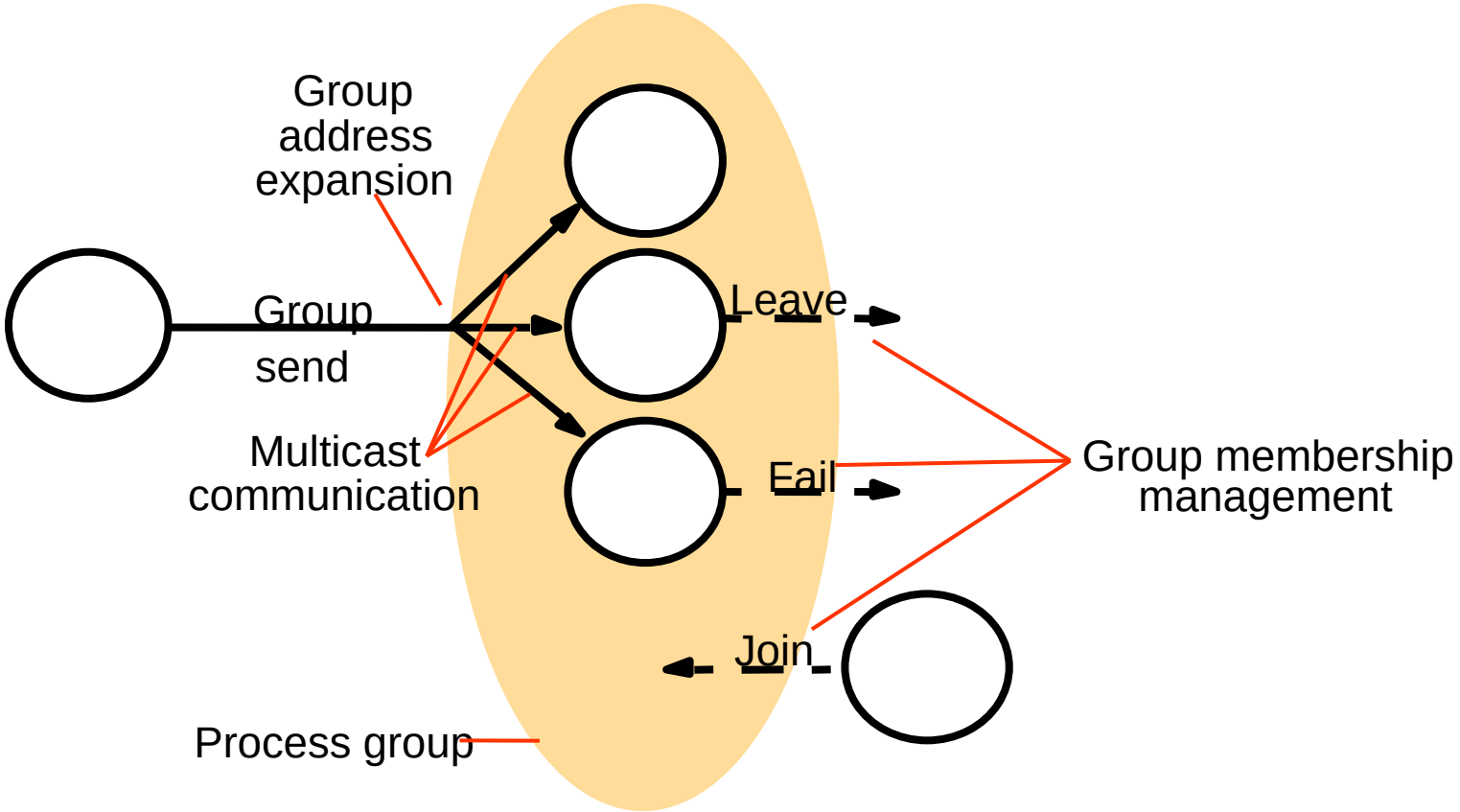
Modelo Geral: Fases para atendimento do pedido do cliente

Coordenação: Ordenação no processamento dos pedidos:

1. FIFO: se o FE processa r e depois r', então qualquer RM consistente tratá também r antes de r'
2. Causal: se o pedido (a um RM) para r aconteceu antes do pedido para r', então se o RM é consistente tratará r antes de r'
3. Total: Se um RM consistente trata r antes de r', então todos os RM consistentes tratarão r antes de r'

Comunicação em Grupo

a troca de mensagens com os RM é mais eficaz através da comunicação para um grupo (multicast)



Réplicas e consistência

Critérios de Consistência:

- consistência sequencial
  - não depende da precisão dos relógios para a verificação dos timestamps de cada operação em máquinas diferentes. Não usa referências temporais mas antes uma ordem (sequência)...
  - um objeto replicado obedece à consistência sequencial se, para cada execução, existe uma sequência de operações desencadeadas por todos os clientes que satisfaz:
    1. a sequência permite alcançar uma única cópia correta dos objetos
    2. a ordem das operações na sequência está de acordo com a ordem no código do programa do cliente que as executa/solicita

Réplicas e consistência

Critérios de Consistência:

- consistência linear
  - um objeto replicado é linearmente consistente se, para qualquer execução, existe algum encadeamento/sequência das operações desencadeadas por todos os clientes que satisfaz:
    1. a sequência permite alcançar uma única cópia correta dos objetos
    2. a ordem das operações na sequência está de acordo com o tempo real a que efetivamente ocorreram
  - mais rígida

Replicação e Tolerância a Falhas

Outra leitura sobre consistência em réplicas:

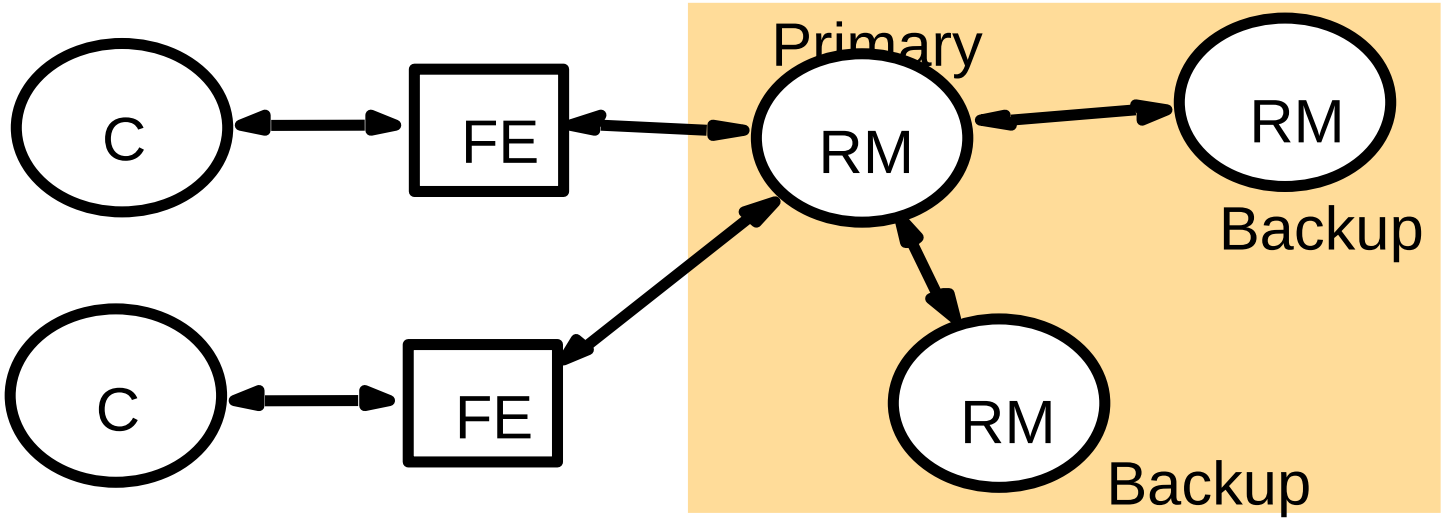
- http://book.mixu.net/distsys/abstractions.html#strong-consistency-models

Modelos de Replicação para Tolerância a Falhas

- replicação passiva
- replicação ativa

Tolerância a Falhas: Replicação Passiva

- existe um único RM primário e vários secundários ou de backup
- os FE comunicam apenas com o RM primário
- o RM primário executa a operação e envia cópias dos dados atualizados aos RM Backup



Tolerância a Falhas: Replicação Passiva

Sequência de eventos para atender um pedido

1. Request: FE passa o pedido, com um identificador único, ao RM primário
2. Coordenação: o primário trata cada pedido pela ordem em que o recebe e de forma atómica. Verifica o identificador e se já o tiver executado reenvia a resposta.
3. Execução: primário executa a operação e guarda a resposta
4. Acordo: em caso de *update,* o primário envia <estado atualizado, a resposta, identificador do pedido> a todos os backups, que lhe respondem com ackowledgement
5. Resposta: primário responde ao FE, que por sua vez responde ao cliente

Tolerância a Falhas: Replicação Passiva

Existe consistência linear

- RM primário lineariza os pedidos

Em caso de falha no RM primário

- um e um só dos backups é promovido a RM primário
- se a nova configuração continua do ponto em que o sistema estava mantém-se a consistência linear
- o RM secundário eleito regista-se como primário no serviço de nomes

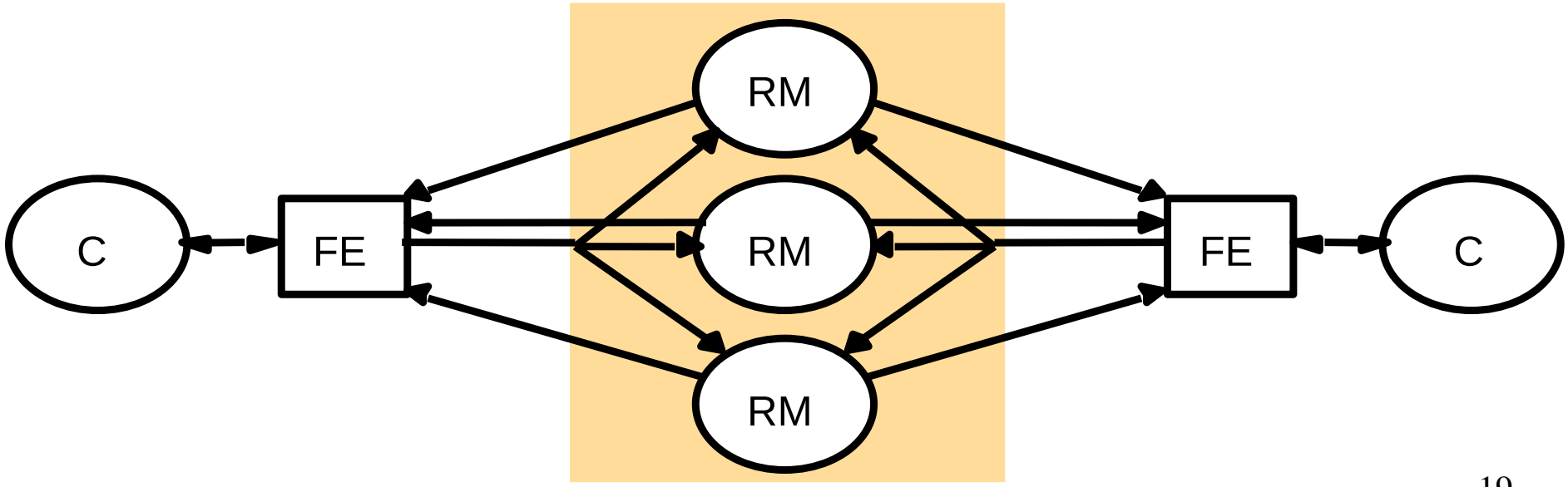
replicação passiva e tolerância a falhas

- permite sobreviver a N crashes de servidores com N+1 RM
- não tolera falhas bizantinas

desvantagem: *overheads* (alguma lentidão) na sincronização entre RM primário e RMs de backup

Tolerância a Falhas: Replicação Ativa

- RM têm igual função e estão organizados como um grupo
- FE envia o pedido por multicast aos elementos do grupo
- cada RM processa o pedido e responde ao FE
- a falha de um RM não tem impacto sobre o serviço



Tolerância a Falhas: Replicação Ativa

Sequência de eventos para atender um pedido

1. Request: FE atribui identificador único ao pedido e faz multicast para o grupo de RM. FE não envia o próximo pedido enquanto não receber a resposta do atual.
2. Coordenação: o sistema de comunicação em grupo difunde o pedido a todos os RM que são membros válidos do grupo, e garante a ordenação dos pedidos
3. Execução: cada RM executa o pedido.
4. Acordo: nenhum procedimento é necessário
5. Resposta: os RM enviam a resposta, juntamente com o identificador único ao FE. O nº de respostas que o FE processa depende dos objetivos:
    1. tolerar apenas falhas do tipo crash: devolver ao cliente a primeira resposta e descartar o resto
    2. normalmente o objetivo é tolerar também falhas bizantinas. Para tal é preciso recolher várias respostas e comparar os valores... procurar o da maioria e devolver (mesmo que haja algum por receber)

Tolerância a Falhas: Replicação Ativa

Existe consistência sequencial

- cada FE trata os pedidos de forma sequencial, como um FIFO

Não existe consistência linear

- a ordem pela qual os RM processam os pedidos pode não ser igual à ordem pela qual os clientes pediram a operação (vários FE)

Tolera falhas por Crash e ainda Falhas Bizantinas

- tolera até N falhas bizantinas com, pelo menos, 2N+1 RM
- cada FE espera até receber N+1 respostas com o mesmo valor e responde isso ao cliente descartando as restantes.

Serviços de Alta Disponibilidade

o passo “Acordo/Agreement” visto antes, onde todos os RM recebem updates antes que o controlo passe para o cliente pode não ser desejável para um sistema onde se deseja Alta Disponibilidade por penalizar o tempo de resposta

objetivo: proporcionar aos clientes o acesso ao serviço, com melhor tempo de resposta (ainda que a consistência entre as réplicas nem sempre seja a melhor)

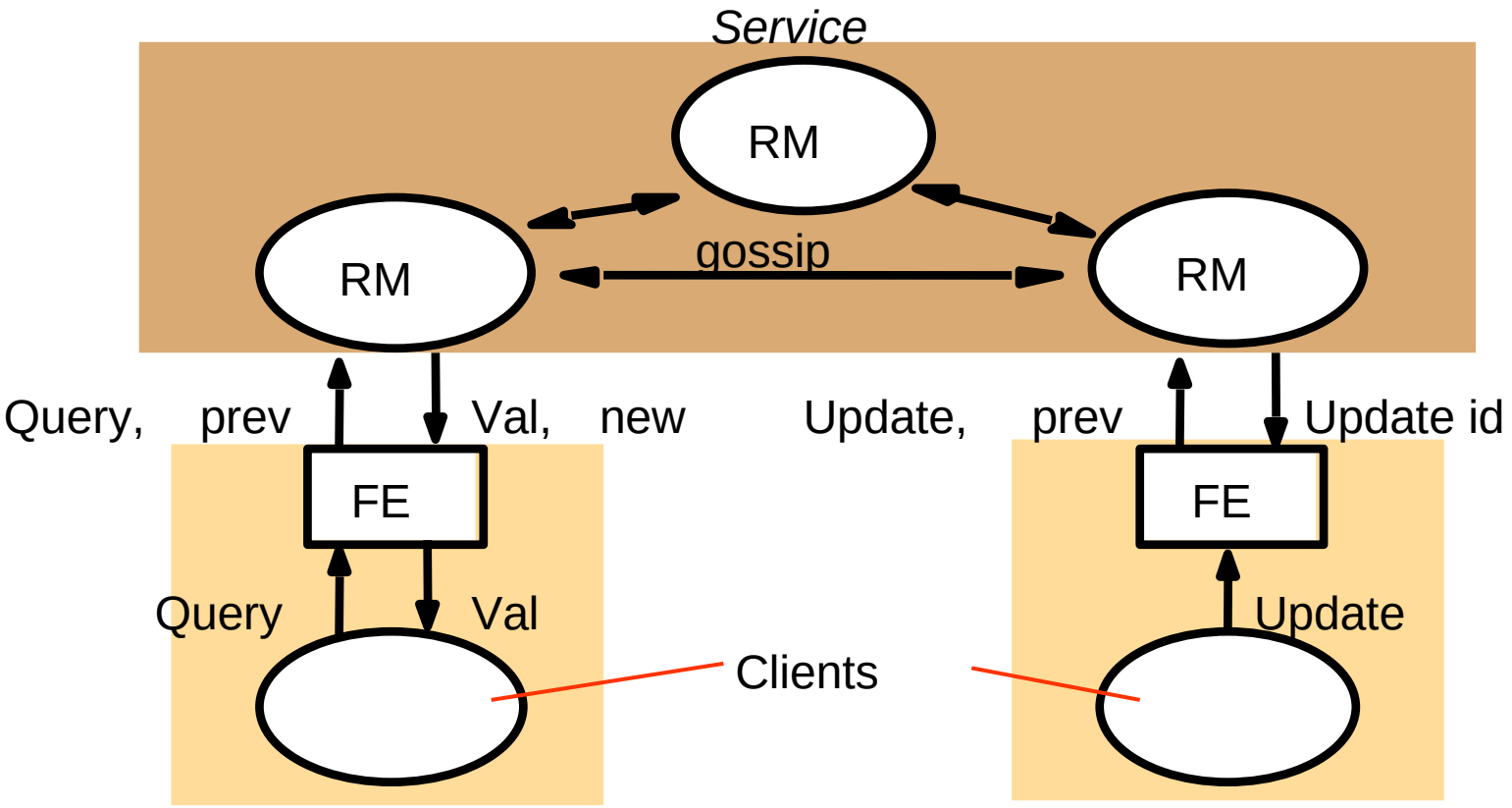
- minimizar tempos de resposta

Exemplo de serviços de Alta Disponibilidade:

- arquitetura `gossip`
  - *Ladin et al.* (1992)

Arquitetura Gossip

os RM podem ficar temporariamente desligados, sofrem updates individualmente. Quando voltam a ligar-se trocam mensagens com as atualizações (***gossip*** *messages*)



Arquitetura Gossip

funcionamento para um pedido:

1. Request
    1. Query: FE contacta um RM, cliente fica bloqueado à espera
    2. Update: possibilidades (updates aqui não podem ter reads)
        - a) FE devolve o controlo ao cliente e propaga o pedido em background
        - b) o cliente espera que o update esteja reflectido em N+1 de 2N+1 RMs (+ fiável)
        - Nota: tanto a) como b) são mais rápidas que o caso usual (esperar por todos)
2. Resposta a pedido Update: RM responde assim que tiver recebido o update
3. Coordenação: o RM que recebe o pedido não o processa até obedecer a uma ordenação. Pode ter de aplicar primeiro *updates* vindos de outros RM.
4. Execução: o RM executa o pedido
5. Query Response: resposta a read ocorre aqui
6. Acordo: RM trocam mensagens *gossip* para partilhar as atualizações, o que por vezes ocorre de uma forma *lazy* (apenas quando se deteta a falta da atualização no RM que atente um cliente)

Arquitetura Gossip

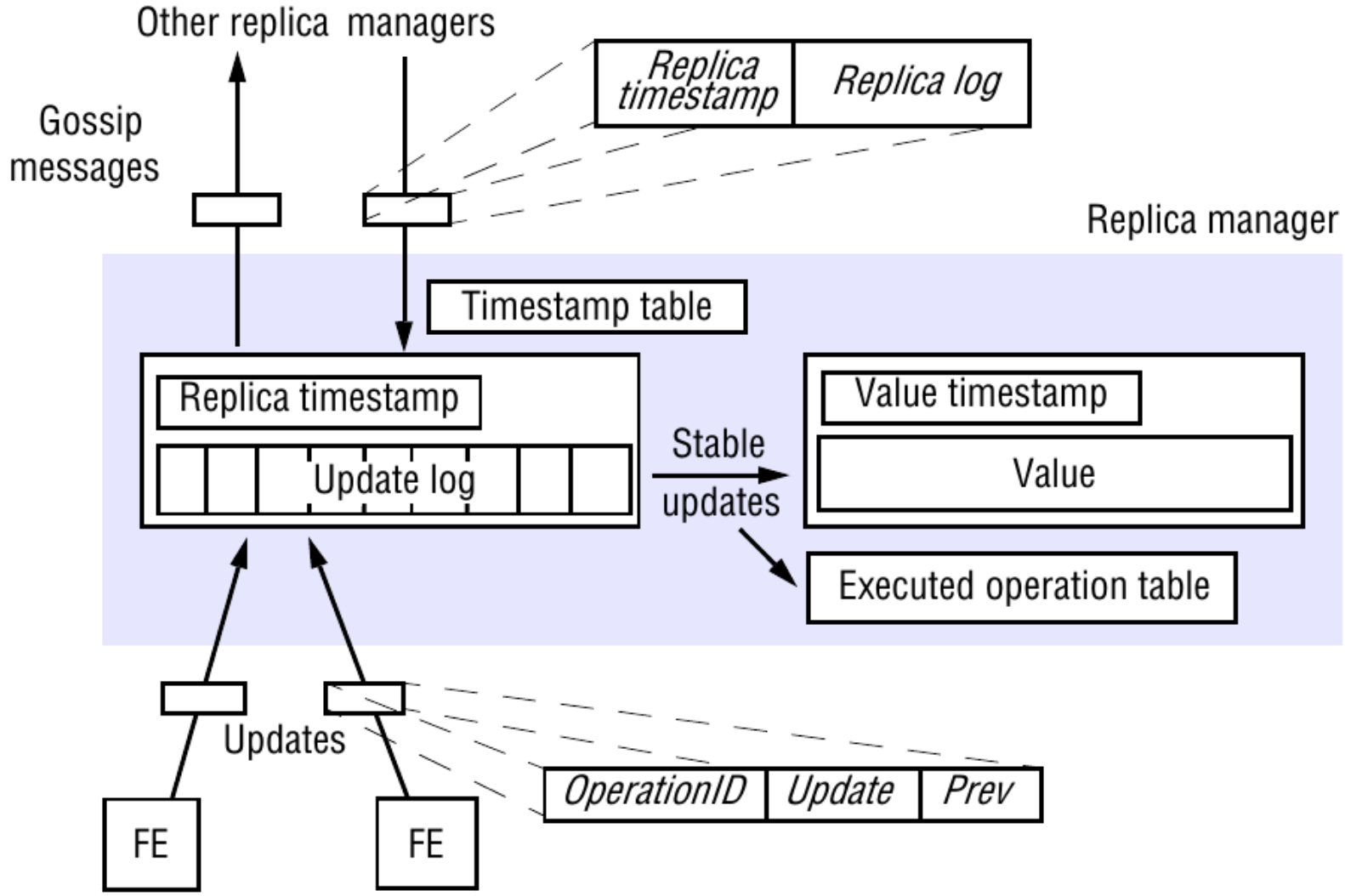
os RM podem ficar algum tempo sem comunicar

Garantias:

- cada cliente observa respostas consistentes ao longo do tempo
  - a cada pedido, os dados refletem, pelo menos, as atualizações que o cliente já observou até então (mesmo que tenham sido noutro RM)
- “relaxed consistency” nas réplicas
  - todos os RM eventualmente recebem todas as atualizações, que aplicam por ordem. Isto permite às réplicas serem suficientemente similares para satisfazer as necessidades do programa

permite alta disponibilidade sacrificando o nível de consistência entre os RM

Arquitetura Gossip



Arquitetura Gossip

- Cada FE tem um *version timestamp* com a versão dos últimos dados observados de cada RM (e passados ao cliente) – prev
  - Esse timestamp vector é enviado para o RM, juntamente com o pedido
    - O RM compara a sua versão estável dos dados necessários com a versão necessária para o cliente/RM – se necessário ativa *gossip messages* para se atualizar

- *Cada RM inclui*
  - *Value* – dados
  - *Value timestamp* – tem uma entrada por cada RM; é atualizado com a info proveniente do FE ou de gossip messages
  - *Update Log* – registo de operações sobre os dados, mantidas numa lista por 2 motivos:
    - Algumas das operações não podem aplicar-se de imediato, só depois do RM ficar em estado atualizado para a operação em causa
    - essa informação pode ser necessária para efeitos de *gossip messages*
  - *Replica timestamp* – inclui versão dos updates efetivamente aplicados no RM
  - *Operation Table* – histórico para deteção de duplicados (pedidos via FE que podem chegar em duplicado via *gossip messages*)

Sharding (fragmentar coleções de dados)

- no contexto das BDs distribuídas, consiste na distribuição dos dados por várias Bds
  - os dados são particionados e cada bloco (shard, fragmento) vai para uma BD/servidor dedicado
    - é a chamada partição horizontal
    - permite distribuir dados... e também a carga sobre os servidores!
- um (único) conjunto de dados é representado num conjunto de BDs

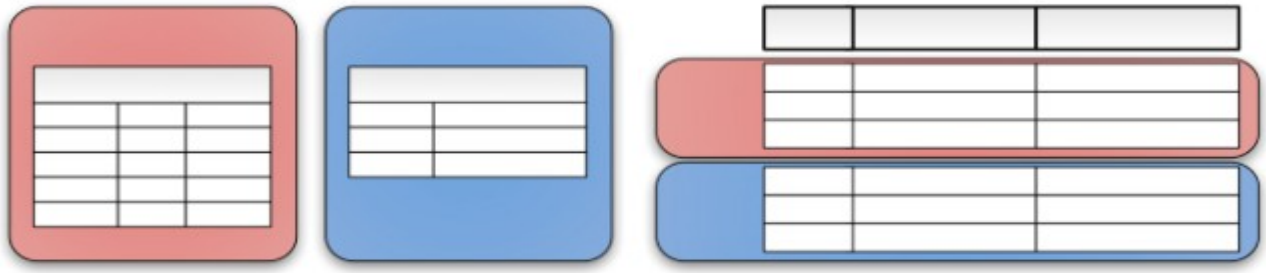
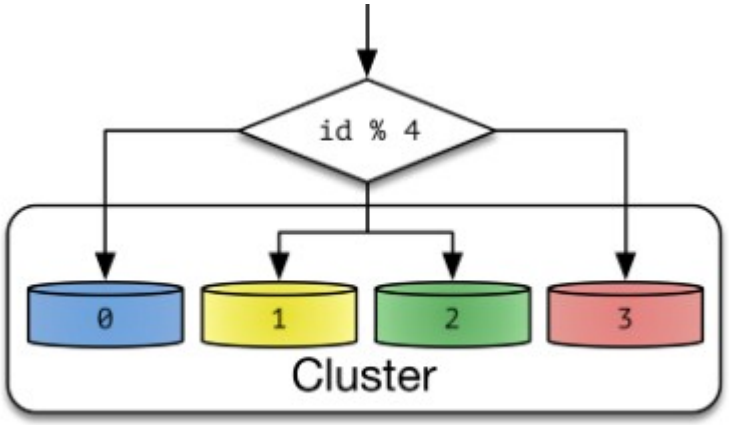


imagem: Jeeyoung Kim (2014). *Read, Write, Execute. https://medium.com/*

Sharding (fragmentar coleções de dados)

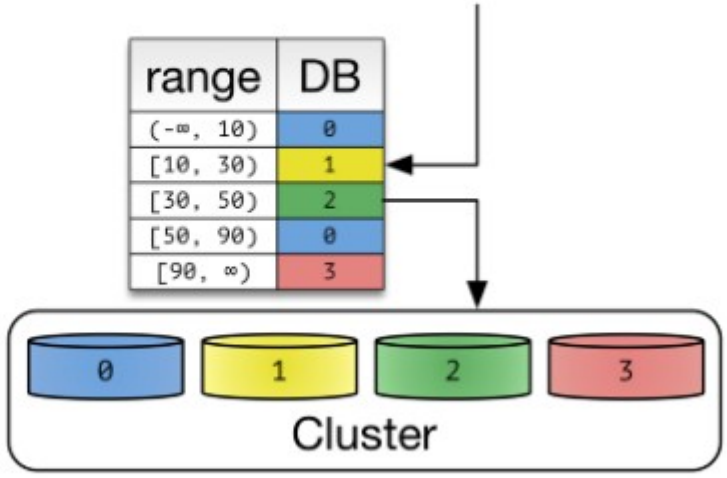
- cada shard é o bloco de dados com a mesma *Partition Key*
  - esta chave é calculada a partir do identificador dos dados e serve para organizar a distribuição dos dados

Case 1 — Algorithmic Sharding



An algorithmically sharded database, with a simple sharding function

Case 2— Dynamic Sharding



A dynamic sharding scheme using range based partitioning.

imagem: Jeeyoung Kim (2014). *Read, Write, Execute. https://medium.com/*

*Sharding (*fragmentar coleções de dados*)*

- A ideia geral é evitar que um só servidor seja alvo de muita carga, enquanto que outros servidores não têm carga
  - isto depende do tipo de dados e dos padrões de acesso aos mesmos
- *Algorithmic sharding*
  - adequado para distribuições homogéneas de dados, com acessos igualmente distribuídos
- *Dynamic sharding*
  - adequado para distribuição de coleções não uniformes de dados

Ideias e propósitos

- *Partitioning (sharding)*
  - escalabilidade
  - reduzir latência
- *Replication*
  - robustez, tolerância a falhas
  - para disponibilidade do serviço
- *Caching*
  - reduzir latência

slide de Jimmy Lin - University of Waterloo