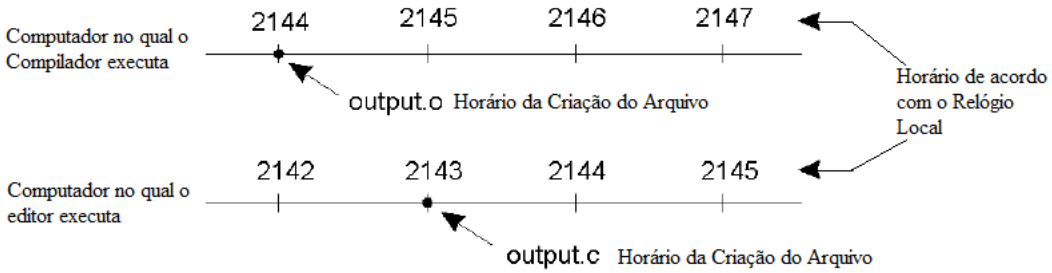
## 1) SINCRONIZAÇÃO DE RELÓGIO

Em um Sistema Distribuido conseguir acordo nos horários não é trivial, pois existem vários nós de computação, cada qual responsável por manter sua informação local de tempo.

Por exemplo, um sistema com arquivos distribuídos em dois nos. Qual arquivo é mais novo? Do ponto de vista de um observador externo quando cada maquina tem seu próprio relógio, um evento que ocorreu após outro evento pode, ainda assim, receber um horário anterior. Conforme mostra a Figura abaixo:




A temporização exata é importante em sistemas de Operações bancárias, Agendamento de pagamento, transferências, etc. Exemplos reais de problemas causados por relógio:

1. Bug do Milênio (Y2K): Não foi um problema sob o ponto de vista do relógio, mas sim sob o ponto de vista da representação de data (ano 2000 = 1900).

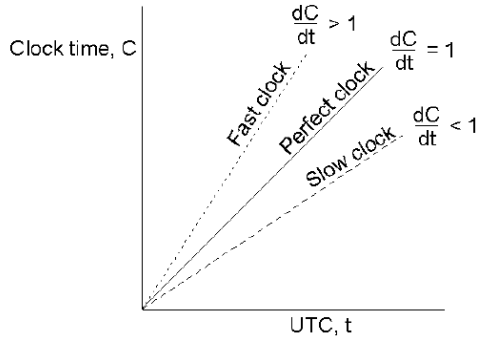
2. Bug do ano 2038: representação máxima $2^{31}$-1 de um número "signed 32-bit integer" (time_t stamp in Unix). Já corrigido em arquiteturas 64-bits.

É impossível garantir sincronismo absoluto dos relógios, uma vez que, os cristais utilizados para medir o tempo possuem frequências diferentes um do outro.

### RELÓGIOS FÍSICOS

São utilizados para fazer a Sincronização Externa, pois alguns sistemas precisam sincronizar a base de tempo local com uma base de tempo físico. Para isto é preciso ter um relógio externo para sincronização, por exemplo, o UTC.

### ALGORITMOS DE SINCRONICAÇÃO DE RELOGIOS




A relação entre tempo do relógio e UTC quando andam em taxas diferentes.

Têm como objetivo manter todas as máquinas o mais juntas possível em relação ao tempo. Os algoritmos são:

- Algoritmo de Cristian (Relogio Fisico);
- Algoritmo de Berkeley (Relogio Fisico);
- Algoritmo de Bully (Algoritmo de Eleição – Relogio Logico);
- Algoritmo Ring (Algoritmo de Eleição – Relogio Logico).

Fast Clock = Relogio Adiantado e Slow Clock = Relogio Atrasado. Taxa máxima de deriva especifica até que ponto a defasagem de um relógio pode chegar.

A diferença entre os vários algoritmos de sincronização de relógio encontra-se exatamente na maneira como essa sincronização periódica é feita.

### ALGORITMO DE CRISTIAN

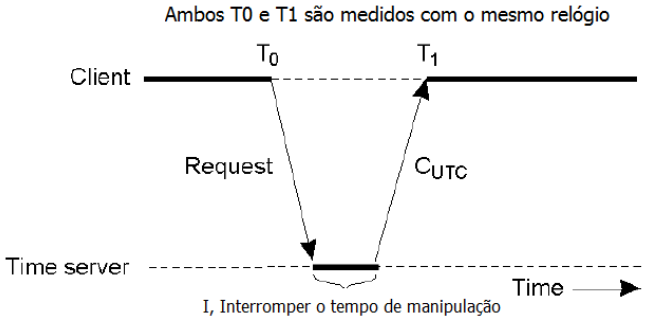
A meta neste caso é manter todas as máquinas sincronizadas a esta. Assume servidor passivo. O método diz que a estação escrava pede à mestra informação de relógio e com base nessa informação, o relógio local é atualizado.

O Algoritmo de leitura de relógio remoto é executado da seguinte forma:

- estação escrava (E) envia mensagem à mestra: "hora=?"
- mestra (M) responde: "hora=T"

Seja D a metade do tempo de trânsito total das mensagens, medido no relógio de E, e min o seu valor mínimo. Se os relógios de E e M estão corretos, o valor do relógio M, quando E recebe a mensagem "hora=T?", está no intervalo de:

**[T + min(1 - r), T + 2D(1 + 2r) - min (1 + r)]**



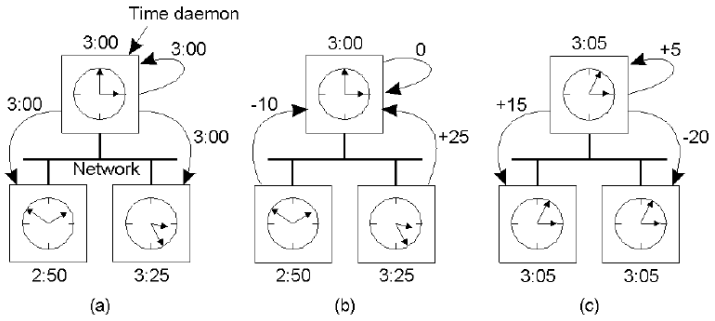

Busca o tempo atual de um servidor de tempo.

Problemas a serem tratados
– Falhas na estação mestra
– Falhas em estações escravas
– Tráfego muito variável no sistema de comunicação

Relógios não podem andar para trás. Se o cliente está atrasado a Solução é Reduzir/aumentar a freqüência do relógio, por exemplo, Uma interrupção adiciona 10mseg. Se Reduzir: interrupção → 9mseg e Adiantar: interrupção → 11mseg

Mensagens não são instantâneas existem um Tempo de propagação da mensagem e o Tempo de tratamento da mensagem (geralmente desconhecido).

**ALGORITMO DE BERKELEY**




a) O servidor pergunta relógio para todas outras máquinas
b) As máquinas respondem
c) O servidor diz a cada um como ajustar seus relógios

Não existe uma máquina com relógio sincronizado externamente. A meta neste caso é manter todas as máquinas sincronizadas entre si tanto quanto possível. Em Berkeley, servidor é ativo, pois Pergunta às máquinas o tempo, Calcula média e Notifica novo tempo a todas as maquinas.

Esse método é adequado para um sistema no qual nenhuma maquina tenha receptor WWV. A hora do Daemon de Tempo tem que ser ajustada manualmente pelo operador de tempos em tempos.

Se a hora marcada pelo daemon nunca fosse calibrada manualmente, não haveria dano nenhum contanto que nenhum dos outros nós se comunique com computadores externos.

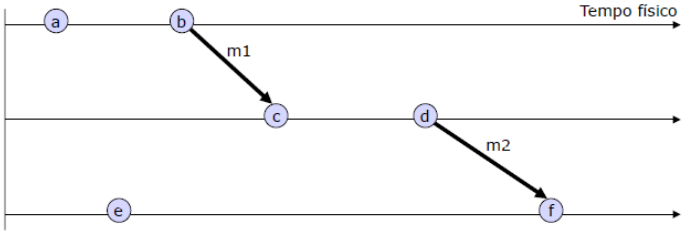
**RELÓGIOS LÓGICOS**

São utilizados para fazer a Sincronização Interna, pois os processos veem eventos ordenados pelos seus relógios locais.

**RELOGIOS LOGICOS DE LAMPORT**

Se dois processos não interagirem, não é necessário que seus relógios sejam sincronizados porque a falta de sincronização não seria observável e, portanto, não poderia causar problemas. Além do mais, de modo geral, o que importa não é que todos os processos concordem com a hora exata, mas com a ordem em que os eventos ocorrem.

Ou seja, o Sincronismo do Relogio é possível sem usar tempo absoluto, pois a ordenação não precisa de precisão, basta saber quem veio antes de quem. A execução de processos é uma seqüência de eventos. Logo, Receber e Enviar mensagens são eventos.




a → b; b → c; c → d; d → f ⇒ a → f

Não há cadeia de mensagens entre *a* e *e*, logo (a || e)

A Relação acontece-antes (→), descreve ordenamento causal entre eventos:

(1) Se a e b são eventos no mesmo processo e a acontece antes de b, então a → b.

(2) Se a é o envio de uma mensagem m de algum processo e b é a recepção da mesma mensagem m por algum processo, então a → b.

(3) Se a → b e b → c então a → c. ( R é transitiva)

Eventos Concorrentes: Dois eventos a e b são concorrentes (a||b), se não a → b nem b → a. Conforme mostra a Figura na próxima página.

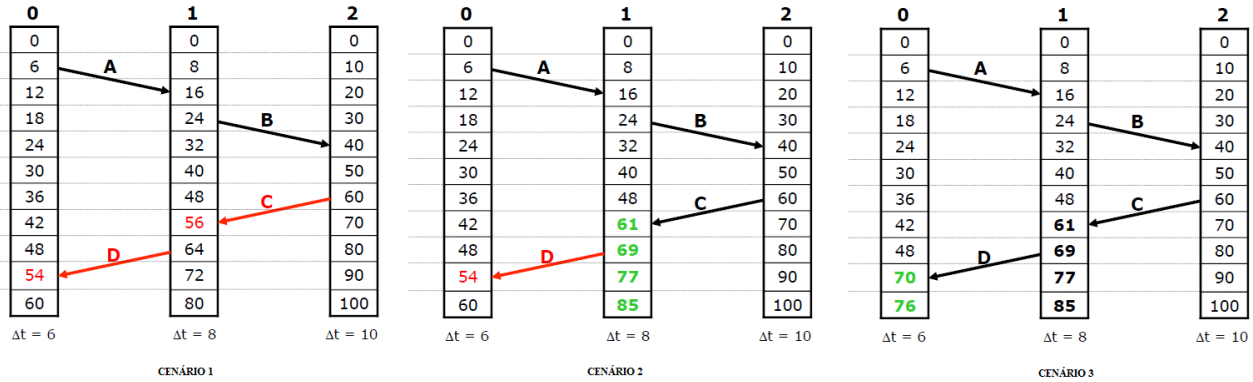
**SISTEMA DE RELÓGIOS LÓGICOS**

Um sistema de relógios lógicos é representado por uma função C que atribui um numero C(a) para todo evento a em um sistema distribuído.

Condição do relógio: Se a →b então C(a) < C(b), Se o evento a ocorre antes do evento b o sistema de relógios deveria mostrar C(a) < C(b). Condição de correção:
(C1) Para quaisquer 2 eventos a e b no mesmo processo Pi, a ocorre antes de b se **Ci(a) < Ci(b)**.
(C2)Se a é o evento enviar mensagem para um processo X e b é o evento de receber mensagem em um processo Y, então Cx(a) e Cy(b) devem ser atribuídos de tal forma que **Cx(a) < Cy(b)**.

As seguintes regras de implementação fazem com que as condições C1 e C2 sejam satisfeitas:
(RI1) Relógio Ci é incrementado entre dois eventos sucessivos em um processo Pi: **Ci = Ci + d (d>0)**
(RI2) (a) Se a é o envio de m pelo processo Pi então m é selada com **Tm=Ci(a)**.
(b) Se b é o recebimento de m pelo processo Pj então Cj recebe um valor >= Cj e > que Tm: **Cj = max( Cj, ,Tm+1 )**

**Exemplo de Relogio Logico de Lamport:** Três processos, cada um com seu próprio relógio, diferentes taxas. Algoritmo de Lamport corrige os relógios.




**Outro exemplo de aplicação de Relogio Logico de Lamport é no Multicast totalmente ordenado**. Considere a situação em que um banco de dados foi replicado em vários sites. Digamos que na cidade de Nova York uma copia do banco de dados é atualizada, por exemplo, deposito em uma conta. Depois na cidade do Rio de Janeiro outra copia é atualizada, outro deposito é realizado nessa mesma conta.

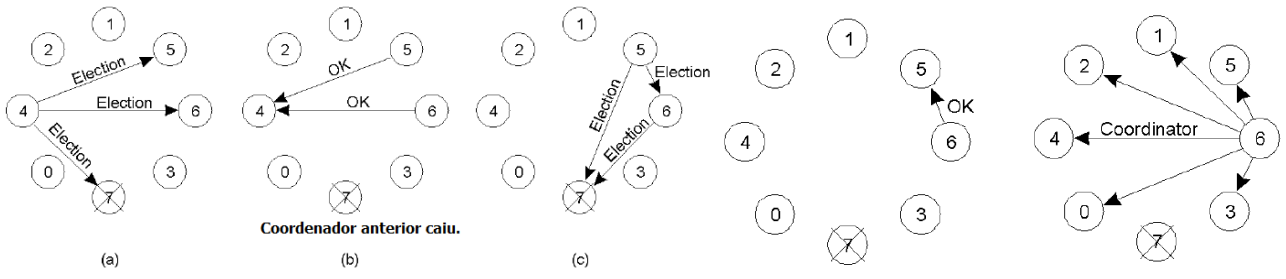
A questão importante nesse exemplo é que ambas as copias devem ser exatamente as mesmas. Situações como estas requerem um multicast totalmente ordenado, isto é, uma operação multicast pela qual todas as mensagens são entregues na mesma ordem a cada receptor.

## ALGORITMO DE ELEIÇÃO

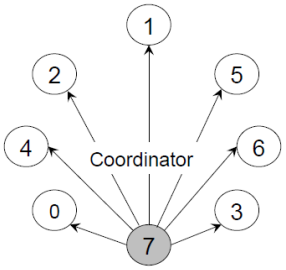
Muitos algoritmos distribuídos são baseados em processo coordenador. Os algoritmos de eleição normalmente assumem as seguintes asserções:

- Cada processo tem um número único (endereço de rede, por exemplo);
- Em uma eleição o processo de mais alta prioridade é eleito coordenador;
- A prioridade por ser obtida através de um número atribuído externamente ou
- Através de um conjunto de características como, % ocupação de CPU, memória disponível, características da interface de rede, custo de processamento, etc;
- Todo processo conhece o número do outro, apenas não sabe quem está ativo ou não.

## ALGORITMO DE ELEIÇÃO – ALGORITMO BULLY




a) Processo 4 inicia uma eleição
b) Processos 5 e 6 respondem, dizendo 4 parar
c) 5 e 6 iniciam cada um uma eleição
d) Processo 6 diz para 5 parar
e) Processo 6 ganha e diz para todos

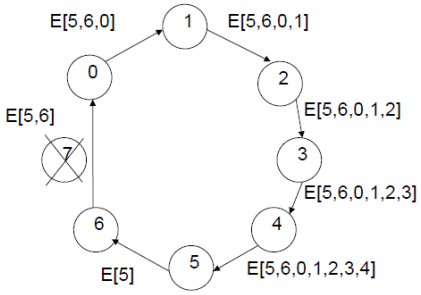



Nó 7 se recupera. Sendo o com prioridade mais alta, ele envia uma mensagem para todos anunciando que ele é o novo *coordenador*

Quando um processo nota que o coordenador não esta mais respondendo as requisições ele inicia uma eleição.

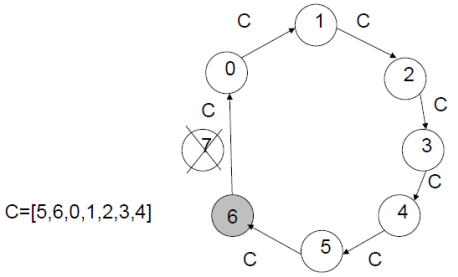
1. O Processo P envia uma mensagem election para todos os processoz com números mais altos (PID - prioridade).
2. Se nenhum responde, P ganha à eleição e torna-se coordenador.
3. Se um dos com prioridade maior responde, ele assume. Neste caso o trabalho de P está feito, ele espera a mensagem coordinator.
4. Se mensagem coordinator não acontecer, inicia nova eleição.

**ALGORITMO DE ELEIÇÃO – ALGORITMO RING**

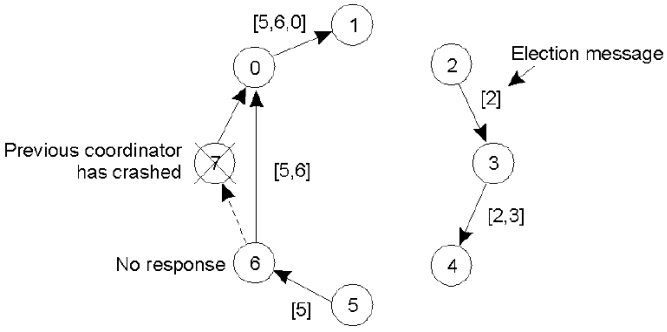



Nó 5 detecta uma falha no 7 e inicia uma eleição circulando a mensagem E.

*Obs: o problema de restauração do anel já estaria resolvido.*




A mensagem E é transformada em uma mensagem C e circulada para informar quem é o novo *coordenador*.




Processos 5 e 2 detectam falha. Duas mensagens E circulam no anel... Não há problema.

Os Processos são arranjados em um anel lógico. Todos sabem seus sucessores:

– P envia mensagem de election com seu PID

– Sucessor recebe mensagem, adiciona seu PID e passa para o próximo.

– Quando voltar a P, a mensagem muda para coordinator e volta a circular no anel.

Cada um assume que o coordenador é o maior PID da lista circulante, e a lista contém os PIDs ativos.

## 2) EXCLUSÃO MUTUA DISTRIBUIDA

Para evitar que acessos concorrentes corrompam o recurso ou o tornem inconsistente, são necessárias soluções que garantam acesso mutuamente exclusivo pelos processos. Na exclusão mutua distribuída quanto maior o numero de nós maior o tempo de resposta.

A Exlusão Mútua Distribuída tem como objetivo garantir que dois processos não usarão as mesmas estruturas de dados ao mesmo tempo, por exemplo, a = 5; a++; printf("%d",a); a = 8; a++; printf("%d",a);

As Regiões críticas nos sistemas centralizados são tratadas por semáforos e monitores, por exemplo. Em Sistemas distribuídos o sistema consiste de n processos; cada processo Pi reside em um processador diferente. Cada processo tem uma seção crítica que requer exclusão mútua. Se Pi esta executando em sua seção crítica, então nenhum outro processo Pj esta executando em sua seção crítica.

**ALGORITMOS DE EXCLUSÃO MUTUA DISTRIBUIDA**

Algoritmos distribuídos de exclusão mútua podem ser classificados em duas categorias diferentes: Soluções baseadas em ficha e Soluções baseadas em permissão.

As Soluções baseadas em ficha consegue a exclusão mútua entre os processos com uma só ficha disponível quem tiver com a ficha pode acessar o recurso compartilhado. Quando o processo termina de utilizar o recurso a ficha é passada para o próximo processo está esperando utilizar o recurso.

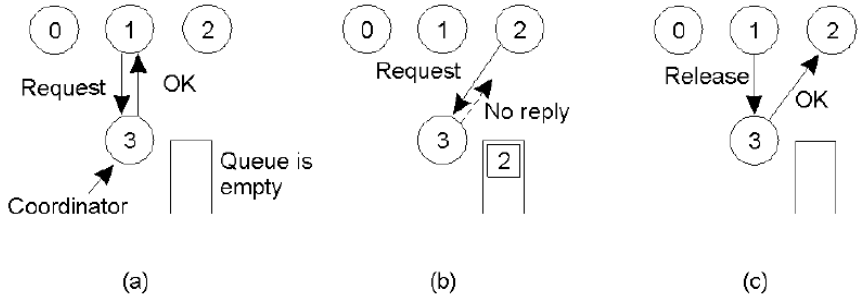
As Soluções baseadas em permissão o processo que quiser acessar um recurso compartilhado em primeiro lugar terá que pedir permissão aos outros processos para utilizar o recurso.

Os algoritmos utilizados na Exclusão Mútua Distribuida são:
- Algoritmo Centralizado;
- Algoritmo Distribuido;
- Algoritmo Token Ring

**ALGORITMO CENTRALIZADO**

No algoritmo centralizado o processo coordenador entra na região crítica e envia uma mensagem. O coordenador dá permissão ou não. Ele utiliza uma fila de espera. Ao sair da região crítica: envia mensagem.

Um processo é eleito como o coordenador. Sempre que um processo quiser acessar um recurso compartilhado, envia uma mensagem de requisição ao coordenador declarando qual recurso quer acessar e solicitando permissão. Se nenhum outro processo estiver acessando aquele recurso naquele momento, o coordenador devolve uma resposta concedendo a permissão.




a) Processo 1 pede permissão ao coordenador para entrar na SC. Permissão garantida
b) Processo 2 pede permissão para entrar na SC. Coordenador não responde.
c) Quando processo 1 sai da SC, ele avisa o coordenador, que responde ao 2

**Vantagens:**
- É justo;
- Não tem esfomeação;
- tem poucas mensagens.

**Desvantagens:**
- O coordenador é um ponto de falha único, portanto, se ele falhar, todo o sistema pode cair;
- Se os processos normalmente bloquearem após emitir uma requisição, não podem distinguir um coordenador inativo de uma permissão negada, visto que, em ambos os casos, nenhuma mensagem volta.

**ALGORITMO DISTRIBUIDO**

O Alogoritmo Distribuido requer que haja uma ordenação total de todos os eventos no sistema. Isto é, para qualquer par de eventos, como mensagens não pode haver ambiguidade sobre qual realmente aconteceu em primeiro lugar. O algoritmo de Lamport é um modo de conseguir essa ordenação e pode ser usado para fornecer marcas de tempo para exclusão mutua distribuída.

Quando um processo quer acessar um recurso compartilhado, monta uma mensagem que contem o nome do recurso, seu numero de processo e a hora corrente. Depois, envia a mensagem a todos os outros processos (inclusive para ele mesmo). Adota-se como premissa que o envio de mensagem é confiável, ou seja, nenhuma mensagem se perde. Quando um processo recebe uma mensagem de requisição de outro processo, a ação que ele executa depende de seu próprio estado em relação ao recurso nomeado na mensagem. Três casos têm de ser claramente distinguidos:

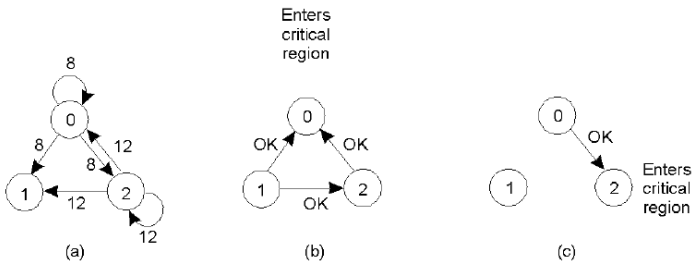
1) Se o receptor não estiver acessando o recurso e não quiser acessa-lo, devolve uma mensagem OK ao rementente.

2) Se o recepetor já tiver acessado o recurso, simplesmente não responde. E coloca a requisição em uma fila.

3) Se o receptor também quiser acessar o recurso, mas ainda não o fez, ele compara a marca de tempo da mensagem que chegou com a marca de tempo contida na mensagem que enviou para todos. A mais baixa vence. Se a marca de tempo da mensagem que acabou de chegar for mais baixa, o recepator devolve uma mensagem OK. Se a marca de tempo de sua própria mensagem for mais baixa, o receptor enfileira a requisição que está chegando e nada envia.

Após enviar requisições que pecam permissão, um processo se detem e espera ate que todos tenham dado permissão. Logo que todas as permissões tenham entrado, ele pode seguir adiante. Quando conclui, envia mensagens OK para todos os processos que estão em sua fila e remove todos eles da fila.

Esse algoritmo é mais lento, mais complicado, mais caro e menos robusto que o Algoritmo Centralizado.



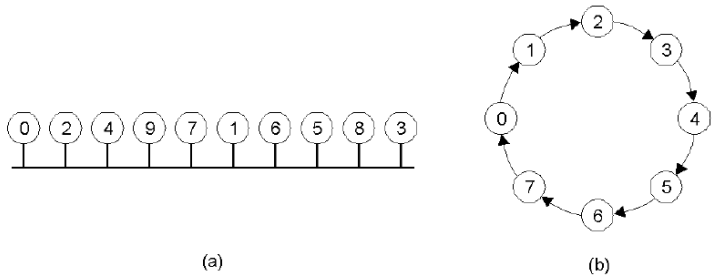

a) Dois processos desejam entrar na RC ao mesmo tempo.
b) Processo 0 tem o *time-stamp* menor, e ganha.
c) Quando processo 0 termina, ele envia OK, 2 pode entrar na RC.

Processo que recebe a mensagem de broadcast
Se não está na RC e não quer entrar, envia um ACK-OK
Se já está na RC, não responde e enfila a mensagem.
Se não está na RC, mas quer entrar, compara o carimbo de tempo.
Se o carimbo de tempo recebido for menor, envia um ACK-OK.
Se o seu carimbo de tempo for menor, não responde e enfila a msg.
Processo que enviou a mensagem espera até receber ACK-OK de todo grupo.

Todos os processos são envolvidos em todas as decisões de RC.

Problema: agora temos n pontos de falha.

**ALGORITMO TOKEN RING**




a) Um grupo de processos não ordenados em uma rede.
b) Um anel lógico construído em software.

Conforme mostra a figura temos uma rede de barramento sem nenhuma ordenação inerente dos processos. Um anel logico é construido em software e a cada processo é designada uma posição no anel.

As posições no anel podem ser alocadas em ordem numérica de endereços de rede ou por alguns outros meios. Não importa qual é a ordenação, o que importa é que cada processo saiba quem é a vez depois dele mesmo.

Quando o anel é inicializado, o processo 0 recebe uma ficha. A ficha circula ao redor do anel. Ele é passada do processo K para o processo K+1 em mensagens ponto-a-ponto.

Quando um processo adquire a ficha de seu vizinho, verifica para confirmar se precisa acessar o recurso compartilhado. Caso necessite, o processo se adiante, faz todo o trabalho que precisa fazer e libera o recurso. Após concluir, passa a ficha ao longo do anel. Não é permitido acessar o recurso novamente, de imediato, usando a mesma ficha. Se um processo receber a ficha de seu vizinho e não estiver interessado no recurso, ele apenas passa a ficha adiante.

**Vantagens**: Não ocorre inanição.

**Desvantagens**: A ficha pode se perder e precisará ser regenerada, mas é difícil detectar que ela se perdeu. Queda de um processo do anel

### 3) REPLICAÇÃO

A replicação de dados consiste na manutenção de múltiplas cópias da mesma informação em dispositivos distintos de um sistema distribuído. O objetivo é aumentar a confiabilidade (eficácia) e do desempenho dos serviços distribuídos. É um conceito de grande importância em Sistemas Distribuídos como Web/Internet, Sistemas de arquivos distribuídos e Banco de dados distribuídos.

Logo, há duas razões primarias para replicar dados: confiabilidade e desempenho. Quando os dados são replicados aumentão a **confiabilidade** do sistema. Por exemplo, se um sistema de arquivos foi replicado, pode ser possível continuar trabalhando após a queda de uma replica simplesmente com comutação para uma das outras replicas. Além disso, oferece melhor proteção contra dados corrompidos.

A replicação dos dados também ajuda a melhorar o **desempenho** do sistema. Por exemplo, um sistema distribuído precisa ser ampliado em quantidade e área geográfica. A ampliação em quantidade ocorre quando um numero cada vez maior de processos precisa acessar dados que são gerenciados por um único servidor. A ampliação de uma área geográfica ocorre quando se coloca uma copia dos dados próxima ao processo que os está usando, o tempo de acesso aso dados diminui. Por consequência o desempenho aumenta.

A replicação pode ter um impacto positivo ou negativo considerável no desempenho de uma aplicação distribuída. O impacto Positivo é o Caching que reduz o impacto negativo da latência na comunicação ao aproximar a informação do local onde ela é requerida; E o processamento concorrente de pedidos por vários servidores em paralelo (leitura de vários dados). O impacto Negativo é que muitas réplicas para gerenciar pode causar lentidão.

Um serviço estará sempre disponível enquanto for possível aos clientes se dirigirem a um servidor alternativo sempre que o servidor habitual falhar. A replicação também é útil para melhorar a disponibilidade de um sistema. Aumenta a proporção de tempo em que o sistema opera corretamente. A replicação é uma forma de contrariar os efeitos dos tipos de falhas que reduzem a disponibilidade de um sistema como FALHA DE SERVIDORES e DESCONEXÃO DA REDE.

| | Probabilidade de Falha | | |
|---|---|---|---|
| #réplicas | 25% | 10% | 5% |
| 1 | 75.000% | 90.000% | 95.000% |
| 2 | 93.750% | 99.000% | 99.750% |
| 3 | 98.438% | 99.900% | 99.988% |
| 4 | 99.609% | 99.990% | 99.999% |
| 5 | 99.902% | 99.999% | 100.000% |
| 10 | 100.000% | 100.000% | 100.000% |

disponibilidade

A disponibilidade dependerá do número de réplicas e da probabilidade de falha de cada réplica. Se os dados de que depende um serviço forem distribuídos por vários servidores com probabilidade iguais de falhar p. A probabilidade de todos n falharem = pn e a Disponibilidade = 1 – pn.

A replicação é uma forma básica de introduzir tolerância a falhas num sistema distribuído, incluindo falhas arbitrárias (ex: sabotagem), pois existindo múltiplas réplicas da mesma informação não bastará comprometer um servidor.

**RAZÕES PARA REPLICAÇÃO**

- Confiabilidade: Menos Falhas (se tem mais de uma "opção" disponível) e proteção contra dados corrompidos;
- Desempenho: Escalabilidade (mais servidores para responder as requisições); Caching (colocar cópia de dados próximo aos processos); Ampliação em quantidade para dividir trabalho de servidor centralizado, diminuindo o esforço para cada servidor e Ampliação geográfica para diminuir tempo de acesso a dados, aumentando o desempenho dos clientes.

**PROBLEMA DA REPLICAÇÃO DE DADOS**

O problema da replicação é que ter múltiplas copias pode levar a **problemas de consistência**, pois sempre que uma copia é modificada se torna diferente das restantes e é preciso modificar todas as outras copias para garantir consistência. Quando os dados replicados estão sujeitos a modificações é necessário assegurar que as replicas permaneçam consistentes, o que pode incorrer em um alto custo que afetará negativamente o desempenho do sistema.

## 4) CONSISTÊNCIA

Um conjunto de copias é consistente quando todas as copia são sempre iguais. Isso significa que uma operação de leitura realizada em qualquer copia sempre retornara o mesmo resultado. Em consequência, quando uma operação de atualização é realizada sobre uma copia, a atualização deve ser propagada para todas as copias antes que ocorre uma operação subsequente, sem importar em qual copia essa operação foi iniciada e realizada. Um exemplo de implementação de Consistência é o HW RAID.

A questão é de quanto em quanto tempo devemos atualizar o estado, pois o Overhead (custo computacional e de comunicação) para manter a coerência pode superar os beneficios de escalabilidade. Por exemplos, páginas Web. Em muitos casos, a única solução real para manter todas as copias iguais é relaxar as restrições de consistência.

Na ausência de um relógio global, é difícil definir com precisão qual operação de escrita é a ultima. Como alternativa, precisamos fornecedor outras definições, o que resulta em um conjunto de modelos de consistência. Cada modelo restringe efetivamente os valores que uma operação de leitura sobre um item de dados pode retornar. Os modelos que tem grandes restrições são fáceis de usar os que têm pequenas restrições são difíceis de usar. No entanto, os modelos fáceis de usar não funcionam tao bem quanto os difíceis de usar. Os Modelos de Consistencia utilizados são:

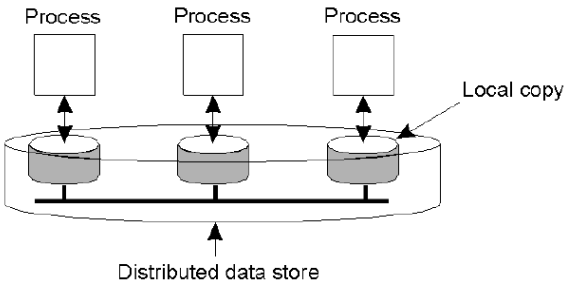
**Modelos de Consistencia Centrados em Dados**

- Consistencia Continua
- Consistencia Sequencial
- Consistencia Causal

**Modelos de Consistencia Centrados no Cliente**

- Consistencia Eventual

**MODELO DE CONSISTÊNCIA CENTRADO EM DADOS**




- Modelo geral de um data store distribuído / replicado.
- Local para alguns processos, remoto para outros.
- Cara processo pode ter sua cópia local (cache, proxy, etc.)

A consistência tem sido discutida no contexto de operações de leitura e escrita em dados compartilhados por meio de: Memória Compartilhada, Banco de Dados Compartilhado e Sistema de Arquivos Compartilhado. Os termos listados anteriormente serão chamados indistintamente de **Depósito De Dados.**

Conforme mostra a Figura ao lado, operações de escrita são propagadas para outras copias. Uma operação de dados é classificada como uma operação de escrita quando altera os dados, caso contrário é classificada como uma operação de leitura.

Cada processo que pode acessar dados do depósito tem uma cópia local (ou próxima) disponível do depósito completo. Operações de escrita são propagadas para outras Cópias.

Um **Modelo de Consistência** é um contrato entre processos e depósito de dados que firma que o depósito funcionará de maneira correta se os processos concordarem em obedecer a certas regras.

**CONSISTÊNCIA CONTÍNUA**

Não existe a melhor solução para replicar dados. A replicação de dados propõe problemas de consitencia que não podem ser resolvidos com eficiência de modo geral. Somente se abrandamos a consistência é que podemos ter esperança de conseguir soluções eficientes. Infelizmente, também não há regras gerais para abrandar a consistência: o que, exatamente, pode ser tolerado depende muito das aplicações.

Há modos diferentes para as aplicações especificarem quais inconsitencias elas podem tolerar. Existe uma abordagem geral que distingue três eixos independentes para definir inconsistências: **desvio em valores numéricos entre replicas, desvio em idade entre replicas e desvio em relação a ordenação de operações de atualização**. Esses desvios formam as faixas da **Consistencia Continua**.

A medicao da inconsistência em termos de **desvios numéricos** pode ser utilizada por aplicações para as quais dados têm semântica numérica. Por exemplo, a replicação de registros que contêm preços do mercado de ações.

O **desvio numérico** também pode ser entendido em termos de numero de atualizações que foram aplicadas a determinada replica, mas que ainda não foram vistas pelas outras. Por exemplo, uma cache web pode não ter visto as atualizações realizadas em um servidor web.

Os devios de idade estão relacionados com a última vez que uma replica foi atualizada. Por exemplo, previsões do tempo em geral.

Por fim, há classes de aplicações nas quais é permitido que a **ordenação das atualizações** seja diferente nas varias replicas, contanto que as diferenças fiquem dentro de um limite.

**CONIT (Consistency Unit)**

Para definir inconsistências foi proposta uma unidade de consistência, abreviada para CONIT. Uma Conit especifica a unidade segundo qual a consitencia deve ser medida. Monitoração de CONITs mede o quão distante uma réplica é em relação à outra.

Embora do ponto de vista de conceito as conits formem um modo atraente de capturar requisitos de consistência, há duas questões importantes que precisam ser tratadas antes que elas possam ser colocadas em uso na pratica. A primeira questão é que, para impor consistência, precisamos ter protocolos. A segunda questão é que os desenvolvedores de programas devem especificar os requisitos de consistências para suas aplicações.

Consistência continua pode ser implementada como um conjunto de ferramentas que, para os programadores, parece apenas uma outra biblioteca que eles integram as suas aplicações. Um conit é simplesmente declarada junto com uma atualização de um item de dados.

Granularidade de Conit: Necessário compromisso para manter Conits de granularidade grossa e Conits de granularid ade fina. Se uma conit representar uma grande quantidade de dados, tal como um banco de dados completo, as atualizações são agregadas para todos os dados na conit. Em decorrência, isso pode levar as replicas a entrar mais cedo em um estado inconsistente.

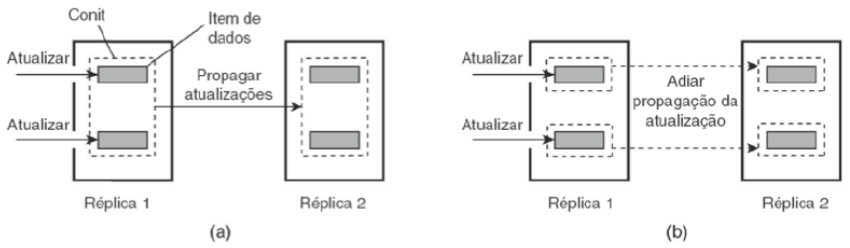



Figura 7.3 Escolha da granularidade adequada para uma conit. (a) Duas atualizações resultam em propagação da atualização. (b) Nenhuma propagação de atualização é necessária (ainda).

Por exemplo, a diferença entre duas réplicas não pode ter mais que uma atualização pendente. Nesse caso, quando cada um dos itens de dado mostrado na Figura 7.3(a) tiver sido atualizado uma ves na primeira replica, a segunda também precisara ser atualizada. Isso não acontece quando escolhermos uma conit menor, como mostra a Figura 7.3(b). Nesse caso, as replicas ainda são consideradas como atualizadas. Esse problema é de particular importância quando os itens de dados contidos em uma conit são usados com total independência; então, diz-se que eles compartilham falsamente a conit.

**ORDENAÇÃO CONSISTENTE DE OPERAÇÕES**

Os Modelos de programação concorrente que tratam de ordenar operações consistentemente em dados compartilhados replicados:

- Consistência Sequencial
- Consistência Causal

Ampliam os modelos de consistência contínua no sentido que, quando for preciso comprometer atualizações provisórias em réplicas, estas terão de chegar a um acordo sobre uma ordenação global dessas atualizações.

**CONSISTÊNCIA SEQUENCIAL**

A consistência sequencial é um modelo de consistência centrado em dados que foi definido por Lamport 1979. Rm geral diz que um deposito de dados é sequencialmente consistente quando satifaz a seguinte condição:

**O resultado de qualquer execução é o mesmo que seria se as operações (de leitura e escrita) realizadas por todos os processos no depósito de dados fossem executadas na mesma ordem sequencial e as operações de cada processo individual aparecessem nessa sequência na ordem especificada por seu programa.**

Essa definição significa que, quando processos executam concorrentemente em maquinas (possivelmente) diferentes, qualquer intercalação válida de operações é aceitável, mas todos os processos vêem a mesma intercalação de operações. Nada é dito sobre o tempo. Nesse contexto o processo "vê" escritas de todos, mas apenas suas próprias leituras.

**Notação:**
• As operações de um processo são representadas ao longo de um eixo de tempo. Eixo de tempo na horizontal (tempo cresce da esquerda para a direita)
• Wi(x)a: Processo i escreve valor a em item de dados x
• Ri(x)b: Processo i lê valor b em item de dados x

**Exemplo:**
• P1 executa uma escrita para um item de dados x, modificando o seu valor para a. Esta operação é feita localmente e depois propagada para os outros processos.
• Mais tarde, P2 lê o valor NIL e, pouco tempo depois, lê a. (Existe um retardo para propagar a atualização de x para P2).


Figura 7.4 Comportamento de dois processos que operam sobre o mesmo item de dados. O eixo horizontal representa o tempo.

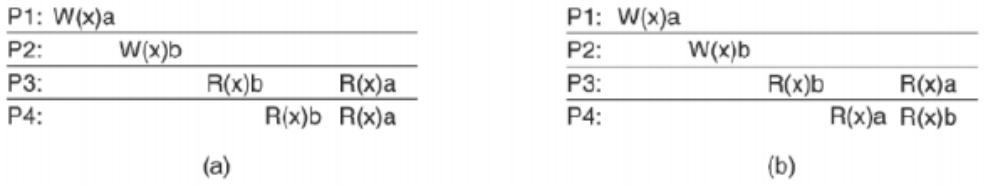



Figura 7.5 (a) Depósito de dados seqüencialmente consistente. (b) Depósito de dados que não é seqüencialmente consistente.

(a) $P_1$ executa W(x)a para ***x***. Mais tarde (tempo absoluto), o processo $P_2$ também executa uma operação de escrita, ajustando o valor de ***x*** para ***b***. Os processos $P_3$ e $P_4$ primeiro lêem o valor ***b*** e, mais tarde, o valor ***a***. A operação de escrita do processo $P_2$ parace ter ocorrido antes da de $P_1$

(b) Neste caso é violada a consistência sequencial porque nem todos os processos vêem a mesma intercalação de operações de escrita. Para o processo $P_3$ parece que o item de dados foi primeiro alterado para ***b***, e mais tarde para ***a***. Por outro lado, $P_4$ concluirá que o valor final é ***b***

| Processo P1 | Processo P2 | Processo P3 |
|---|---|---|
| x ← 1; | y ← 1; | z ← 1; |
| print(y, z); | print(x, z); | print(x, y); |

Figura 7.6 Três processos que executam concorrentemente.

Consideremos três processos que estejam executando concorrentemente $P_1$, $P_2$, $P_3$. Os itens de dados são ***x,y,z***. Cada variável é incializada com 0. Uma atribuição corresponde a uma escrita, enquanto um print corresponde a uma operação de leitura de dois argumentos

Várias sequências são possíveis, mas somente um subconjunto é válido (90 sequências)

| | | | |
|---|---|---|---|
| x ← 1; | x ← 1; | y ← 1; | y ← 1; |
| print(y, z); | y ← 1; | z ← 1; | x ← 1; |
| y ← 1; | print(x, z); | print(x, y); | z ← 1; |
| print(x, z); | print(y, z); | print(x, z); | print(x, z); |
| z ← 1; | z ← 1; | x ← 1; | print(y, z); |
| print(x, y); | print(x, y); | print(y, z); | print(x, y); |
| Impressões: 001011 | Impressões: 101011 | Impressões: 010111 | Impressões: 111111 |
| Assinatura: 001011 | Assinatura: 101011 | Assinatura: 110101 | Assinatura: 111111 |
| (a) | (b) | (c) | (d) |

Figura 7.7 Quatro seqüências de execução válidas para os processos da Figura 7.6. O eixo vertical é o tempo.

Em (a) $P_1$, $P_2$, $P_3$ são executados em ordem. (b), (c), (d) demonstram intercalações diferentes, mas igualmente válidas.

O contrato entre processos e o depósito de dados é que os processos devem aceitar todos os resultados válidos como respostas adequadas e devem trabalhar corretamente se qualquer um deles ocorrer.

**CONSISTÊNCIA CAUSAL**

O modelo de Consistencia Causal representa um enfraquecimento da consistência sequencial no sentido que faz distinção entre eventos que são potencialmente relacionados por causalidade e os que não são. Se o evento b é causado ou influenciado por um evento anterior a, a causalidade requer que todos vejam primeiro a e, depois, b. Operacoes que não estão relacionadas por causalidade são denominadas concorrentes.

Para um depósito de dados ser considerado consistente por causalidade, é necessário que obedeça à seguinte condição:

**Escritas que são potencialmente relacionadas por causalidade devem ser vistas por todos os processos na mesma ordem. Escritas concorrentes podem ser vistas em ordem diferente em máquinas diferentes.**

| | | | | | |
|---|---|---|---|---|---|
| P1: | W(x)a | | | W(x)c | |
| P2: | | R(x)a | W(x)b | | |
| P3: | | R(x)a | | R(x)c | R(x)b |
| P4: | | R(x)a | | R(x)b | R(x)c |

Figura 7.8 Essa seqüência é permitida quando o depósito é consistente por causalidade, mas não quando o depósito é seqüencialmente consistente.

Neste exemplo temos uma sequência de eventos permitida quando o depósito é consistente por causalidade, mas proibida quando o depósito é sequencialmente consistente. As escritas $W_2(x)b$ e $W_1(x)c$ são concorrentes, não sendo exigido que todos os processos as vejam na mesma ordem

| | | | | |
|---|---|---|---|---|
| P1: | W(x)a | | | |
| P2: | | R(x)a | W(x)b | |
| P3: | | | R(x)b | R(x)a |
| P4: | | | R(x)a | R(x)b |

(a)

| | | | |
|---|---|---|---|
| P1: | W(x)a | | |
| P2: | | W(x)b | |
| P3: | | R(x)b | R(x)a |
| P4: | | R(x)a | R(x)b |

(b)

Figura 7.9 (a) Violação de um depósito consistente por causalidade. (b) Seqüência correta de eventos em um depósito consistente por causalidade.

(a) Temos $W_2(x)b$ potencialmente dependente de $W_1(x)a$ porque *b* pode ser resultado de um cálculo ue envolva o valor lido por $R_2(x)a$. As duas escritas são relacionadas por causalidade, portanto temos uma violação na ordenação das operações

(b )Como a leitura R(x)a foi removida, $W_2(x)b$ e $W_1(x)a$ agora são escritas concorrentes. Um depósito consistente por causalidade não requer que escritas concorrentes sejam ordenadas globalmente.

Portanto, implementar consistência causal requer monitorar quais processos viram quais escritas, ou seja, significa que é preciso contruir e manter um gráfico de dependência que mostre qual operação é dependente de quais outras operações. Uma maneira de fazer isso é por meio de marcas de tempo vetoriais.

**CONSISTÊNCIA versus COERÊNCIA**

Um modelo de consistência descreve o que pode ser esperado com relação aquele conjunto quando vários processos operam concorrentemente sobre aqueles dados. Então, diz-se que o conjunto é consistente se ele aderir as regras descritas pelo modelo.

Onde a consistência de dados se refere a um conjunto de itens de dados, modelos de coerência descrevem o que pode ser esperado para só um item de dados. Nesse caso, consideramos que um item de dados é replicado em diversos lugares; diz-se que ele é coerente quando as varias copias aderem as regras como definidas por seu modelo de coerência associado.

**MODELO DE CONSISTÊNCIA CENTRADOS NO CLIENTE**

A capacidade de manipular operações concorrentes sobre dados compartilhados e, ao mesmo tempo, manter a consistência sequencial é fundamental para sistemas distribuídos. Os depósitos de dados que focalizaremos são caracterizados pela ausência de atualizações simultâneas ou, quando tais atualizações acontecem, elas podem ser resolvidas com facilidade. Esses depósitos de dados oferecem um modelo de consistência eventual.

Evitar consistência no sistema todo, concentrando em o que os clientes querem, e não o que deveria ser mantido pelos servidores.

Os Sistemas Distribuidos de grande escala (banco de dados) aplicam replicação para escalabilidade, mas podem suportar apenas consistência fraca, são eles:

- DNS → mudanças são propagadas devagar, inserções podem não ser imediatamente visíveis.
- NEWS → artigos e reações são colocadas e puxadas através da internet, de forma que reações podem ser vistas antes.
- Lotus Notes → servidores geograficamente dispersos replicam documentos, mas não fazem tentativa de manter modificações (concorrentes) mutuamente consistentes.
- WWW → Caches estão em todo lugar, mas não tem precisa garantia que está lendo a mais recente versão da página.

**CONSISTÊNCIA EVENTUAL (CONSISTÊNCIA PARA USUÁRIOS MÓVEIS)**

Banco de dados (de grande escala) distribuídos e replicados que toleram um grau relativamente alto de inconsistência, por exemplo, DNS e WWW. Esses bancos de dados têm em comum é que, se, nenhuma atualização ocorrer por tempo bastante longo, todas as replicas ficarão gradativamente consistentes. Essa forma de consistência é denominada **consistência eventual**.

Assim, depósitos de dados de consistência eventual têm a seguinte propriedade: na ausência de atualizações, todas as replicas convergem em direção a copias idênticas umas as outras. Em essência, consistência eventual exige apenas a garantia de que as atualizações serão propagadas para todas as replicas. De modo geral, conflitos escrita-escrita são relativamente fáceis de resolver quando consideramos que somente um pequeno grupo de processos pode realizar atualizações.

Depósitos de dados de consistência eventual funcionam bem, contanto que os clientes sempre acessem a mesma replica. Contudo, surgem problemas quando são acessadas replicas diferentes em um curto período.

Por exemplo, considere uma base de dados distribuída (conforme mostra a Figura ao lado) na qual você tem acesso através de seu notebook. Assumir que o notebook age como um front end para a base de dados. Na localização A acesso a base fazendo leituras e escritas.

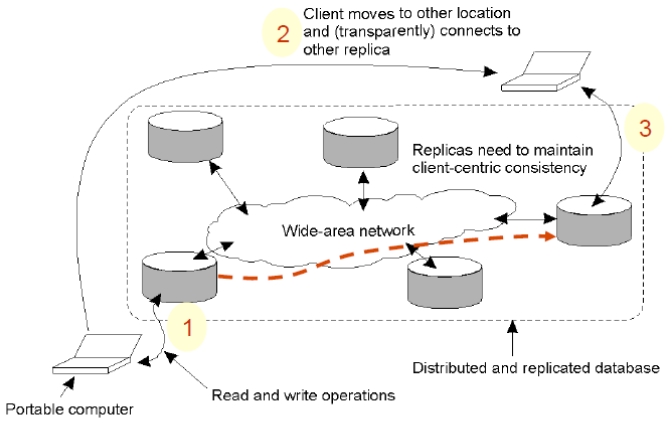
Na localização B continua seu trabalho, mas a não ser que você acesse o mesmo servidor acessado na localização A, você pode detectar inconsistências como:

• Suas atualizações em A podem não terem sido propagadas ainda para B;

• Você pode estar lendo novas entradas daquelas disponíveis em A

• Suas modificações em B podem eventualmente estar em conflito com aquelas em A.

Nota: A única coisa que você quer é que as entradas que você atualizou e/ou leu em A, estão em B da forma como você deixou em A. Assim, a base parecerá consistente para você.




O princípio de um usuário móvel acessando diferentes réplicas de uma base de dados distribuída.

Esse exemplo é típico de depósitos de dados de consistência eventual e é causado pelo fato de que, as vezes, os usuários podem operar sobre replicas diferentes. O problema pode ser amenizado com a introdução de **consistência centrada no cliente** que dá a um único cliente uma garantia de consistência de acesso a um deposito de dados por esse cliente; não há nenhuma garantia para acessos concorrentes por clientes diferentes.

Modelo de consistência centrado no cliente tem como premissa que a conectividade de rede é não confiável e sujeita a vários problemas de desempenho. Existem **quatro modelos de consistência centrados no cliente** que são:

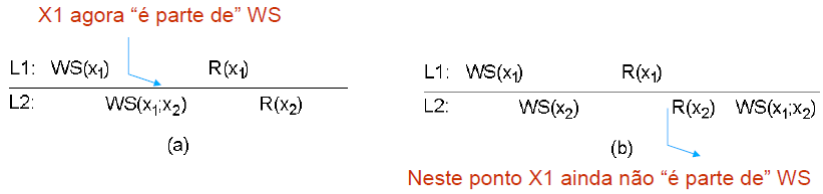
- **Reads Monotônicos (Leituras Monotonicas)**
- **Writes Monotônicos (Escritas Monotonicas)**
- **Ler Escritas (Leia-suas-escritas)**
- **Escrita-segue-Leitura (Escritas-seguem-leituras)**

**READS MONOTÔNICOS**

O primeiro modelo de consistência centrado no cliente é o de leituras monotonicas. Diz-se que um deposito de dados oferece consistência de leitura monotonica se a seguinte condição for cumprida:

**Se um processo ler o valor de um item de dados X, qualquer operação sucessiva de leitura em X executada por esse processo sempre retornará o mesmo valor ou um valor mais recente (não lê valores antigos).**

Exemplo de Reads Monotonicos é um banco de dados distribuído de email.




As operações read realizadas por um processo *P* em 2 cópias diferentes do mesmo data store (L1, L2).
a) Um armazém read monotônico consistente
b) Um armazém que não fornece read monotônico.

Na Figura acima (a) mostra o processo P realizando primeiro uma operação de leitura em X em L1, retornando o valor de x1 (naquele instante). Esse valor resulta das operações de escrita em WS(x1) realizadas em L1. Mais tarde, P realiza um operação de leitura em x em L2. Para garantir **consistência de leitura monotonica**, todas as operações em WS(x1) deveriam ter sido propagadas para L2 antes de ocorrer a segunda operação de leitura. Precisamos saber com certeza que WS(x1) é parte de WS(x2) o que é expresso por WS(x1;x2).

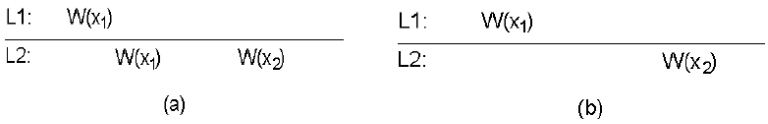
Na Figura (b) mostra a situação na qual a consistência de leitura monotonica não é garantida. Depois de ler x1 em L1, o processo P realiza a operação R(x2) em L2. Contudo, somente as operações de escrita em WS(x2) foram realizadas em L2. Não há nenhuma garantia de que esse conjunto também contem todas as operações contidas em WS(x1).

**WRITES MONOTÔNICOS**

Em muitas situações, é importante que operações de escrita sejam propagadas na ordem correta para todas as copias do deposito de dados. Essa propriedade é expressa em **consistência de escrita monotonica**. Em seu deposito vale a seguinte condição:

**Uma operação de escrita executada por um processo em um item de dados x é concluída antes de qualquer operação de escrita sucessiva em x pelo mesmo processo.**

Exemplo de Writes Monotonicos é uma biblioteca de software.




As operações write realizadas por um processo *P* em 2 cópias diferentes do mesmo data store (L1, L2)..
a) Um armazém write monotônico consistente
b) Um armazém que não fornece write monotônico

Na Figura acima em (a) o processo P realiza umaa operação de escrita em x na copia local L1. Mais tarde, P executa outra escrita em L2. Para garantir consistência de escrita monotonica, é necessário que a operação de escrita anterior em L1 já tenha sido propagada para L2.

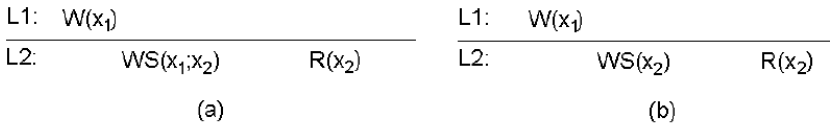
Na Figura acima em (b) está faltando a propagação de W(x1) para copia L2. Em outras palavras, não é garantida a consistência de escrita monotonica, pois não é possível garantir que a copia de x na qual a segunda escrita está sendo realizada tem o mesmo valor ou o valor mais recente no tempo W(x1) concluído em L1.

**LER ESCRITAS**

A seguir, apresentamos um modelo de consistência centrado no cliente que está intimamente relacionado com leituras monotnica. Diz-se que um deposito de dados fornece consistência leia-suas-escritas, se a seguinte condição for valida:

**O efeito de uma operação de escrita por um processo no item de dados x sempre será visto por uma operação de leitura sucessiva em x pelo mesmo processo**.

Exemplo de Ler-escritas é o browser Web.




A) Um armazém que fornece consistência read-your-writes.
B) Um armazém que não fornece.

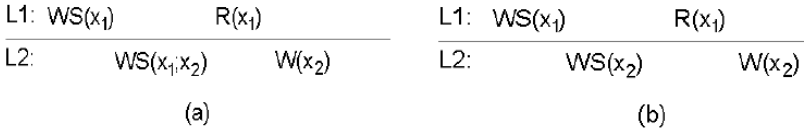
Na Figura acima em (a) o processo P realizou uma operação de escrita W(x1) e, mais tarde, uma operação de leitura em uma copia local diferente. A consistência leia-suas-escritas garante que os efeitos da operação de escrita podem ser vistos pela operação de leitura subsequente. Isso é expresso por WS(x1;x2), que declara que W(x1) é parte de WS(x2).

Na Figura (b) W(x1) foi deixada de fora de WS(x2), o que significa que os efeitos da operação de escrita pelo processo anterior P não foram propagados para L2.

**Escrita-segue-Leitura**

O ultimo modelo de consistência centrado no cliente é um modelo no qual as atualizações são propagadas como resultado de operações de leitura precedentes. Diz-se que um depositi de dados provê **consistência de escritas-seguem-leituras**, se a seguinte condição for válida:

**Uma operação de escrita por um processo em um item de dados x em seguinte a uma operação de leitura anterior em x pelo mesmo processo ocorre sobre o mesmo valor, ou sobre o valor mais recente de x que foi lido.**




a) Um armazém escrita-segue-leitura consistente.
b) Um armazém que não fornece consistência escrita-segue-leitura.

Na Figura acima em (a) um processo lê x na copia local L1. As operações de escrita que levaram ao valor que acabou de ser lido também aparecem no conjunto de escrita em L2, onde o mesmo processo realiza, mais tarde, uma operação d escrita.

Na Figura em (b) não é dada nenhuma garantia de que a operação em L2 é realizada sobre uma copia consistente com que acabou de ser lida em L1.

## 5) GERENCIAMENTO DE REPLICAS

Uma questão fundamental para qualquer sistema distribuído que suporta replicação é decidir onde, quando e por quem as replicas devem ser posicionadas e, na sequencia, quais mecanismos usar para manter as replicas consistentes. O problema do posicionamento em si deve ser subdividido em dois subproblemas: o de posicionar servidores de replicas e o de posicionar conteúdo.

O posicionamento de servidor de replicas refere-se a achar as melhores localizações para colocar um servidor que pode hospedar um deposito de dados (ou parte dele).

Posicionamento de conteúdo refere-se a achar os melhores servidores para colocar conteúdo.

## 6) PROTOCOLOS DE CONSISTÊNCIA

Um Protocolo de Consistência descreve uma implementação de um modelo específico de consistência. Os modelos mais importantes e aplicados são aqueles onde as operações são globalmente serializadas (seqüencial, fraca c/sincronização, transações). Enfatizamos apenas protocolos para consistência sequencial que são:

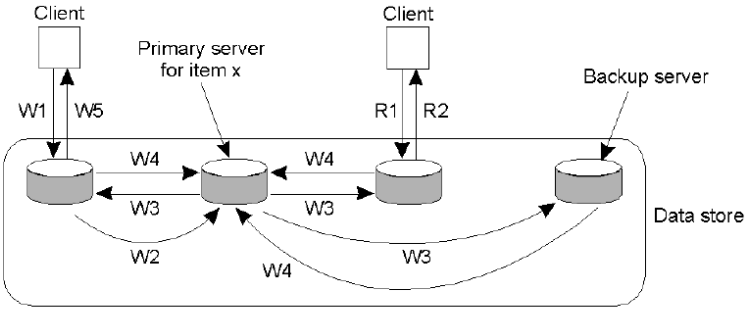
- **Protocolo baseado em cópia primária**
- **Protocolo escrita remota**
- **Protocolo escrita local**
- **Protocolo escrita-replicada: Replicação ativa**
- **Protocolo de Votação**
- **Protocolo de coerência de cache**

**PROTOCOLO BASEADO EM CÓPIA PRIMÁRIA**

No caso de consistência sequencial prevalecem os protocolos baseados em primários. Nesses protocolos, cada item de dados x no deposito de dados tem um primário associado, que é responsável por coordenar operações de escrita em x.

**PROTOCOLO ESCRITA REMOTA**

O protocolo mais simples baseado em primário e que suporta replicação é aquele em que as operações de escrita precisam ser enviadas para um único servidor fixo. Operações de leitura podem ser executadas no local. Esses esquemas também são conhecidos como protocolos de primário e backup. Ele funciona como mostra a Figura abaixo:




W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

R1. Read request
R2. Response to read

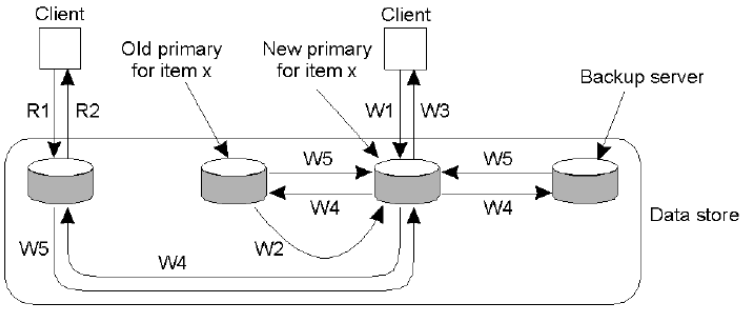
O protocolo baseado em primário c/ backup.

Um processo que quer realizar uma operação de escrita, em um item de dados, envia essa operação para o servidor de primários para x. O servidor primário executa a atualização em sua copia local de x e, na sequencia, envia a atualização para os servidores de backup. Cada servidor de backup também efetua a atualização e envia um reconhecimento de volta ao servidor primário. Quando todos os servidores de backup tiverem atualizado sua copia local, o servidor primário envia um reconhecimento de volta ao processo inicial.

Um problema potencial de desempenho desse esquema é que pode levar um tempo relativamente longo antes que o processo que iniciou a atualização tenha permissão para continuar. Na verdade, uma atualização é implementada como uma operação de bloqueio. Uma alternativa é usar uma abordagem não bloqueadora. Tao logo o servidor primário tenha atualizado sua copia local de x, ele retorna um reconhecimento.

O principal problema de protocolos de primário backup não bloqueadores tem a ver com tolerância a falha. Com um sistema bloqueador, o processo cliente sabe, com certeza, que a operação de atualização é apoiada por vários outros servidores. Isso não ocorre com uma solução não bloqueadora.

**Protocolos Escrita-Local**




W1. Write request
W2. Move item x to new primary
W3. Acknowledge write completed
W4. Tell backups to update
W5. Acknowledge update

R1. Read request
R2. Response to read

Protocolo primário-backup onde o primário migra para o processo esperando para realizar a modificação.

Um variante dos protocolos de primário e backup é aquela em que a copia primaria migra entre processos que desejam realizar uma operação de escrita.

Sempre que um processo quer atualizar um item de dados x, ele localiza a copia primaria de x e, na sequencia, move essa copia para sua própria localização, como mostra a Figura ao lado.

A principal vantagem dessa abordagem é que múltiplas operações sucessivas de escrita podem ser executadas no local enquanto processos leitores ainda podem acessar sua copia local.

Contudo, so se pode conseguir tal melhoria se for seguido um protocolo não bloqueador pelo qual as atualizações são propagadas para as replicas após o servidor primário ter concluído as atualizações realizadas localmente.

**PROTOCOLOS ESCRITA-REPLICADA**

Em protocolos de escrita replicada, operações de escrita podem ser executadas em varias replicas em vez de em só uma, como no caso de replicas baseadas em primários. Pode-se fazer uma distinção entre replicação ativa, na qual uma operação é repassada para todas as replicas, e protocolos de consistência baseados em voto majoritário. Protocolos deste tipo são:

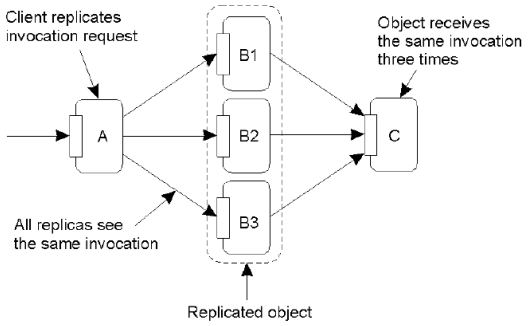
- Replicação Ativa → onde uma operação é repassada para todas as réplicas
- Votação da maioria → fazer com que os clientes requisitem e adquiram a permissão de múltiplos servidores antes de ler ou escrever um item de dados replicado

**REPLICAÇÃO ATIVA**

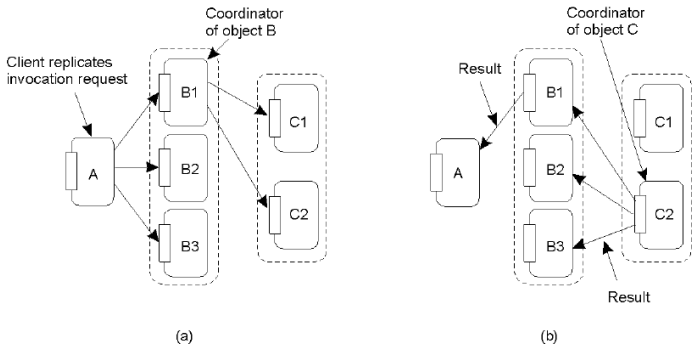
Em replicação ativa, cada replica tem um processo associado que realiza as operações de atualização. Ao contrario de outros protocolos, de modo geral, as atualizações são propagadas por meio da operação de escrita que causa a atualização.

Um problema da replicação ativa é que as operações precisam ser executadas na mesma ordem em todos os lugares. Por isso, é preciso de um mecanismo de multicast totalmente ordenado. Tal multicast pode ser implementado com o uso de relógios lógicos de Lamport.

Infelizmente essa implementação de multicast tona-se muito complexa em grandes sistemas distribuídos. Como alternativa, pode-se conseguir ordenação total usando um coordenador central, também denominado **sequenciador**. Contuto, ele não resolve o problema de escalabilidade.




O problema das invocações replicadas.




a) Repassando uma requisição de invocação de um objeto replicado para outro.
b) Retornando uma resposta de um objeto replicado para outro.

**Protocolos de Votação**

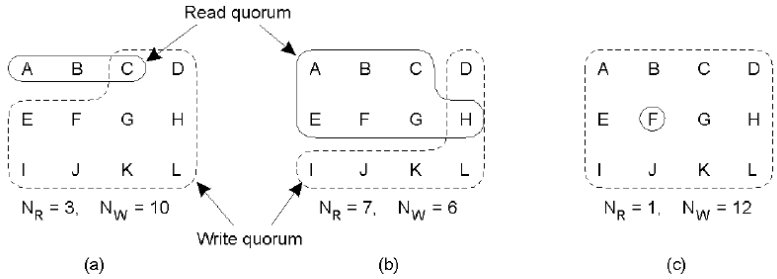
Uma abordagem diferente para suportar escritas replicadas é usar votação. A ideia básica é exigir que clientes requisitem e adquiram a permissão de vários servidores antes de ler ou escrever um item de dados replicado.

O esquema do protocolo de votação diz que para ler um arquivo do qual existem N replicas, um cliente precisa conseguir um quórum de leitura, um conjunto arbitario de quaisquer NR servidores, ou mais. De maneira semelhante, para modificar um arquivo, é exigido um quórum de escrita de, no mínimo, NW servidores. Os valores de NR e NW estão sujeitos as duas restrições seguintes:

**1. NR + NW*N**
Usada para evitar conflitos leitura-escrita.

**2. NW*N/2**
Usada para impedir conflitos escrita-escrita.

Para ver como esse algoritmo funciona, considere




Exemplos do algoritmo de votação :
a) uma escolha correta para o conjunto de leitura e escrita
b) uma escolha que pode levar a conflitos escrita-escrita
c) uma escolha correta, conhecida como ROWA (read one, write all)

Para ver como esse algoritmo funciona, considere a Figura acima, em (a) NR = 3 e NW = 10. Imagine que o quórum de escrita mais recente consistiu em 10 servidores, C a L. Todos eles obtem a nova versão e o novo numero de versão. Qualquer quórum de leitura subsequente de três servidores terá de conter, no mínimo, um membro desse conjunto. Quando o cliente vir os números de versão, saberá qual é a mais recente e a adotará.

Em (b) pode ocorrer conflito escrita-escrita porque NW<=N/2. Em particular, se um dos clientes escolher {A,B,C,E,F,G} como seu conjunto de escrita e um outro cliente escolher {D,H,I,J,K,L} como seu conjunto de escritas, então estaremos claramente em dificuldades porque ambas as atualizações serão aceitas sem detectar que, na verdadem estão em conflito.

Em (c) é de especial interesse porque fixa NR em um, o que possibilita ler um arquivo replicado descobrindo e usando qualquer copia.

**Protocolos de Coerencia de cache**

Caches são um caso especial de replicação, no sentido de que, em geral, são controladas por clientes, em vez que servidores. Contudo protocolos de coerência de cache que garantem que uma cache é consistente com as replicas iniciadas por servidor, em principio, não são muito diferentes dos protocolos de consistência.

Em primeiro lugar, as soluções de cache podem ser diferentes quanto a **estratégia de detecção de coerência**, isto é, **quando** as inconsistências são realmente detectadas. Em soluções estáticas, a premissa adotada é que um compilador realize a analise necessária anterior a execução e determine quais dados podem realmente levar a inconsistências porque podem ser colocados em cache. O compilador simplemente insere instruções que evitam inconsistências.

Nos sistemas distribuídos são aplicadas soluções dinâmicas. Nessa soluções, as inconsistências são detectadas em tempo de execução. Por exemplo, é feita uma verificação no servidor para ver se os dados em cache foram modificados desde que entraram na cache. No caso de banco de dados distribuídos, os protocolos baseados em detecção dinâmica ainda podem ser classificados considerando exatamente em que ponto de uma transação a detecção é feita. Distinguem-se em três casos seguintes.

No primeiro, quando um item de dados em cache é acessado durante um transação, o cliente precisa verificar se esse item de dados ainda é consistente com a versão armazenada no servidor (possivelmente replicado). A transação não pode prosseguir e usar a versão em cache até que sua consistência tenha sido definitivamente validada.

Na segunda abordagem, a otimista, a transação pode prosseguir enquanto ocorre a verificação. Nesse caso, a premissa adotada é que os dados em cache estavam atualizados quando a transação comecaou. Se, mais tarde, essa premissa mostrar ser faksam a transação terá de ser abortada.

A terceita abordagem é verificar se os dados em cache estão atualizados somente quando a transação for comprometida. Essa abordagem é comparável ao esquema otimista de controle de concorrência. Na verdade, a transação apenas inicia a operação nos dados em cache e espera que o melhor aconteca. Depois que todo o trabalho foi realizado, é verificada a consistência dos dados acessados. Quando forem usados dados antigos, a transação é abortada.

Uma outra questão de projeto para protocolos de coerência de cache é a **estratégia de imposição de coerência**, que determina **como** as caches são mantidas consistentes com as copias armazenadas em servidores. A solução mais simples é não permitir que dados compartilhados sejam colocados em cache. Em vez disso, dados compartilhados são guardados somente nos servidores, que mantem consistência usando um dos protocolos baseados em primários ou de replicação de escrita.

Quando dados compartilhados podem ser colocados em cache, há duas abordagens para impor coerência de cache. A primeira é permitir que um servidor envie uma invalidação a todas as caches sempre que um item de dado for modificado. A segunda é simplesmente propagar a atualização.