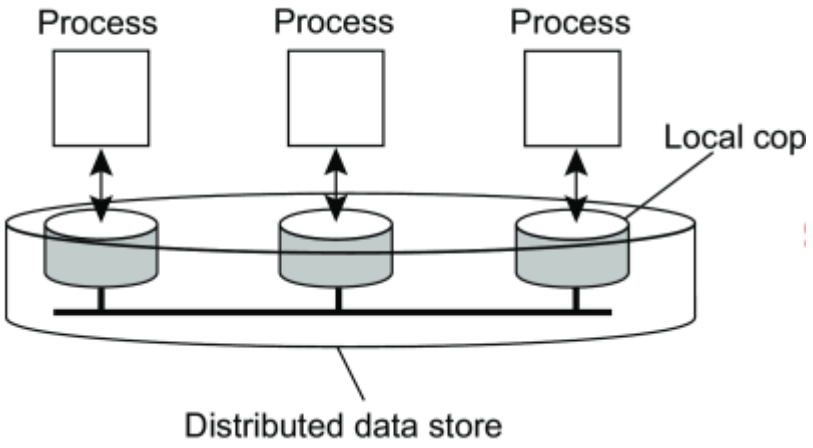
Capítulo 7: Consistência e Replicação

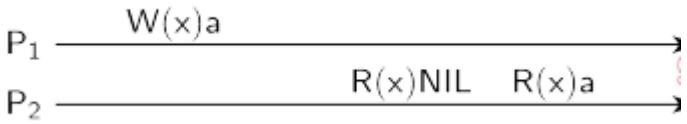
client-centric consistency models: Concentra na consistência na perspectiva de um único cliente

tight consistency provida por synchronous replication:Todas as cópias sobre uma operação de read são iguais, além disso, quando um dado é atualizado ele é propagado para todas as cópias antes de efetuar qualquer alteração. Um update é realizado sobre todas as cópias como uma única operação atômica ou transação, essas operações devem ser completadas rapidamente. As cópias devem chegar em um acordo sobre quando um update deve ser feito localmente.

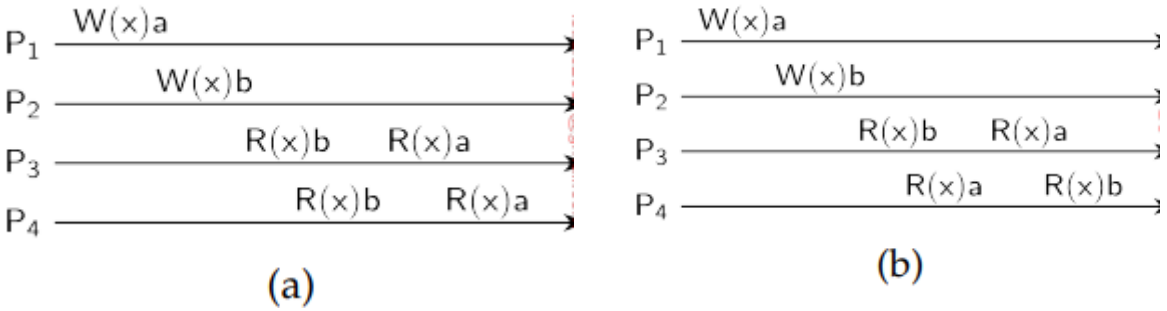
7.2 Date centric consistency model: Contrato entre processos e o “data store”. Os processos devem seguir certas regras e em troca o “data store” promete funcionar corretamente.



Sequential Consistency:



Definido por Lamport em 1979. Um “data store” é sequencialmente consistente quando satisfaz a condição de:
O Resultado de qualquer execução é a mesma se as operações de Read e Write, por todos os processos no “data store” forem executados em qualquer ordem sequencial, e as operações de cada indivíduo aparecem nesta sequência na ordem especificada pelo programa.



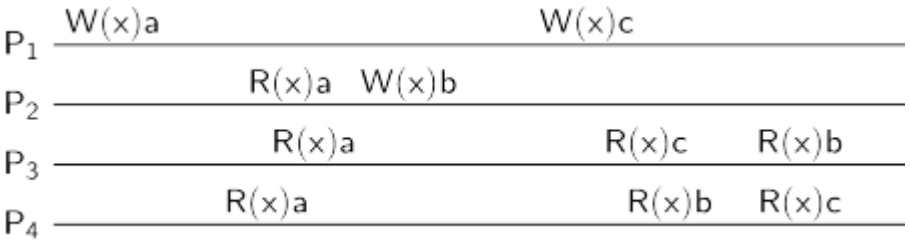
a) Sequencialmente Consistente
b) Não é sequencialmente consistente.

Causal Consistency: Hutto and Ahamad 1990.
Caso um processo P1 escreva no “data item” x. Então, P2 lê x e depois escreve y. Logo, a leitura de x e a escrita de y são “causally related”

Casos dois processos simultaneamente e espontaneamente escrevam 2 coisas diferentes em um mesmo “data item”. Então elas são uma operação concorrente “concurrent”

Para um “data store” ser causally consistent, devem ser seguidas as seguintes condições:

“escritas potencialmente “causally related” devem ser vistas por todos os processos na mesma ordem. Escritas concorrentes podem ser vistas em ordem diferentes em diferentes máquinas.”



Grouping Operations: Semelhante a SO, semáforos e travas. Região crítica e etc.

We now demand that the following criteria are met [Bershad et al., 1993]:

- Acquiring a lock can succeed only when all updates to its associated shared data have completed.
- Exclusive access to a lock can succeed only if no other process has exclusive or nonexclusive access to that lock.
- Nonexclusive access to a lock is allowed only if any previous exclusive access has been completed, including updates to the lock’s associated data.

Linear.

Entry consistency: Ligado ao **Grouping Operations** usa travas para manter a consistência.

Consistência != Coerência

Consistência: O que pode ser esperado de um “set” quando processos concorrentes operam sobre um dado. O “set” é consistente quando este segue as regras descritas pelo modelo.

Coerência: O que pode ser esperado de um único “data item”. Assumimos que que o “data item”. É coerente quando as várias cópias seguem as regras definidas pelo seu “consistency model”. No caso de escritas concorrentes, todos veem o mesmo resultado.

Eventual Consistency: Não há necessidade de atualização imediata da rede. Eventualmente, todas as réplicas vão convergir para somente uma réplica idêntica. Requer essencialmente que os updates sejam garantidos de serem propagados a todas as réplicas. Assumindo que somente um pequeno grupo de máquinas pode realizar operações de write, então, conflitos write-write podem ser tratados. Em caso desses conflitos, uma operação específica é dita vencedora e sobre escreverá os os efeitos da outra operação write conflitante.

Strong eventual consistency: Se há updates conflitantes, por consequência as réplicas de onde esses updates foram propagados estão no mesmo estado. (Shapiro (2011) **Conflict-Free Replicated Data Type (CRDT)**, um dado que pode ser replicado indefinidamente, mas pode ser atualizado com updates concorrentes sem necessidade de coordenação avançada.)

Segunda aproximação: **Program Consistency**

Continuous Consistency: Yu and Vahdat (2002). Definiram as **continuos consistency ranges**. Basicamente, flutuações e alterações entre dados replicados distribuídos que são toleradas pelo sistema, ou seja, o sistema é tolerante a inconsistências dentro dos padrões estabelecidos.
absolute numerical deviation: Duas cópias não podem se diferenciar mais que 0.02 R$
relative numerical deviation: Pode especificar que ambas as cópias podem se diferenciar em 0.5%

Para definir inconsistências: **consistency unit (conit)**, especifica as unidades que a consistência será medida, por exemplo, dados de uma única ação na bolsa de valores.

7.3 Client-centric consistency models: prover uma visão consistente de uma “data store”, necessário prover consistência no caso de processos concorrentes. Garante que para um único cliente, que não ocorra problemas de consistência, não há garantias para clientes diferentes.

Client-centric consistency models:

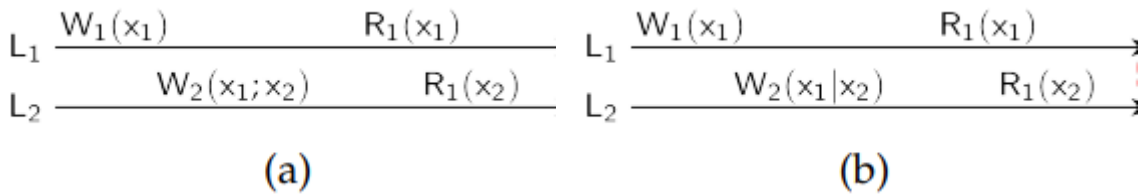
Monotonic reads: Um “data store” provê monotonic-read consistency se:

“Se um processo lê o valor de um dado item x, qualquer operação de leitura bem sucedida de x por tal processo sempre vai retornar o mesmo valor ou um mais recente.”

Garante que caso um processo veja um valor de x, ele nunca verá uma versão mais velha que x. Exemplo, acesso ao email em diferentes localizações, a caixa de entrada sempre terá as mensagens que foram acessadas na ultima localização, com a adição ou não de mensagens recentes.

“W2(x1; x2) indicates that process P2 is responsible for producing version x2 that follows from x1. Likewise, W2(x1|x2) denotes that process P2 producing version x2 concurrently to version x1 (and thus potentially introducing a write-write conflict).”

L : Local Data Store.



a) **monotonic-read consistent data store**
b) **não prove monotonic-read consistent**

Monotonic writes: Para ser um monotonic-write consistent store, deve-se:

“Uma operação de escrita por um processo em um dado item x é completado antes de qualquer operação de escrita suscetível em x no mesmo processo”

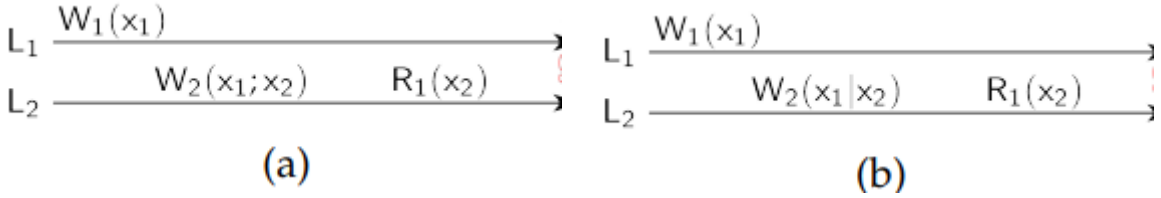
Uma operação de escrita de um item x é performada somente se aquela cópia foi revelada por outra operação de escrita precedente a ela, em um mesmo processo. Se necessário, uma nova escrita deve aguardar pelas escritas antigas terminarem.

Preza-se pela consistência em um único processo, semelhante ao modelo FIFO de ordenação. Por definição, operações de escrita realizadas por um mesmo processo são performadas na mesma ordem em que elas são inicializadas.

Read your Writes: Um data store provê read-your-writes consistency se:

“O efeito de uma operação de escrita de um processo em um dado item x sempre vai ser visto por uma operação de leitura suscetível sobre x no mesmo processo”

Ou seja, uma operação de escrita sempre será completada antes de uma operação de leitura em um mesmo processo, sem importância sobre a localização da operação de leitura.



a) **read-your-writes consistency**
b) **Não provê (as alterações da escrita W1(x1) foram perdidas pela escrita em concorrência de x2)**

Writes follow reads: Updates são propagados como resultado das operações de leituras passadas. Um data store prove writes follow reads quando:

“Uma operação de leitura por um processo em um dado item x, seguido de uma operação de leitura passada em x de um mesmo processo é garantido que retorne o mesmo ou um valor mais recente do x que foi lido”.

“Suponha que um usuário leia um artigo A, então esse usuário reage postando uma resposta B sobre o artigo A. Então, pelo writes-follow-reads, B vai ser escrito em qualquer cópia do grupo somente depois que A também for escrito.

7.4 Replica Management:

3 possibilidades:

Propagar somente uma notificação de update: (invalidation protocols) informa as cópias que um update ocorreu e que os dados que eles possuem não são válidos, somente a notificação é propagada.

Transferir dados de uma cópia a outra: útil quando a frequência read-to-write é alta. Nesse caso, a probabilidade de uma operação de leitura acontecer antes de uma operação de update é alta. Também é possível que ao invés de se propagar os dados modificados é possível documentar as mudanças e só enviar esses “logs” para salvar a largura de banda.

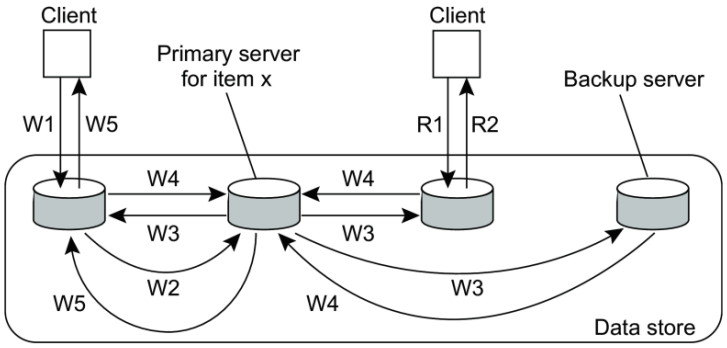
Propagar a operação de update para outras cópias: (active replication) . Essa abordagem não transfere quaisquer modificações, mas diz a cada réplica qual operação de update deve ser performada. Assume que cada réplica é um processo capaz de ativamente se mandar atualizado por meio de operações. Demanda poder de processamento, mas mínima largura de banda.

7.5 Consistency Protocols

1. **Protocolos Baseados em Primário (Primary-based Protocols)**:
    - **Descrição**: Um servidor primário é responsável por todas as atualizações de um dado antes de replicá-las para os backups.
    - **Cenário Ideal**: Útil em sistemas onde há mais leituras do que escritas, como bases de dados centralizadas ou sistemas de cache. Exemplo: sistemas de arquivos distribuídos que usam escrita local e sincronizam as atualizações posteriormente.

Primary-based protocols

Primary-backup protocol



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

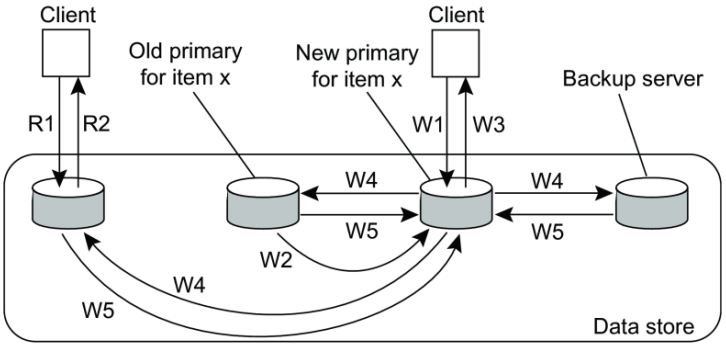
Example primary-backup protocol

Traditionally applied in distributed databases and file systems that require a high degree of fault tolerance. Replicas are often placed on the same LAN.

2. **Protocolos de Escrita Replicada (Replicated-write Protocols)**:
    - **Descrição**: Permite que várias réplicas aceitem operações de escrita simultaneamente.
    - **Subtipos**:
        - **Replicação Ativa**: Todas as réplicas processam operações na mesma ordem, exigindo multicast totalmente ordenado.
        - **Protocolos Baseados em Quórum**: Leituras e escritas são realizadas apenas após obter permissão de um número mínimo de servidores (quórum).
    - **Cenário Ideal**: Adequado para aplicações críticas que exigem alta disponibilidade e consistência, como bancos de dados distribuídos com forte tolerância a falhas.

Primary-based protocols

Primary-backup protocol with local writes



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

Example primary-backup protocol with local writes

Mobile computing in disconnected mode (ship all relevant files to user before disconnecting, and update later on).

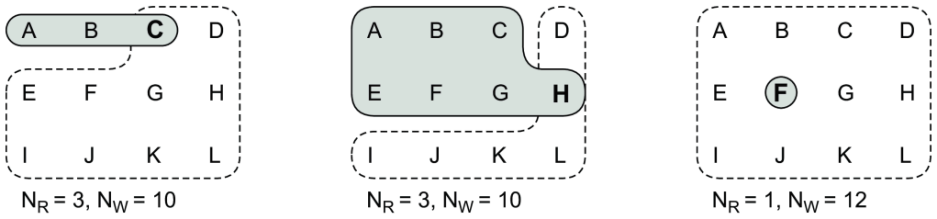
3. **Protocolos Baseados em Consistência Eventual (Eventual Consistency Protocols)**:
    - **Descrição**: As réplicas são atualizadas de forma assíncrona, garantindo consistência apenas eventualmente.
    - **Cenário Ideal**: Sistemas onde a baixa latência é crucial e a consistência forte pode ser relaxada, como redes de entrega de conteúdo (CDNs) e aplicativos de mídia social.

Replicated-write protocols

Quorum-based protocols

Assume N replicas. Ensure that each operation is carried out in such a way that a majority vote is established: distinguish read quorum N_R and write quorum N_W. Ensure:

1. $N_R + N_W > N$ (prevent read-write conflicts)
2. $N_W > N/2$ (prevent write-write conflicts)



4. **Protocolos de Consistência Contínua (Continuous Consistency)**:
    - **Descrição**: Permite ajustes no nível de consistência, como divergência de valores, idade dos dados ou ordem das operações.
    - **Cenário Ideal**: Aplicações com requisitos específicos de consistência, como análises financeiras ou controle de inventário.

Continuous consistency: Numerical errors

Principal operation

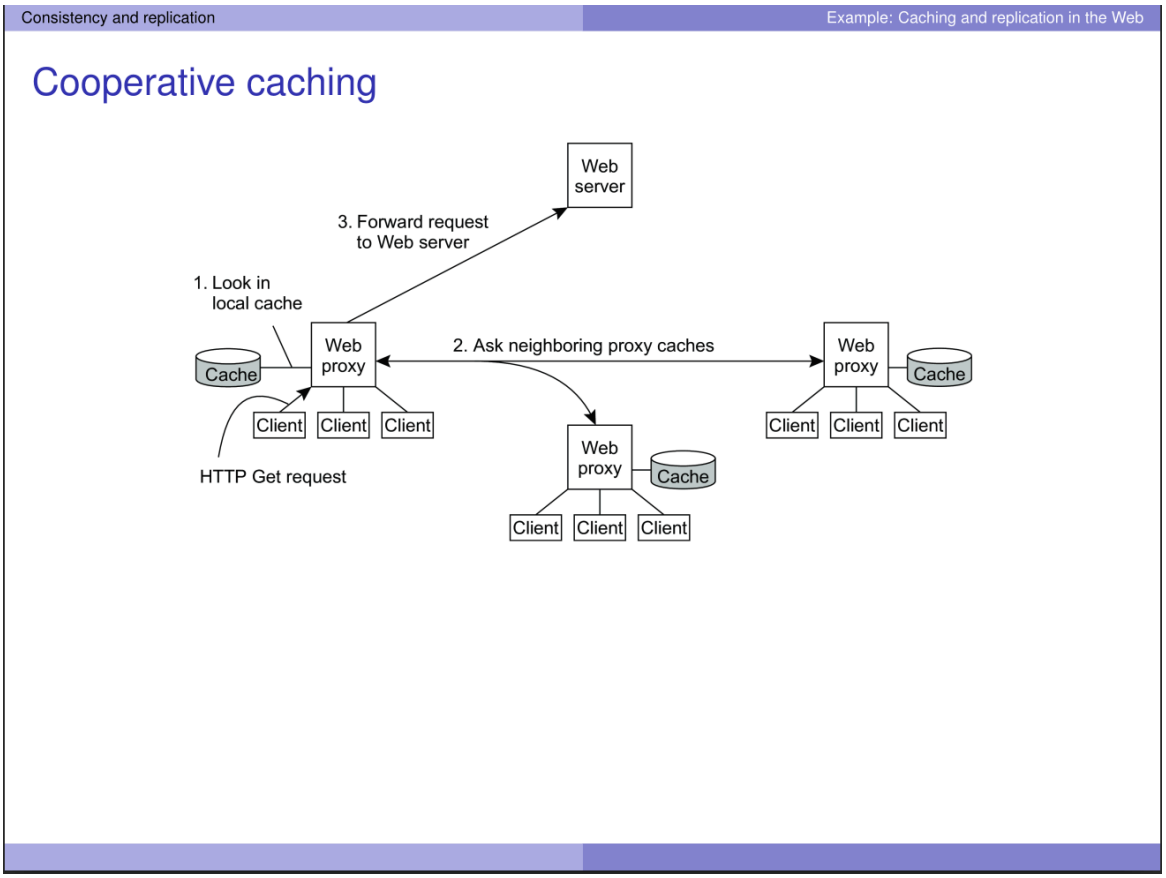
- Every server S_i has a log, denoted as L_i.
- Consider a data item x and let $val(W)$ denote the numerical change in its value after a write operation W. Assume that

$$\forall W : val(W) > 0$$

- W is initially forwarded to one of the N replicas, denoted as $origin(W)$. $TW[i,j]$ are the writes executed by server S_i that originated from S_j:

$$TW[i,j] = \sum\{val(W) | origin(W) = S_j \ \& \ W \in L_i\}$$

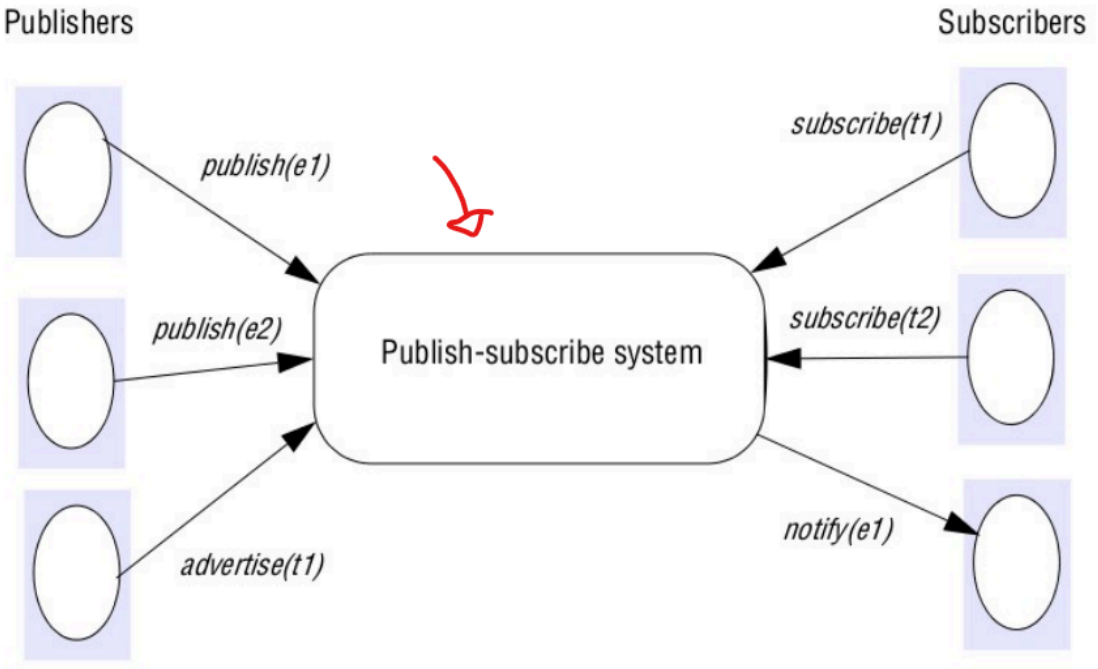
5. **Protocolos de Cache Iniciados pelo Cliente (Client-initiated Caches)**:
    - **Descrição**: As réplicas são criadas localmente pelo cliente para melhorar os tempos de acesso.
    - **Cenário Ideal**: Adequado para sistemas com muitas leituras locais e baixa modificação dos dados, como sistemas de armazenamento em nuvem locais.



Anotações em Geral:

Endpoint: Nada mais é que um serviço que aguarda uma requisição em uma porta.

Publish-subscribe system: “Brokers”



Heterogeneidade: Componentes que não foram projetados para interoperar, podem trabalhar juntos.

Assincronicidade: Notificações são enviadas assincronicamente pelos publicadores para todos os inscritos que expressaram interesse neles, prevenindo a necessidade de sincronização com os inscritos.
“a interação entre as partes não exige que o remetente espere pela resposta do destinatário para continuar sua execução.”

Desacoplado no Tempo: Quando não é necessário que 2 processos estejam ativos ao mesmo tempo, logo, estão desacoplados no tempo.

Desacoplado no Espaço: Se sabe o endereço do destinatário.

Assíncrono != Desacoplado.

Relógios Lógicos: Cada processo vai ter seu próprio relógio lógico,

Utilizando dos Relógios Lógicos de Lamport. Podemos garantir a ordenação de processos, no qual cada processo, ao enviar uma mensagem também enviará seu relógio lógico para que os processos que receberem essa mensagem possam se ajustar.

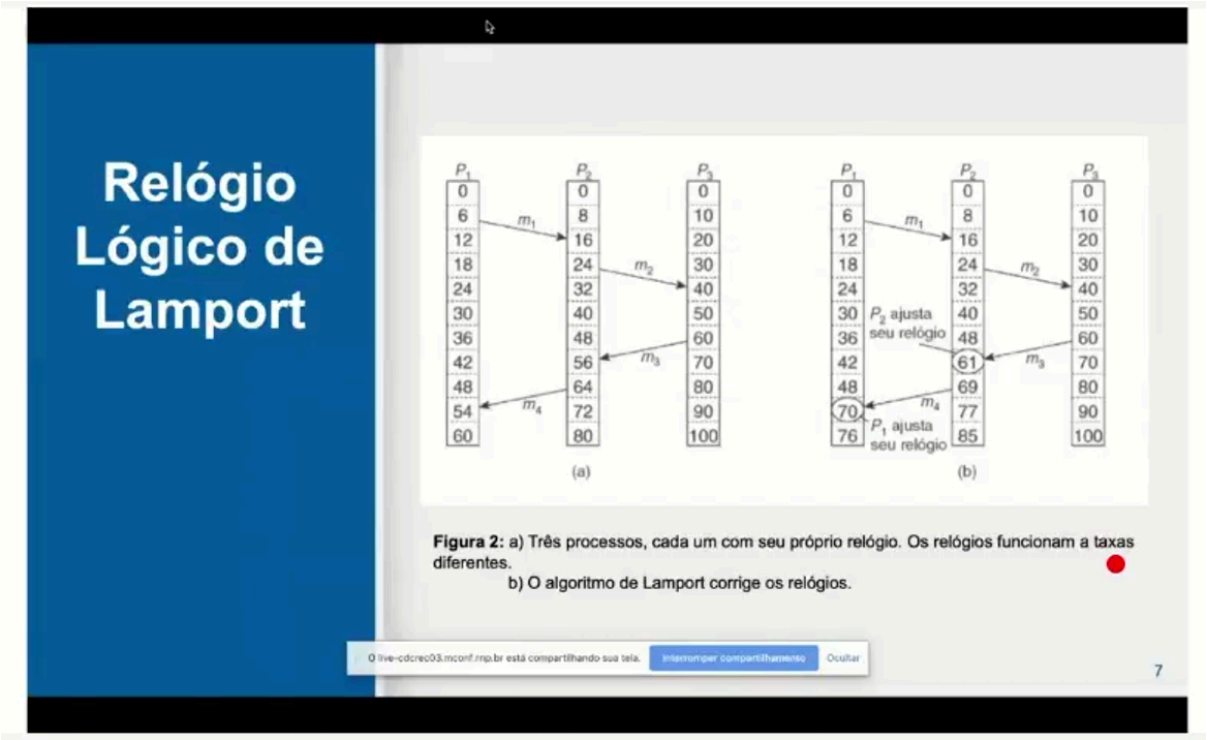
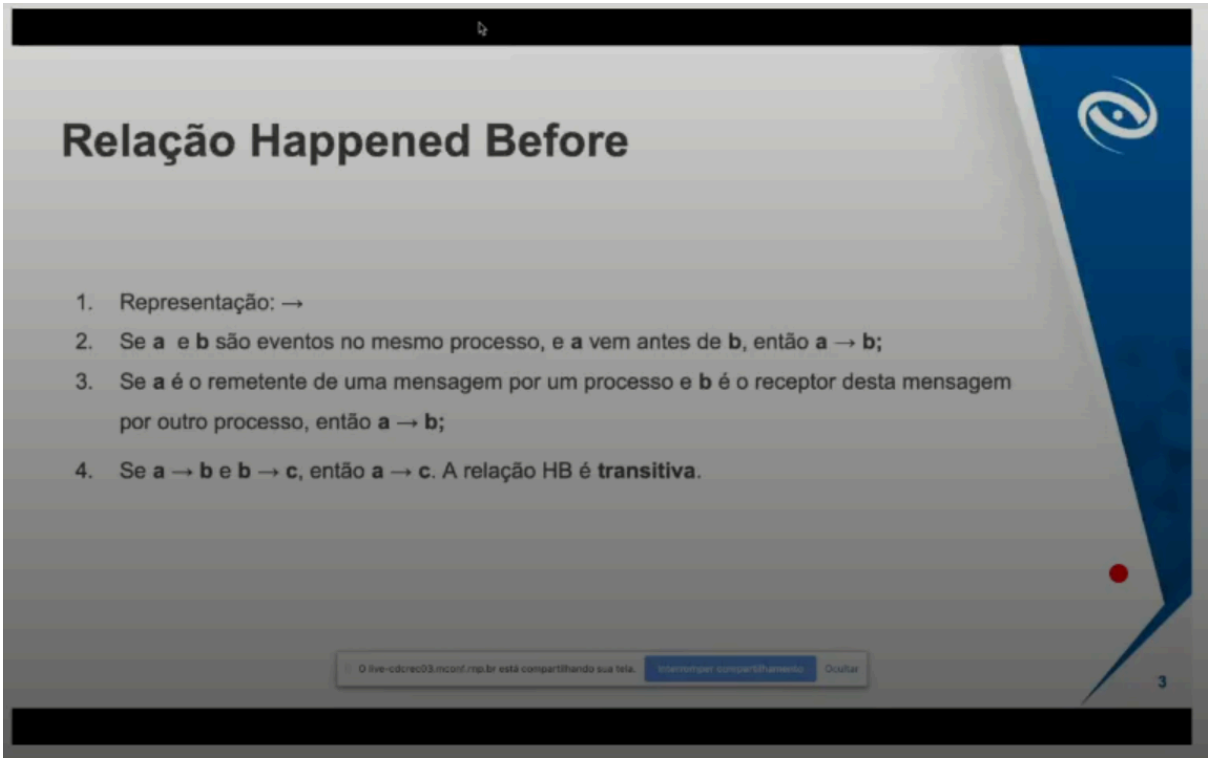


Figura da esquerda: Relógio Lógico de Lamport.

Relógios Físicos não são úteis em um sistema distribuído.

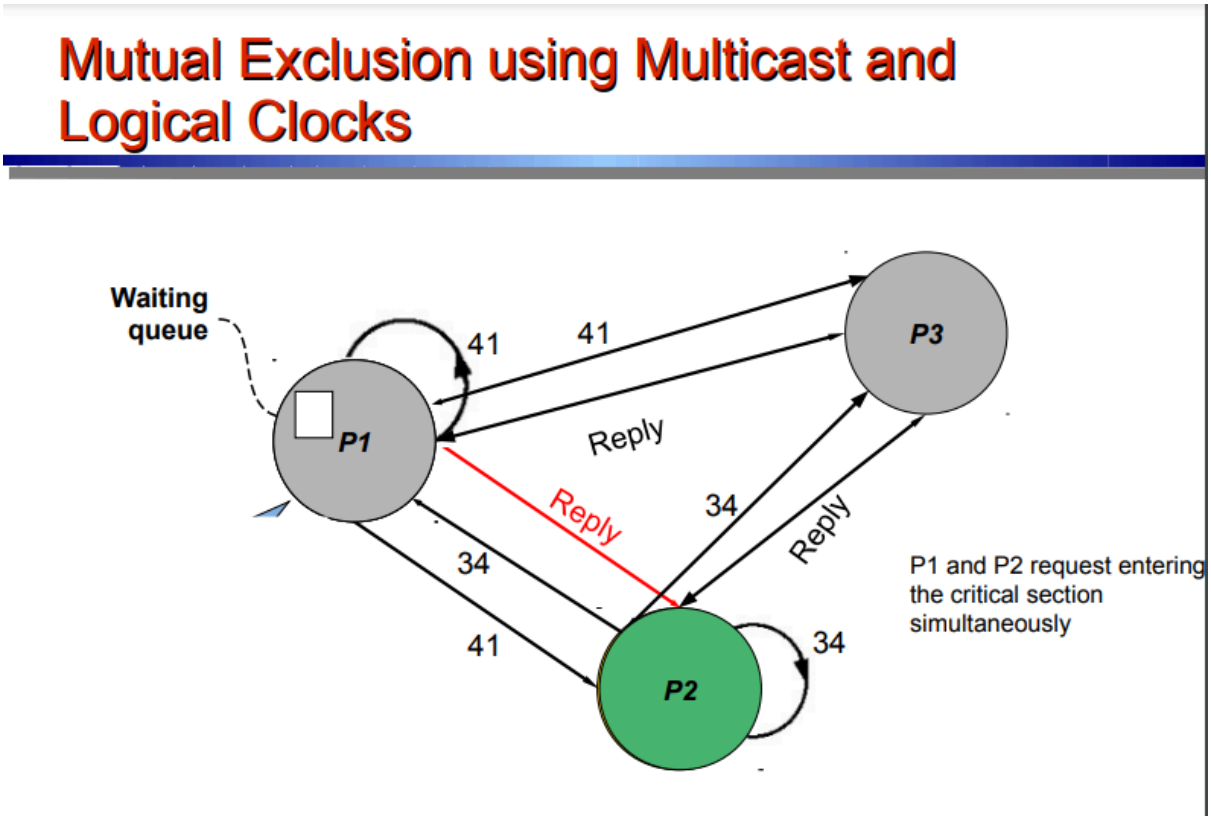


Mash/Mashing: Serialização ??

Proxy: o proxy é um objeto que atua como um substituto local para um objeto remoto. Ele é parte fundamental do mecanismo de comunicação entre cliente e servidor em sistemas distribuídos e segue o padrão de projeto Proxy.

Stub (no lado do cliente): O **stub** é o componente do lado cliente que atua como um proxy local para a interface remota. Ele usa a descrição da IDL para saber quais métodos podem ser chamados e como os dados devem ser serializados para envio ao servidor.

skeleton (no lado servidor) também é gerado com base na IDL e é responsável por deserializar as solicitações recebidas, delegando-as à implementação real da interface.



Pi pede para entrar na sessão crítica.
É enviado a todos os processos, juntamente com o timestamp que P1 quer entrar na sessão crítica (Multicast). E é aguardado até que todos os processos tenham recebido o pedido.

Quando um processo recebe um pedido <Ti, pi> no pj (processo j).
Se o estado for HELP “OU” estado = Wanted E T, pj < Ti, pi.
Então será enfileirado o pedido de pi sem responde-lo.
Caso contrário deve ser respondido a pi

Temos que o Stub (Toco), é utilizado pelo cliente para interagir com métodos remotamente, como se estivesse em seu próprio local de execução. Logo, ele “engana” o compilador para que não sejam detectados erros antes de recuperar essas informações de outro local.

Lista Geral Sistemas Distribuídos:

1)

1. V
2. F
3. F
4. F
5. F

2)

V
V
F

3)

3. Correta
4. Correta
5. Correta

4)

a

RPC/RMI

1)

1. Sobre invocação remota de método:
    1. Por que o desenvolvedor deve descrever a interface remota usando uma IDL?
    2. Qual a relação entre a descrição da interface remota e o stub?
    3. Por que o stub tem a sua implementação baseada no padrão de projeto proxy?
    4. Explique o funcionamento do serviço de nomes e o porquê de seu emprego na invocação remota.

1. IDL (Interface Definition Language) é utilizada para definir os métodos e parâmetros que podem ser acessados remotamente, independente da linguagem utilizada garantindo a heterogeneidade entre um cliente e um servidor

2. Temos que o Stub (Toco), é utilizado pelo cliente para interagir com métodos remotamente, como se estivesse em seu próprio local de execução. Logo, ele “engana” o compilador para que não sejam detectados erros antes de recuperar essas informações de outro local.

3. O stub utiliza o padrão de projeto **proxy** porque ele atua como um intermediário que imita o comportamento da interface remota no lado cliente.

   Ele fornece uma representação local da interface remota, permitindo que o cliente faça chamadas a métodos remotos como se fossem métodos locais.

   O stub cuida da serialização dos dados enviados ao servidor, da comunicação através da rede e da deserialização da resposta.

4. O serviço de nomes é utilizado para encontrar a localização de um método em específico em um ambiente distribuído. Seu emprego na invocação remota consiste em um cliente fornecendo o nome de um método em específico e a partir desse nome fornecido ele consegue recuperar um método em um servidor. Quem é responsável por apontar o endereço do servidor que contém esse nome ao cliente é o serviço de nomes.

2. Na construção de uma aplicação com objetos distribuídos:
   1. Qual a função do stub e do skeleton? Por que também são conhecidos como proxy?
   2. Qual a função do Módulo de Referência Remota?
   3. Qual o papel da referência ao objeto? Explique cada uma das informações que a compõem.

1. Qual a função do stub e do skeleton? Por que também são conhecidos como proxy?

Função do Stub:

- O **stub** é o componente no lado do cliente que atua como um intermediário para a comunicação com o objeto remoto no servidor.
- Ele:
  1. Serializa (ou "marshal") os parâmetros da chamada do método para enviá-los ao servidor.
  2. Envia a solicitação para o servidor.
  3. Recebe a resposta do servidor e a deserializa (ou "unmarshal") para apresentá-la ao cliente.

Função do Skeleton:

- O **skeleton** está no lado do servidor e serve como o intermediário entre o stub e a implementação real do objeto remoto.
- Ele:
    1. Recebe as solicitações do stub.
    2. Deserializa os parâmetros recebidos.
    3. Invoca o método correspondente no objeto remoto real.
    4. Serializa a resposta e a envia de volta ao stub.

    Por que são conhecidos como proxy?

- Tanto o stub quanto o skeleton seguem o **padrão de projeto Proxy**:
    - O **stub** age como um **proxy local**, permitindo que o cliente invoque métodos no objeto remoto como se ele estivesse no mesmo processo.
    - O **skeleton** pode ser visto como o **proxy no servidor**, cuidando da comunicação entre a interface de rede e a implementação do objeto.

---

2. Qual a função do Módulo de Referência Remota?

O **Módulo de Referência Remota (MRR)** é responsável por:

- Gerenciar a comunicação entre o stub e o skeleton.
- Estabelecer conexões de rede, garantindo que os dados sejam enviados corretamente do cliente para o servidor e vice-versa.
- Identificar e localizar o objeto remoto a partir de sua referência.
- Gerenciar a serialização e deserialização dos objetos complexos.

Ele age como uma camada de abstração entre a aplicação e os detalhes de transporte e rede, simplificando a implementação da comunicação remota.

3. Qual o papel da referência ao objeto? Explique cada uma das informações que a compõem.

???

5. Um dos mecanismos de comunicação mais usados em sistemas distribuídos é o de chamada remota de procedimento (RPC).
    1. Que vantagens o uso de chamada remota de procedimentos pode oferecer ao programador, se comparado com o uso direto de uma API de mensagens como a de sockets? Há desvantagens? Se sim, quais?
    2. Em relação ao RMI, que vantagens este pode oferecer ao programador, se comparado com o uso direto de uma API de mensagens como a de sockets? E quando comparado ao RPC?

1. **Vantagens do RPC**:
   Abstração do mecanismo de mensagem, funciona como uma função, passando parametros para outro host.

Favorecem a heterogeneidade.

Desvantagens:
Pode causar sobrecarga na rede

2. **Vantagens do RMI:**
   Paradigma da orientação a objetos, favorecendo chamadas a objetos complexos.
   permite que um cliente execute métodos distribuídos como se estivessem locais.

   Desvantagens:
   Não favorece a heterogeneidade, ou seja, é extremamente complexo fazer com que duas linguagens diferentes utilizem do RMI para se comunicarem.
   Pode causar uma sobrecarga na rede maior que o RPC e Sockets.