CD

-Resumo Ch7-

Considerações Iniciais

Razões para replicação:

- **Aumentar Reliability:** Mesmo que uma replica crash podemos continuar a trabalhar
- **Data Performance:** Ajuda no escalonamento em termos de **tamanho** (quando o # de processos a aceder aos dados vai aumentando, aumentamos performance ao fazer com que eles acedam aos dados por replicas diferentes) e em termos **geográficos** (colocar uma replica de dados mais geograficamente próxima de um cliente torna o seu acesso mais rápido)

Problemas da replicação:

- Problemas de consistência...
- Modificaçoes numa replica tem de ser realizadas em todas as suas copias

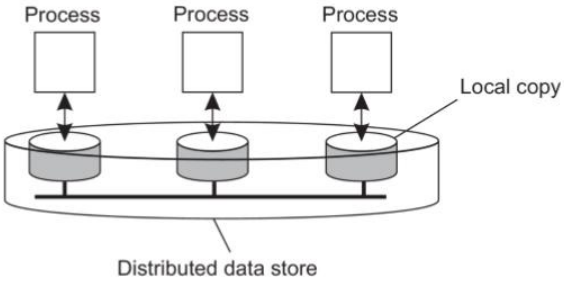
Replication as a scaling technique:

- Podemos aumentar performance através de replicação, mas podemos ser obrigados a gastar mais network bandwidth para manter as replicas up to date
- Ou seja, tentamos resolver scalability problems criando replicas, mas quantas mais replicas tivermos, mais difícil sera mantelas consistentes o que nos leva a problemas de scalability...
- Uma solução é relaxar os requerimentos de consistência e não obrigar a que todas as copias estejam updated at all times...o grau de consistência depende portanto do sistema que estamos a criar

Data-Centric Consistency Models

-Considerações-

Data Store:



- Termo geral que engloba elemntos de dados partilhados (como shared memory, databases, file systems, etc)
- Pode estar fisicamente distribuído em múltiplas maquinas
- Cada processo que pode aceder a dados de um data store e assumido como tendo um local copy desse data store
- Write operations num local data store são propagadas para todas as outras copias

Consistency Model:

- Contrato entre processos e o data store
- Basicamente um processo aceita obedecer a certas regras e em troca o data store promete funcionar corretamente (i.e não retornar valores todos mamados)
- **Maior restriction -> Mais fácil de usar**
- **Less Restrictions -> Melhor performance (mas preciso ter mais cuidados)**

Nota++: Data Centric consistency models procuram fornecer systemwide consistente view na data store. Assume-se que processos concorrentes podem simultaneamente fazer updates a esse data store e portanto e necessário fornecer consistencia.

Data-Centric Consistency Models

-Continuous Consistency-

Continuous Consistency:

- Existem formas de uma aplicação especificar que incosistencias pode tolerar
- 3 tipos de inconsistency axes são:
  - **Numerical Deviation**
    - Absolute Numerical Deviation – e.g “Valor pode estar incorreto num range de +-0.2”
    - Relative Numerical Deviation – e.g “Valor pode deviar 0.5% do valor correto”
    - Tambem podemos interpretar numerical deviation em termos do numero de updates aplicados a uma certa replica mas que ainda não foram vistos por outros
  - **Staleness Deviation**
    - Remete para a ultima vez que uma replica foi atualizada
  - **Deviation in Ordering of Updates**
    - Permitir que os upadtes não sejam feitos pela mesma ordem em todas as replicas

Conit:

- Unidade de consistência, i.e, especifica os dados sobre os quais devemos medir a consistência
- Para serem uteis precisamos de estabelecer protocolos de consistência

- Precisamos que developers especifiquem os consistency requirements para as suas aplicações

Nota: Aplicar estes deviations de contínuos consistency obriga a que as replicas saibam o quanto estão a deviar umas das outras!

Data-Centric Consistency Models

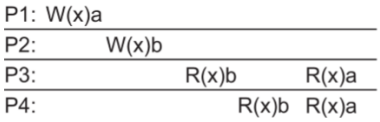
-Consistent Ordering of Operations-

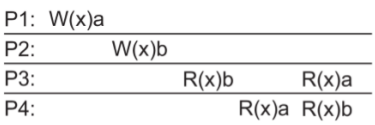
Modelos que lidam com o ordenamento de operações em shared replicated data de forma consistente

Usados to augment continuous consistency no sentido que, quando updates a replicas tem de ser commited, replicas vao ter de chegar a um acordo num ordenamento global desses commits/updates

Sequential Consistency:

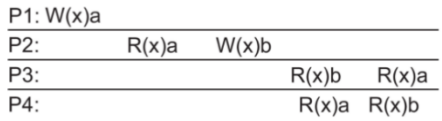
- "*O resultado de qualquer execução é o mesmo se todos operações em todos processos forem executadas na mesma ordem sequencial, e as operações de cada processo individual aparecerem na sequencia na ordem especificada pelo programa*"
- Traduzido significa que quando processos correm concorrentemente em maquinas diferentes, qualquer execução de read/write operations é aceite MAS todos os processos tem de ver a execução pela mesma ordem
- Exemplos:
  - **A) Está correto pois tanto o processo 3 com o 4 lêm primeiro a o resultado da operação de write efetuada pelo P2 e depois a do P1**
  - **B) Está incorreto pois tanto o processo 3 lê primeiro o Write do processo 2, e o processo 4 lê primeiro o write do processo 1. Não há Sequential Consistency pois P4 e P3 vêm uma ordem de execução das write operation de P1 e P2 diferentes!!**

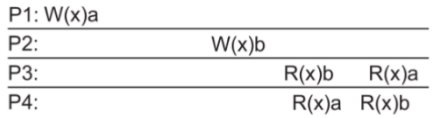




Causal Consistency:

- "*Escritas que estão potencialmente relacionadas por causalidade, têm que ser vistas por todos processos com a mesma ordem. Escritas concorrentes podem ser vistas com ordens diferentes por diferentes processos.*"
- Traduzido significa que quando uma escrita causalmente precede outra, todos os processos vão ter que ver os resultados dessa escrita pela mesma ordem. Caso as escritas não estejam causalmente relacionadas, a ordem de leitura é cagativa.
- Em geral representa um "weakening" da sequential consistency
- Exemplos:
  - **A) Está incorreto pois a escrita de P1 causalmente prece a escrita de P2 (visto que P2 lê primeiro o resultado da escrita de P1 antes de fazer a sua propria escrita) e P3 e P4 lêm os resultados por ordens diferentes**
  - **B) Está correto pois a escrita de P1 e P2 não estão causalmente relacionadas (mas note-se que violaria Sequential Consistency)**





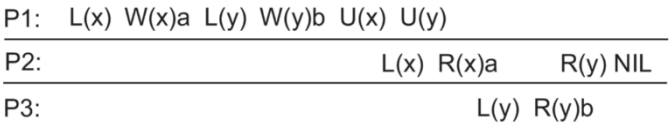
- Implementar Causal Consistency implica que se mantenha um registo sobre que processo viu que writes
- Precisamos de um dependancy graph de que operações estão dependentes de que outras operacoes
- Nota: A forma como estamos a verificar por causalidade neste processo pode gerar falsos positivos (mas nunca falsos negativos)

Grouping Operations:

- Muitas vezes, a granularidade oferecida pelos modelos de consistência que se baseiam no ordenamento de read and writes é desnecessária.

- Muitas vezes vemos que consistência entre programas que partilham dados pode ser mantida com o uso de synchronization mechanisms de **exclusão mutua** e transactions
- Basicamente o que acontece e que read and write operations devem ser limitadas a processos que tenham entrado na **Critical Section**
- Ao entrar na Critical Section o processo é assegurado como tendo toda a data no seu local store up to date (caso seja preciso esta é updated), depois pode fazer os reads e writes que quiser, e sair.
- Estas operações ficam então protegidas contra acessos concorrentes que levariam a inconsistências de dados
- Entrada e saída da zona critica pode ser feita com uso de Shared Synchronization Variables – i.e **Locks**
- Quando um processo entra na zona critica, recebe os locks relevantes, e quando sai, liberta-os
- Cada lock pode estar a ser segurado por apenas 1 processo de cada vez
- O processo com o lock pode fazer os reads e writes que pretender
- Podemos também defenir nonexclusive access a um lock, mas isto so e possível se nenhum processo estiver a usar o lock como exclusive access
- **Requirements**:
    - Receber um lock so pode suceder se todos os updates relacionados com os dados partilhados tiverem sido completados
    - Acesso exclusivo a um lock so pode suceder se mais nenhum processo estiver com acesso exclusive ou não exclusive a esse lock
    - Acesso não exclusivo a um lock so pode suceder se qualquer acesso exclusivo prévio tiver sido completado (incluindo updates nos dados associados ao lock)
- **Exemplo**:

- **A) Como o processo 3 acedeu ao data item y sem primeiro adquirir o lock, o data item y na sua local data store não foi dito para ser atualizado, logo não leu o valor correto – Entry Consistency did not occur**



- Um dos problemas com uso de Entry Consistency e associar os dados aos locks!

Data-Centric Consistency Models

-Eventual Consistency-

Eventual consistency:

- O quão rápido e que updates devem ser propagados para **read-only processes?**
- Muitas vezes developers optam por cagar, e escolher propagar esses updates de forma lenta, assumindo que, se os clientes forem sempre redirecionados para a mesma replica, nunca vão experienciar inconsistências, logo não há stress
- Bom exemplo disto no mundo real são os websites. Se um user aceder ao site e este tiver sido updated, mas o update ainda não tiver sido propagado para a replica a que este acede, ele não notara diferenças nem inconsistências.
- Há vários exemplos de distributed e replicated databases que toleram high degree of inconsistency, e todos tem em comum o facto de que, se nenhum update for realizado durante muito tempo, **todas as replicas irão, eventualmente tornar-se consistente** – A isto se chama Eventual consistency
- Data stores com consistência eventual tem a propriedade de que, quando não existem write write conflcits, todas as replicas convergem para copias idênticas umas das outras, tendo apenas que se garantir que os updates são propagados para todas as replicas
- This is consistency is VERY CHEAP AND EZ TO IMPLEMENT WOOHOO! (Mas pode dar merda se um user aceder a mais do que uma replica...)

Consistency VS Coherence

Consistency model:

- Descreve o que deve ser esperado quando um conjunto de processos concorrentemente opera sobre um set de dados. Um set pode ser dito consistente se aderir as regras descritas pelo modelo

Coherence Model:

- Descreve o que deve ser esperado quando se trata de apenas 1 único data item. Assume-se que esse data item e replicado
- Diz-se que há coerência se todas as copias seguirem as regras associadas ao respetivo modelo de consistência

Client-Centric Consistency Models

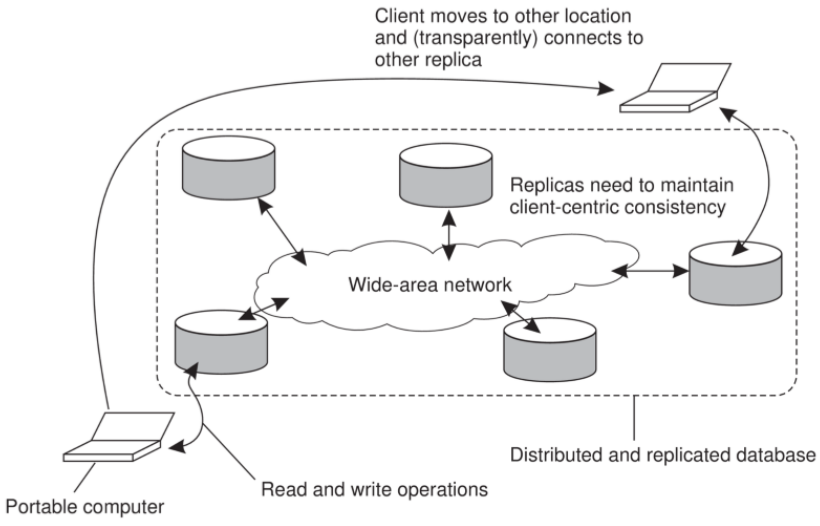
- Considerações –

Usados em classes especial de Distributed Data Stores – Data stores caracterizados pela falta de updates simultâneos (ou que quando acontecem pode ser facilmente resolvidos)

Maioria das operações involve ler dados

Estes data stores oferecem fracos modelos de consistencia (como Eventual consistency) MAS introduzindo special cliente-centric consistency models podemos esconder as inconsistências de forma barata!

Como exemplo, usarmos Eventual Consistency (which is weak) pode dar merda quando um client é móvel e pode aceder a mais do que uma replica



Porem, introduzindo Client Centric Consistency Models, podemos aliviar este problema!

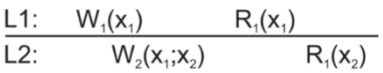
TL;DR – Client Centric Consistency Models sao usados para garantir consistencia de acesso a data stores por apenas 1 cliente

Client-Centric Consistency Models

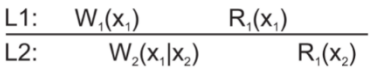
- Monotonic Reads –

Monotonic Reads:

- "*Se um processo ler o valor do dado x, qualquer operação de leitura sucessiva sobre x* ***pelo mesmo processo*** *retorna o mesmo valor ou um mais recente.*"
- Traduzido, Monotonic Reads garante que apos um processo ter visto um valor de x, nunca vera um valor mais velho de x
- Exemplos:
  - **A) Está correto pois o processo P1 criou a versao x1 no local data store L1 e depois lê essa versão (no problem there). No local data store 2, o processo P2 primeiro produz a versao 2 de x, utilizando a versao 1, logo quando P1 executa um read no store L2, vai encontrar a versao mais recente de x, x2!**



  - **B) Está incorreto. P1 cria na mesma a versao x1 no local data store L1, e lê essa versao no mesmo store. Porem a versao que P2 produz no L2 é concurrente de x1 (i.e, não é uma versao atualizada de x1, mas sim uma unrelated que não tem x1 em conta). Portanto quando P1 lê x no L2, não pode ler x2, teria que ler x1!**

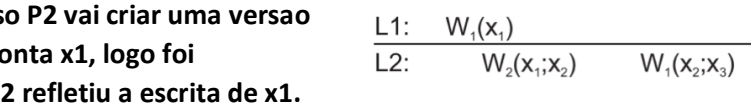


Client-Centric Consistency Models

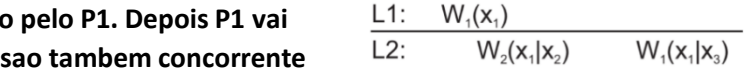
- Monotonic Writes –

Monotonic Writes:

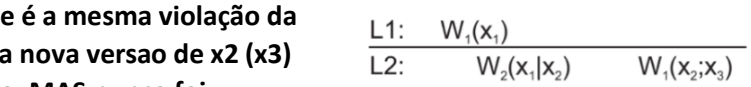
- "*Uma operação de escrita por um processo num dado x é completada antes de qualquer operação de escrita sucessiva em x* ***pelo mesmo processo.***"
- Basicamente significa que se tivermos 2 operacoes de escrita sucessivas pelo mesmo processo, Wk(xi) e Wk(xj), então independentemente de onde acontecer, teremos que ter também algures W(xi;xj) para confirmar que xj foi feito tendo em conta a escrita de xi (sendo que este ultimo W pode ser realizado por qualquer processo)
- Exemplos:
  - **A) Está correto pois o processo P1 criou a versao x1 no local data store L1. Depois, no store L2, o processo P2 vai criar uma versao updated de x2, que tem em conta x1, logo foi assegurado que a escrita de x2 refletiu a escrita de x1. Quando P1 se move para o L2, encontra uma versao updated de x1, x2, e escreve sobre ela x3**



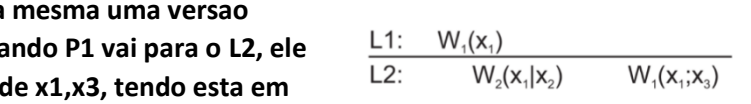
  - **B) Está incorreto. P1 cria na mesma a versao x1 no local data store L1. P2 produziu uma versao concorrente de x1, i.e, que não reflete o write inicial efetuado pelo P1. Depois P1 vai para o L2 e cria uma nova versao tambem concorrente de x1, x3. Ou seja, P1 escreveu uma nova versao de x sem ter avido primeiro uma escrita do tipo W(x1;xN) ! Foi violdada Monotonic Writes**



  - **C) Está incorreto. Basicamente é a mesma violação da situação B. P1 tenta fazer uma nova versao de x2 (x3) que tem em consideração esta, MAS nunca foi acknowledged o primeiro write (rip x1…)**



  - **D) Está correto. P2 escreve na mesma uma versao concorrente de x1, porém quando P1 vai para o L2, ele proprio faz uma nova versao de x1,x3, tendo esta em conta. Woo saved!**

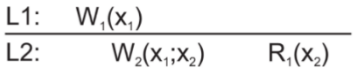


Client-Centric Consistency Models

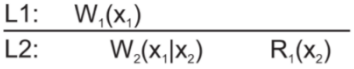
- Read your writes –

Read your writes:

- "*O efeito da operação de escrita por um processo nos dados x, será sempre visto pelas operações de leitura sucessivas em x* ***pelo mesmo processo.***"
- Resumidamente, uma write operation é sempre completada antes de uma read operation sucessiva, realizada pelo mesmo processo que fez o write. Este read pode ser feito EM QUALQUER LUGAR, tem é que ser feito.
- Exemplos:
  - **A) Está correto pois o processo P1 criou a versao x1 no local data store L1. Depois, no store L2, o processo P2 vai criar uma versao updated de x2, que tem em conta x1. Quando P1 vai para o L2 vai entao encontrar a versao atualizada de x1, x2, e le-la. Efetivamente ele não leu o write que escreveu (x1), mas como x2 é um follow up de x1, e tem-no em conta, P1 pode ler x2 à vontade e não violar read your writes.**



  - **B) Está incorreto pois o processo P1 criou a versao x1 no local data store L1. O P2 no L2 vai criar uma versão concorrente de x1, ou seja não a tem em consideração. Quando P1 lê x2 no L2 isto viola o Read your Writes pois x2 não reflete x1! Logo P1 nunca leu a escrita de x1 :(**

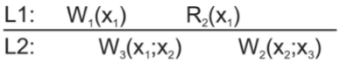


Client-Centric Consistency Models

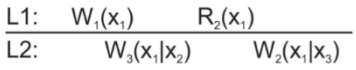
- Writes Follows Reads –

Writes Follows Reads:

- "*Numa operação de escrita por um processo sobre os dados x, após uma operação de leitura prévia sobre x* ***pelo mesmo processo****, é garantido que ocorre sobre o mesmo valor de x que foi lido ou um mais recente.*"
- Ou seja, qualquer operação de write sucessiva por um processo num data item x, será realizada sobre uma copia de x que esta atualizada com o valor mais recentemente **lido** por esse processo
- Exemplos:
  - **A) Está correto. P1 escreveu x1 no L1. De seguida, no L2, P3 escreveu uma versao followed de x1, que a reflete, chamada x2. Depois disto um processo P2 lê no L1 a versão mais recente de x que lá existe -> x1. Porêm, quando P2 efetua uma escrita no L2, encontra uma versão que reflete x1 e está mais atualizada, x2, logo vai escrever sobre esta e não sobre x1! (Note-se que se escrevesse sobre x1 também não haveria problema pois foi o ultimo valor que P2 leu)**



  - **B) Está incorreto. P1 escreveu x1 no L1, de seguida P3 escreveu uma versão concorrente de x1 no L2, x2, que não toma reflete x1. Depois P2 lê no L1 a versao x1. O problema acontece quando, no L2 o P2 efetua a operação de escrita concurrente que leva a criação de x3..Wait pq q aqui há erro? Wtf? O erro n deveria ocorrer apenas se a operação fosse W2(x2|x3) !?**



Replica Management

- Considerações –

O problema de saber onde devemos colocar e lançar replicas divide-se em 2:

Placing Replica Servers – Encontrar o melhor local para colocar um server que possa ser host de um data store

Placing Content – Saber qual o melhor servidor onde colocarmos o conteúdo

Obviamente antes de colocarmos conteúdo temos de colocar os servers logo vamos ver primeiro essa situação :)

Replica Management

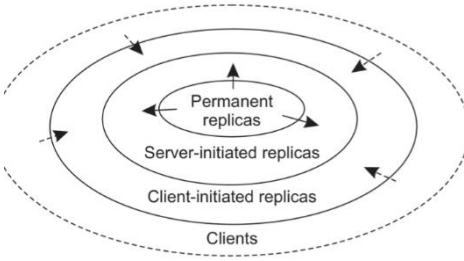
- Finding the best server location –

Finding the best server location:

- Antigamente era muito mais relevante do que agora, visto que agora temos vários large-scale data centres localizados por toda a internet e conectividade continua a ser cada vez melhor...but still lets learn a bit :/
- A optimização é sempre: Escolhar K de entre N localizações para colocarmos os nossos servers.
- Temos varias maneiras:
    - A) Selecionar um servidor de cada vez tal que a distancia media entre o servidor e os seus clientes seja mínima dado que k servidores já tenham sido colocados (temos portanto que provar N-k locais)
    - B) Ignorar posição dos clientes. Considerar o maior sistema autonomo possível, colocar um servidor no router com maior numero de network interfaces e repetir o processo para o segundo maior AS
    - Opçoes A e B são COMPUTACIONALMENTE CARAS AS FUCK DUDE! EVITAR!
    - C) Assume se que os nodes estão posicionados num m-dimensional geometric space. A ideia é identificar as K maiores clusters, e selecionar um node de cada cluster para ser o host de conteúdo replicado. Para identificar estas clusters, dividimos o nosso espaco em varias partições, aka cells. EZ

Replica Management

- Content Replication and Placement –



Note-se que podemos categorizar, no que toca a contente replication and placement, 3 tipos de replicas:

Permanent:

- Set inicial de replicas que constitui o distributed data store.
- Normalmente é um set pequeno
- Existem diversos tipos de distribuição (p.ex mirroring para websites – web site e copiado para vários mirror sites, aka copias deste, dispersos por vários servers geograficamente afastados)

Server-Initiated:

- Copias de data stores que existem para melhorar performance
- Criadas por iniciativa do dono do data store
- Dynamic Placement! (will explain in a bit)

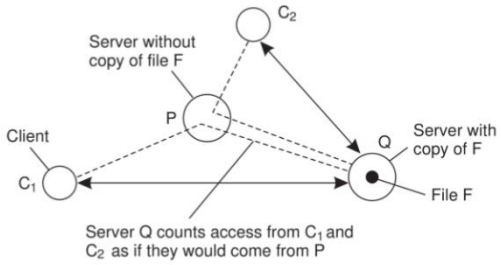
Client-Initiated:

- A.K.A Client Caches
- Sao Local Storage Facilities usadas por clientes para temporariamente guardar uma copia dos dados que acabou de pedir/adquirir.
- Normalmente, o datastore de onde a cache foi buscar dados não tem nada a ver com a cache em si, sendo que a gestão desta é feita pelo cliente
- Usadas para melhorr access times a dados

- Para prevenir o uso de dados muito antigos, os dados são mantidos em cache por um período limitado
- Sempre que dados são buscados da cache, diz-se ter ocurrido um **Cache Hit**
- Podemos também partilhar caches entre clientes..but thats kinda iffy
- Caches são colocadas ou na maquina do cliente ou num local store perto deste (há exceções mas fuck them)

Dynamic Web Content Placement (used by Server initiated Replicas):

- Cada servidor guarda o numero de acessos por ficheiro (e de onde e que veio esse acesso)
- Quando o numero de pedidos para um ficheiro especifico, F, num servidor S, cai para baixo de um threshold de deletion pre-defenido, esse ficheiro é removido de S (caso, claor, não seja a ultima copia!)
- Temos também um threshold de replicação, sempre maior que o de replication, que indica, caso tenha sido passado, que o numero de acessos a um ficheiro e tao alto que justifica que este seja replicado para outro servidor.



- Caso o numero de acessos esteja entre os dois thresholds, o ficheiro pode apenas ser migrado. Não replicado e Não apagado.

Replica Management

- Content Distribution –

Managemente de replicas tambem lida com a propagação de updated contentes para replica servers (relevantes). Temos varias coisas a considerar.

State VS Operations

- Temos de escolher o que propagar:
  - **Notificação de Update**
    - Replicas são informadas que um update aconteceu e que já não possuem dados que estão up to date.
    - APENAS uma notificação é propagada
    - +Usam pouca network bandwidth
    - +Otimos para quando temos muitos updates e poucos reads (small read-to-write ratio)
  - **Transferencia de dados entre copias**
    - São transferidos os dados entre as replicas
    - Podemos apenas enviar as alterações feitas para poupar bandwidth
    - +Otimos para quando temos muitos reads poucos updates (read-to-write ration é alto), visto ser provável que os dados atualizados vão ser utilizados/lidos
  - **Propagar a update operation entre copias**
    - AKA **Active Replication**
    - Assenta na ideia de que cada replica e representada por um processo capaz de ativamente manter os seus dados atualizados através de operações

        - Passamos portanto apenas o necessário para que a replica se possa atualizar a si mesma
        - Poupa-se bandwidth, assumindo que temos poucos parâmetros a passar
        - Mas necessita de mais processing power por parte das replicas

Pull vs Push Protocols

- Temos de escolher como propagar:
    - **Pull**
        - Cliente ou servidor pede outro servidor que lhe mande um update que tenha.
        - Usados normalmente por client caches
        - Eficiente quando o read to update ratio é baixo
        - O response time aumenta no caso de uma cache miss
        - Os clientes tem que primeiro provar o server para ver se efetivamente existe algum update antes de pedirem updates (mais mensagens a serem enviadas)
        - Usado com unicasting
    - **Push**
        - Updates são propagados para replicas sem que essas replicas tenham de pedir pelos updates
        - Normalmente usados entre permanente e server initiated replicas
        - Otimos para quando queremos alta consistencia
        - Bom para quando o read-to-update ratio é alto em cada replica
        - Obrigam a que os servers tenham de ter uma lista de replicas para mandar updates...big sad
        - Usado com multicasting

- Leases (Hybrid)
  - Uma Lease é uma promessa feita por um server que diz que enviara updates ao cliente durante algum tempo. Quando a lease expira o cliente passara a ter de fazer pull dos updates ou então pedir uma nova lease
  - Existem 3 tipos:
    - Age Based:
      - Dados a data items que dependam da ultima vez em que o item foi modificado. Assume-se que se um dado não for modificado durante muito tempo, também não e tao cedo que o sera, logo damos uma lease maior
    - Renewal-Frequency Based:
      - Server da leases maiores aos clientes que precisarem de mais vezes fazer refresh dos seus dados
      - Server terá que fazer keep track basicamente apenas dos clientes em que a data e mais popular
    - State Based
      - Quando o servidor esta a ficar overloaded da leases mais pequenas.
      - Quando estiver mais calmo, da leases maiores
      - Dinamicamente gere leases baseados no quão trabalhado esta a ser

Consistency Protocols

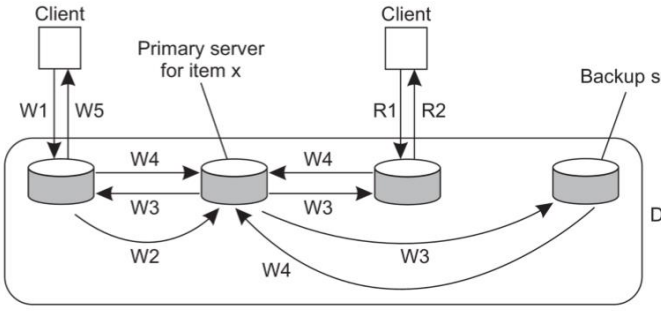
- Primary Based –

Usados para implementar modelos de Consistent Ordering of Operations

Primary Based são particularmente relevantes para implementar Sequential Consistency

Remote-Write Protocols:

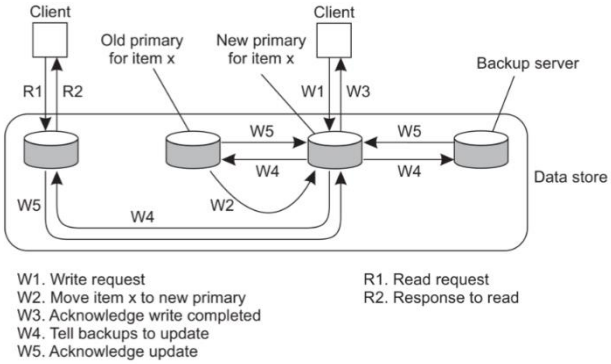
- Protocolo simples que implementa Sequential Consistency
- Todas as write operations sao reencaminhadas para um fixed single server
- Read operations podem ser feitas localmente
- Um processo que queira escrever num data item, reencaminha a operação para o Primary Server desse data item.
- Este primary server reencaminha essa operação para todos os backup servers
- Cada backup server realiza o update e manda um acknowledgement ao primário
- Quando todos os backups tivere atualizado as suas copias locais, o Primary manda um acknowledgement ao processo inicial que informara o cliente
- - Pode causar problemas de performance visto que o write demorara algum tempo a ser efetivado. Como o update é realizado como uma blocking operation



- Podemos implementar uma forma de non blocking writes mas vamos ter problemas de fault tolerance (ganhando porem alguma velocidade)

Local-Write Protocols:

- Variação do Remote Write Protocol
- Primary Copy vai migrando entre processos que queiram realizar uma write operation
- Read operations podem ser feitas localmente
- Um processo que queira escrever num data item, encontra o primary desse item, e move-o para a sua localizacao
- Tem a vantagem de que múltiplos successive write operations podem ser realizadas localmente, pelo processo que tiver o primário, enquanto que Reading processes podem continuar a ser feitos de forma local



- So e possível se implementarmos non-blockign update operations é usado, no qual os updates são propagados para as replicas assim que um primário terminar se localmente realizar updates

Consistency Protocols

- Replicated-Write Protocols –

Write operations sao realizadas em varias replicas em vez de apenas 1 (como é o caso de Primary Protocols)

Active Replication:

- Cada replica tem um processo associado que realzia update operations.
- Updates são geralmente propagados por meios das write operations que causam o update – ou seja, a operação e mandada para todas as replicas (ou podemos tmb mandar so o update)
- Obriga a Total Ordered Multicasting visto que as operações tem de ser realizadas pela mesma ordem em todas as replicas
- Para tal podemos utilizar um coordenador central – Sequencer
- Mandando para o sequencer cada operação, este associa lhe um unique sequence number e reencaminha a operação para todas as replicas (EZ)

Quorum-Based:

- Ideia baseia-se em fazer com que os clientes peçam e adquiram permissão de vários servers antes de realizarem operações de leitura ou escrita num data item
- Se N replicas existirem, é preciso criar um Read Quorum (coleção arbitraria de Nr servidores ou mais) para ler, e um Write Quorum (coleção arbitrare de Nw servidores ou mais) para escrever.

- Precisamos que:
  - Nr + Nw > N – Para prevenir read-write conflicts
  - Nw > N/2 – Para prevenir write-write conflicts

(a) Escolha correcta de read e write (b) Posivel write-write (c) Escolha correcta chamada ROWA (Read One Write All)