

# What Good are Models? and What Models are Good?

Two approaches:

⇒ **Experimental Observation:** We build things and observe how they behave in various settings. A body of experience accumulates about approaches that work. Even though we might not understand why something works, this body of experience enables us to build things for settings similar to those that have been studied.

- Podem generalizar com problemas implementados ou concretos - se os atributos que os objetos

⇒ **Modeling and Analysis:** We formulate a model by simplifying the object of study and postulating a set of rules to define its behavior. We then analyze the model - using mathematics or logic - and infer consequences. If the model accurately characterizes reality, then it becomes a powerful mental tool.

- Podem simplificar domínio o modelo

- Without experimental observation, we have no basis for trusting our models.
- Without models, we have no hope of mastering the complexity that underlies distributed systems.

## Good Models

Model → for an object is a collection of attributes and a set of rules that govern how these attributes interact.

- O modelo é

- **preciso** - na medida em que a sua amáli se produz verdades sobre o objeto de interesse

- **tractável** (tractable) - se tal amáli se tornar realmente possível

As constraints Sistemas distribuídos separam **2 fundamentos**

**Feasibility**: what classes of problems can be solved?

To garantir tempo em design em implementação e teste

**Cost**: For those classes that can be solved, how expensive must be the solution be?

In conseguimos evitar solução lutar as cores

— — —

**MTBF** → mean-time-between-failures

• Failures models commonly found in distributed systems literature include:

⇒ **failstop** → um processador que falha. Uma vez parado, continua nesse estado. O facto de ter parado é detectado por outros processadores

⇒ **crash** → um processador falha ao parar. Uma vez parado, continua nesse estado. O facto de ter falhado não pode ser detectado por outros processadores

⇒ **crash + link** → Um processador falha ao tentar parar. Uma vez parado permanece nesse estado. A Lemis fails ao tentar algumas msgs → mas não atinge, duplica ou ~~recep~~ retransmite as msgs

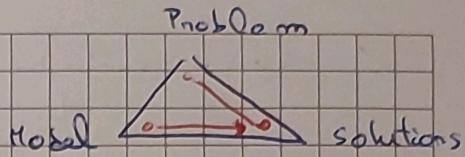
⇒ **Receive - Omission** ⇒ um processador falha ao receber apenas um sub-conjunto de msgs que realmente tentava enviar ou permanece parado

⇒ **Send - Omission** ⇒ um processador falha ao transmitir apenas um sub-conjunto das msgs que realmente tentava enviar ou permanece parado

⇒ **General - Omission** ⇒ um processador falha ao ~~receber~~ <sup>retransmitir</sup> um sub-conjunto de msgs que não foram enviadas; transmitindo um sub-conjunto das msgs que ele realmente tentava enviar ou permanece parado

⇒ **Byzantine failures** ⇒ um processador falha exibindo comportamento arbitrio

Modelo  $\Rightarrow$  visão que temos sobre o mundo  
 $\rightarrow$  pode ser o que quisermos



Modelo Synchrony:

In una msg de mala mun  
 máximo x tempo para  
 efectuar una da da acá,

Modelo Asynchrony

No max se sabe quando vai  
 chegar, só se sabe que  
~~eventualmente~~ vai chegar

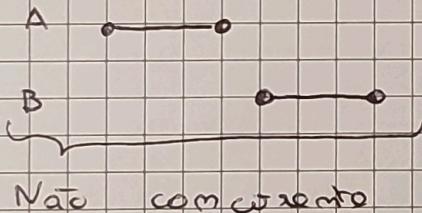
Omissivo :: fail-stop; crash-top; crash-recovery; receive,  
 send and general omissions

Assertive :: Syntactic / Semantic Byzantine

## Data Replication

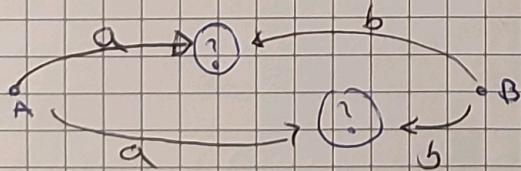
-DROWA :: Read - One - Write - All

- concurrent writes  $\Rightarrow$  2 eventos são concorrentes se terminam  
 acaba antes do outro começar



① Problema  $\Rightarrow$  Vão permitir que as réplicas devigam

Os 2 eventos podem querer escrever mas réplicas, mas  
 não conseguimos garantir que as ~~escrevem~~ réplicas  
 contêm a mesma informação



Solução: Pode ser a adição de locks.

1º  $\rightarrow$  pede o lock das réplicas

2º  $\rightarrow$  só depois efetuam a escrita

Ir com estes um Leitor quando pretende ler é obrigado  
 a adquirir locks antes de começar a leitura

(2º) Problema: Não permitem escritas do "passado"

↳ Partimos do princípio que aqui não há locks

Solução: Adicionar versões

↳ Cada réplica tem a sua versão

- Um dado cliente; 1º vai ler e ~~apenas~~ vai que existe a versão \*

2º Ao escrever, informa que a versão é  $x+1$

3º A aplica a receber, verifica que a versão que também é anterior à versão a escrever. Se assim for, ela escreve; caso contrário não

→ No caso de haver empate

$$\text{Version} = (\text{counter}, \text{pid})$$

$$\text{Counter 1} == \text{Counter 2} \quad \text{if } \text{pid1} > \text{pid2}$$

--- ---

BOWA → Não é tolerante a falhas

↳ Problemas de desempates

- a escrever, temos que esperar pelo tempo da réplica mais lenta a escrever. Isto exige conformação e comunicação entre todos os nós e a que pode trazer latência significativa.
- se uma réplica falha? Ambas

Tomaremos que 1 das réplicas falhou

Se formos realizar uma leitura, esta pode retornar valores diferentes dependendo da réplica a ser acessada.

Isto compromete a consistência dos dados, pois diferentes partes do sistema possuem visões diferentes do estado dos outros.

↳ Para garantirmos essa consistência é necessário implementar mecanismos de redundância.

## Quorums

(2)

- Two Quorums

- $Q_w \rightarrow$  write quorum

- $Q_r \rightarrow$  read quorums

Requer:

$$\bullet |Q_r| + |Q_w| > m \quad (\text{os conjuntos, interseção - se})$$

$$\bullet 2 \cdot |Q_w| > m \quad , \text{ any 2 write quorums always}\newline \text{intersect}$$

-// -// -

Ler :

$\rightarrow$  Vai ler de todos os  $Q_r$

$\rightarrow$  escolhe a versão Maior

$\Rightarrow$  Retorna essa

Escrever :

$\rightarrow$  Vai ler de todos os  $Q_r$

$\rightarrow$  escolher a versão Maior (x)

$\rightarrow$  Vai escrever com a versão  $x+1$

$\hookrightarrow$  Obvio que se ele tiver uma versão  $\rightarrow x+1 \rightarrow$  não atualiza

-// -// -

• O uso de quórum permite fazer trade-offs em vários aspectos do sistema. Os quórum permitem ajustar os requisitos de consistência, disponibilidade e desempenho de um sistema distribuído. Dependendo do tipo de Quorum utilizado, é possível obter diferentes níveis de tolerância a falhas, latência e carga máx.

• Com falha de omissão, a tolerância a falhas pode ser maximizada usando quorum de maioria estrita: Em casos de omissão, com que nós parem de responder sem falhar completamente, a maximização é alcançada usando quórum de ~~aceita~~ maioria estrita:

↳  $|Q_{n,1}| = |Qu| = \lceil \frac{m+1}{2} \rceil \rightarrow$  Isso significa que o tamanho do Quórum para operações de escrita e op. de leitura é igual e é calculado como metade do nº total de réplicas ( $m^1$ ) acrescido de 1, arredondado para cima. Esse valor garante a maioria estrita para as operações.

↳  $|Q_{n,1}| = \lceil \frac{m+1}{3} \rceil \rightarrow$  Dá a operações de escrita menos custosas em termos de desempenho e latência.

•  $|Q_{n,1}|$  pode determinar o Working bias (níveis de carga de trabalho). Isso afeta o custo de leituras e compacta o custo da escritas.

↳ Se o nº de ~~uma~~ réplicas envolvidas em operações de escrita for maior, o custo dessas leituras será maior em termos de desempenho e latência.

↳ Além disso, o tamanho do Qr também afeta o custo de operações de escrita, pois a concorrência entre um nº maior de réplicas pode contribuir para atrasos e impactar o desempenho das escritas.

## Controlo de concorrência

- ⇒ falta de controlo de concorrência em operações de escrita pode levar a "inconsistências".  
 (Isso ocorre quando operações de escrita interferemumas mas outras, resultando em resultados imprevisíveis)
- ⇒ Leituras realizadas durante uma escrita concorrente podem levar a resultados inexplicáveis.
  - ↳ Ex: Tomos uma escuta com um da memória concorrentemente com operações de leitura.

Imaginemos que a escuta está num estalo. Interrompe-se durante a execução, e as leituras podem capturar esse estado transitoriamente, resultando em valores inconsistentes ou imprevisíveis.
- ⇒ Execução concorrente pode facilmente violar a semântica da app. No entanto, quando as operações concorrentes são feitas juntas, pode ocorrer uma situação em que é incremento de uma op. Seja dividida pelo incremento da outra, resultando num valor menor que o esperado.

Podemos de adicionar mecanismos de controlo: bloqueios, transações ou técnicas de sincronização

Ex: lock → quando o guia prático

↓

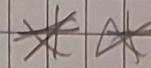
Caso um lock falhe ou exista deadlock, fica difícil de resolver

① Total order: Nunca 2 operações aparecem

fora de ordem

## Distributed Consensus

→ Serve para resolver o prob das réplicas



A abordagem conceptualmente complementar conhecida é  
fácil de entender para a replicação é o uso do  
protocolo de comunicação da difusão atómica  
(atomic broadcast)

Clientes e réplicas comunicam por meio de primitivas  
abcast (difusão atómica) e abdeliver (entrega atómica).

Através disto garantimos:

• **integridade**: Se um processo abdeliver  $m$  (ou seja, entrega a msg  $m$ ), ele faz no máximo uma vez e somente se  $m$  tiver sido previamente abcast (ou seja, difundido atomicamente). Essa propriedade garante que não há msgs duplicatas ou perdas de msgs.

• **terminação**: Se uma msg abcast em é não efetuado então ele eventualmente abdelivers em.

↳ Ou seja esta propriedade garante que a difusão das msgs ocorre de forma completa e que todas as msgs recebem as msgs

• **acordo (agreement)**: garante que ~~as execuções~~ todas as réplicas concordam sobre quais msgs são entregues, evitando divergência na replicação

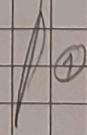
• **Total order**: Garantir ordenação consistente e previsível das msgs entre elles



Isolation: Não vamos ter 2 escritas ao mesmo tempo

|| No partition

↳ usam QW



Como é que as réplicas concordam umas com as outras?

- As réplicas devem escolher a operação satisfazendo as seguintes propriedades:
  - Non-triviality: a operação escolhida foi proposta
  - Agreement: Nenhuma 2 réplicas escolhem diferente
  - Termination: Todas as réplicas que não falharam eventualmente votam

O Paxos algoritmo faz-se messas pro piores.

# The Paxos algorithm

①

=> Two significant drawbacks:

- It is exceptionally difficult to understand
  - ↳ Fíjate esto: hay versiones muitas más extensas para explicar o algoritmo de forma simple.
  - ↳ Es preciso leer varios documentos.  
↳ Este processo pode levar até 1 ano
- Is that it doesn't provide a good foundation for building practical implementations
  - ↳ Baeten's descriptions are mostly about single-owner Paxos; He sketched possible approaches to multi-Paxos, but many details are missing.
  - Another problem is that Paxos uses a symmetric peer-to-peer approach at its core (though it eventually suggests a weak form of leadership as a performance optimization)  
↳ Es más fácil escalar con líder, o dejar otros líderes coautores como decisores

"There are significant gaps between the description of the Paxos algorithm and the needs of a real-world system ... the final system will be based on an unproven protocol"

Because of these problems, we concluded that Paxos doesn't provide a good foundation either for system building or for education.

↳ Com isto surgiu o algoritmo de Raft

- Raft → we separate p - leader election  
 - log replication  
 - safety  
 - membership changes

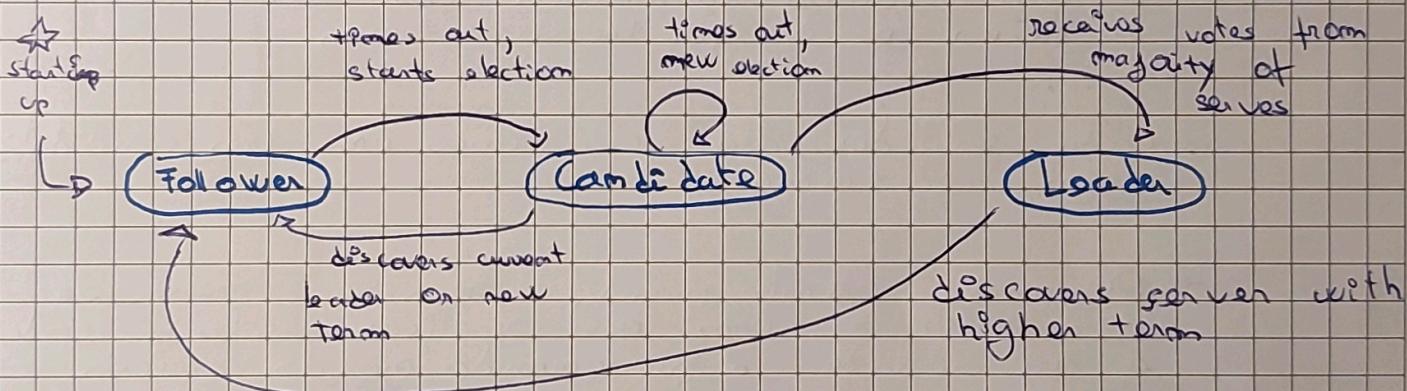
## The Raft consensus algorithm

- 1º escolher o líder
- 2º o líder passa para controlo o log nos pares  
 põe grida os "replicated log"
- No caso de o líder falhar o mecanismo escolher outro líder
- **leader election** :: a new leader must be chosen when an existing leader fails
- **log replication** :: the leader must accept log entries from clients and replicate them across the cluster forcing the other logs to agree with its own
- **safety** :: the key safety property for raft is the State Machine Safety Property : if any server has applied a particular log entry to its state machine then no other server may apply a different command for the same log index  
 Raft ensures this property: the solution involves additional restrictions on the election mechanism
- **membership changes** ⇒ vai ser visto mais a frente

Raft assegura cada uma dessas propriedades:

- ⇒ **Election Safety** :: at most one leader can be elected in a given term
- ⇒ **Leader Append-only** :: a leader never overwrites or deletes entries in its log; it only appends new entries
- ⇒ **Log machine** :: if two logs contain an entry with the same index and term, then the logs are identical for all entries up through the given index
- ⇒ **Leader completeness** :: if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms
- ⇒ **State machine safety** :: if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index

### Server States



**Follower** :: are passive: they issue no requests on their own but simply respond to requests from Leader and can be voted.

**Leader** :: handles all client request (No caso de um cliente mandar com pedir a um follower este deve redirecionar para o leader)

**Candidate** :: is used to elect a new leader

## Leader Election

- Raft uses a heartbeat mechanism to trigger leader election
- Quando o servidor começa, começam todos os seguintes
  - ↳ Fica neste estado até se tornar só o líder ou então ser informado que há um líder
  - ↳ No caso de não receber nenhuma mensagem dentro de tempo chamado election time, assume que não há um líder disponível e começa uma nova eleição

Começo da eleição:

- Implementam o current term
- É passo para candidate

A candidate wins an election:

- Se receber votos da maioria dos servidores do cluster ~~per~~ for the same term
- Cada servidor vota pelo menos em um candidato num dado termo
- Quando alguém manda heartbeat messages para todos os outros os réplicas estão trabalhando à autoridade e preventindo novas eleições

Waiting for votes:

- Pode receber AppendEntries RPC de outro servidor a reclamar a liderança.
  - ↳ Se o termo que recebeu for maior ou igual ao de aceita
  - ↳ Se for menor, irá rejectá-lo

- Raft uses a randomized election timeouts to ensure that split votes are rare and that they are resolved quickly. (150 - 300 ms)

## Log Replication

- ↳ Líder recebe um pedido
- ↳ Adiciona-o aos seus logs e em paralelo manda para todos os servidores uma replicação do log (AppendEntries RPCs)
- ↳ Quando a entry for safely replicated, o líder aplica a operação no seu estado e retorna o resultado para o cliente
- ↳ Se um follower crasht ou estiver lento, o líder vai repetir o AppendEntries RPC imediatamente (mesmo de pois de já ter respondido ao cliente) até todos os réplicas guardarem eventualmente esse log entries.
- ↳ Cada entry contém:
  - ↳ term number when the entry was received by the leader
  - ↳ comando
- ↳ The líder keeps track of the highest index it knows to be committed, and it includes that index in future AppendEntries RPC (including heartbeats) so that others servers eventually find out.
  - ↳ Once a follower learns that a log entry is committed, it applies the entry to its local state machine (in logader)

Propriedades a serem respeitadas:

↳ Se 2 entradas com dif logs tem o mesmo index e termo, entao armazemam o mesmo comando

Ao mandar com comando \$ o líder informa o term e o index  
para seguir \$ assim garantimos que uma réplica  
consegue guardar os logs na mesma ordem

- ⇒ num mesmo termo, os index são sucessivos
- ⇒ se uma réplica receber um index maior  
do que o que tem irá recusar
- ⇒ Se não confirma que recebeu o líder ao  
receber a confirmação tem a certeza que  
os logs dessa réplica estão iguais aos do

In Raft, the leader handles inconsistencies by forcing the followers' logs to duplicate its own. This means that conflicting entries in follower logs will be overwritten with entries from the leader's log.

To bring ~~an~~ a follower's log into consistency with its own, the leader must find the latest log entry where the two logs agree, delete any entries in the follower's log after that point, and send the follower all of the leader's entries after that point.

O líder mantém para cada follower a next index  
 que representa o próximo index que o líder vai enviar  
 aquele follower.

- Se ao mandar o follower receber o líder  
 decrementa o next index e envia o AppendEntries.  
 Eventualmente o líder e o follower ficam com  
 同步 (síntesis).
- ⇒ O líder nunca elimina ou dá overwriting aos  
 seus próprios logs.

## Safety

Precisamos de garantir que um dado follower  
 possa ficar vulnerável enquanto o líder manda  
 several log entries, e o próprio possa ser eleito  
 líder sem dar overwrite these log entries que  
 devem ser recebidos.

### Election Restriction

↳ logs entries only flow in one direction, from  
 leaders to followers

↳ leaders never overwrite existing entries in  
 their logs.

↳ Um candidato tem que contar a maioria  
 dos logs.

↳ Se receber um Request Vote e os logs  
 que ele têm estiverem + atualizados, etc  
 → aceita

↓  
 compara-se os termos e os index

## Committing entries from previous terms

Raft never commits log entries from previous terms by counting replicas

## Follower and candidate crashes

Se alguém receber um AppendEntries request que já inclui mais entradas do log presentes no seu log, ele ignora essas entradas na nova solicitação

## Timing and availability

The system satisfies the following timing requirement:

$$\text{broadcastTime} \ll \text{electionTimeout} \ll \text{HTBF}$$

↓  
is the average time it takes a server to send RPCs in parallel to every server in the cluster and receive their responses

↓  
é o tempo de eleição  
↓  
10 - 500 ms

↓  
 $t_f$  is the average time between failures for a single server.  
↓  
menos de 1 s  
alguns meses ou mais

## Cluster membership changes

→ No caso dos nodes falharem, é necessário mudar a config.  
 Isso pode ser feito reiniciando todos os nodes, no entanto é mau porque deixa o sistema inavailable durante algum tempo.

Para resolver isso usamos o join consensus

→ 1º mandamos o consensus.

- log entries are replicated to all servers in both configurations
- any server from either configuration may serve as leader
- A agreement (for elections and entry commitment) requires separate majorities from both the old and new configurations.

-----

→ Quando um líder recebe um pedido para mudar a config. from C<sub>old</sub> to C<sub>new</sub>, ele guarda os logs e replica.

→ Ele decide quando aplicar.

→ Todas as decisões futuras são feitas a partir de C<sub>new</sub>, old.

→ Quando o líder crash, um novo líder pode escolher entre C<sub>old</sub> e C<sub>new</sub>.

\* In any case, C<sub>new</sub> cannot make unilateral decisions during this period

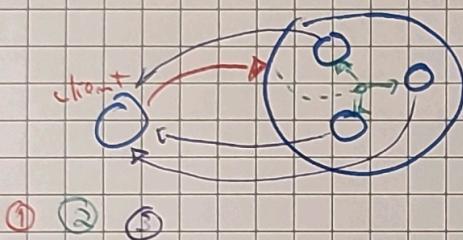
→ No caso de C<sub>old</sub> never tiver sido committed, apenas candidatos com essa config é que serão aceitos

# State Machine Replication

- Vamos considerar replicação de uma qualquer função genérica
- As falhas que consideramos são falhas de omissão (falta de msg ou um processo pode parar de trabalhar)

## Replicação ativa

- envolve a execução simultânea das funções replicadas nas dif réplicas; com a coordenação entre as réplicas para garantir a consistência dos resultados



① ② ③

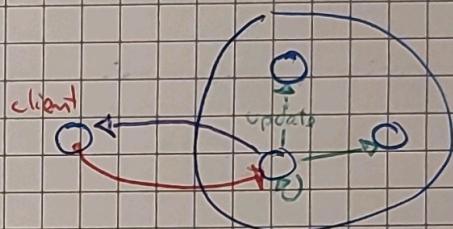
- apenas funções determinísticas
- ↳ produzem o mesmo resultado
- Replicação é transparente
- Falha de réplicas é transparente

## Integridade de réplicas

- Quando desejarmos aumentar ou restaurar a resiliência de um sistema desbibulado, podemos adicionar réplicas. Antes dessas réplicas adicionais podem lidar com solicitações e assentir que o sistema está está consistente com o sistema das restantes. Isto deve ser feito com o mínimo de interrupção possível

## Replicação passiva

- envolve a execução sequencial de uma réplica primária (principal) que executa a função e replica o resultado para as demais réplicas, garantindo assim a consistência



Em termos de transparência:

- transparência para o cliente
  - ↳ o cliente não sabe que existem réplicas de back up, pois este só conhece a primária e não está envolvido no processo das réplicas
- pode não ser transparente
  - no caso da réplica principal falhar o cliente precisa de se envolver no processo das estan réplicas (ento do switch e redirecionar os seus pedidos para a outra réplica. (Adicão de manipulação ou configuração no lado do cliente)
- não transparente:
  - ↳ o cliente está envolvido no processo de replicação

Replicação ativa

# Database Replication

Algumas características importantes dos sistemas de gerenciamento de bancos de dados transacionais:

1º transações: Deve garantir que todas as réplicas executam as transações de forma consistente.

2º solicitações multi-interação: O banco de dados pode precisar lidar com solicitações que podem envolver várias interações com a base de dados. Essas solicitações podem incluir várias consultas ou atualizações que precisam ser executados de forma consistente em todas as réplicas.

3º serviços altamente concorrentes → bases de dados devem ser capazes de lidar com várias solicitações ao mesmo tempo de forma eficaz e que garanta a consistência.

- A replicação da DB é um caso particular de replicação da máquina de estado (onde o estado a ser replicado não é só os dados, mas também os metadados, logs de transações...)
- As transações têm características especiais: como atomicidade, consistência, isolamento e durabilidade.
- Precisa ter mecanismo de concorrência
- Técnica de Primary / Backup (Replicação passiva)
- A replicação pode ser:

## Sinronia

- todos os atualizações são confirmadas após serem replicadas em todas as réplicas.
- garante que todos terão uma cópia consistente dos dados mas p/tno hzg latencia

## Assíncronia

- As atualizações são confirmadas logo no servidor primário e replicadas posteriormente às réplicas.
- leva a inconsistência temporária entre réplicas

- A replicação pode ser classificada como

### Singla - Master

- Único master, apenas este pode receber atualizações e as réplicas são leitores

### Multi - Master

- Várias réplicas podem receber atualizações e propagar isso para outras réplicas

⇒ Maior escalabilidade e flexibilidade

— / — / —

- Sem reconciliação : Não há resolução de conflitos entre réplicas inconsistentes, em vez disso o objetivo é garantir que todas as réplicas estejam consistentes desde o início eliminando a necessidade de reconciliação posterior

- Uso de técnicas de comunicação em grupo

- Protocolos Multi - master : A réplica permite a atualização em qualquer réplica (recebe e propaga para outras)

⇒ Usam Atomic Multicast para falar entre si e todos os msg. são totalmente ordenados para todos as réplicas

- Usam o Modelo do Replicante State - Machine

— / — / — / —

- Execução conservativa → abordagem para controlar a execução de transações em sistemas desbalanceados, a fim de evitar conflitos e garantir a consistência dos dados.

↳ As transações são classificadas em classes de conflito com base nas suas operações e recursos utilizados

- Transações ordinárias, de acordo com a sua classificação de conflito:
  - ↳ evitamos executar essas transações simultaneamente (\*)
  - ↳ podemos executar outras op. em paralelo visto que sejam de conflito diferente.

São exceções sequencialmente

## Execução otimista

- > Permite a execução concorrente de transações conflitantes, com a verificação de conflitos ocorrendo após a execução.
- > Não há ordem prévia
- > Após a execução, ocorre verificação de conflitos
  - b) Se for detectado um conflito, mecanismos de resolução de conflitos são aplicados, como a reversão de 1 ou 2 transações envolvidas no conflito.
- > É otimista porque assume que a maioria das transações não contraria com conflito e podem ser executadas sem restrições.

