

Unidade 01: Introdução

Definição SD: coloção de comp. indep. que se apresenta aos usuários como único sistema coeso (transp.). Aspectos importantes:
composto de comp. autônomos (cooperando) \rightarrow não é pressuposto seu tipo e como estão conectados; capacidade de ocultar diferenças na interconexão e organização interna; interface uniforme e consistente em qualquer ponto do SD; elastic computing (grau de adaptabilidade conforme demanda) e escalabilidade \rightarrow justificativas para implementar SD's. Intercon. é necessário uniforme entre nós em um SD.

Middleware: camada intermediária que simplifica comunicação e integração entre aplicativos distribuídos, fornecendo abstração para as complexidades reais, facilitando desenvolvimento. Oferecem interfaces/APIs de alto nível, não é mandatório \rightarrow manualmente.

1. Disponibilidade de recursos: facilitar o compartilhamento de infas e colaboração. Aumentar a importância da segurança com o aumento da conectividade e compartilhamento de dados.

2. Transp. de distribuição: apresentar o SD como único para os usuários (nós e programas)

- Nexo: ocultar diferenças de representação de dados e como os recursos são acessados
- Localização: ocultar onde um recurso está localizado
- Migração: ocultar que um recurso pode ser movido para outro local. (Ira em um nó, vai em outro, sem que o programador saiba)
- Realização: ocultar que um recurso pode ser movido para outro local enquanto está sendo usado
- Repliação: ocultar que um recurso pode ser repliado
- Concorrência: ocultar que um recurso pode ser compartilhado por múltiplos usuários concorrentes
- Falhas: ocultar a falha e recuperação de recursos

3. Interoperabilidade aberta: um sistema aberto é possível de integrar com outros sistemas de sistemas abertos, seguindo regras padronizadas para serviços, geralmente especificadas por interfaces em uma IDL (linguagem de descrição de interface). Portabilidade é essencial

4. Escalabilidade \rightarrow vertical (aumentar capacidade do sistema); horizontal (adicionar de nós paralelos \rightarrow menor limite que vertical). Propriedade essencial medida em dimensões de tamanho, geografia (distância próximo do usuário) e administrativa (preocupações com órgãos regulamentares de internet). (existe em cada uma das dimensões \rightarrow problemas diferentes de escalabilidade):

- Serviços centralizados: lógica de negócios em um único nó \rightarrow gargalos e pontos críticos de falha (único ponto p/ todos usuários)
- Algoritmo central: alg. executado em único nó, limita capacidade de processamento e aumenta latência (necessita de todo info completa)
- Dados cent.: todos dados armazenados em um único local \rightarrow problema de capacidade e desempenho

Técnicas de escalabilidade:

1. Ocultando latência: entre máx. espera e uso de comunicação direta. Reduzir comunicação anárquica + menor processamento dos servidores para o cliente em aplicações intensivas.

armazenamento

2. Distribuição: dividir componentes em partes menores e distribuir-las para reduzir problemas de capacidade limitada de processamento

3. Repliação: distribuir várias instâncias do mesmo componente para aumentar disponibilidade e equilibrar a carga do sistema. Cache: uma forma especial de replicação, mas conservação de dados é um desafio.

Premissas falsas:

- Rede é confiável, segura e homogênea
- Latência é 0
- Custo de Transp. é 0
- Topologia inalterável
- Segurança de banda infinita
- Há ^{mais} um administrador

Tipos de SDs:

- Stand-alone
- Sist. Comp. Distribuído: em grid (rede WAN) ou clusters (rede local)

• Sist. Processamento de Transações: vários nós \Rightarrow mesmo conjunto de atendentes; TP Monitor (Middleware ger. modo)

• Sist. Processos Distribuídos: IoT's, infraestrutura de sensores \Rightarrow robustez p/ propagar alertas, dados seguros, comunicação entre sensores, manter integridade de dados. Ex: sist. healthcare para idosos

• Stand-alone: sistema/servidor opera independentemente em unica máquina/nó, nem depende de outros sistemas ou nós. Todos componentes e processos existem no mesmo hardware

Unidade 02: Arquitetura

- Estilos Arquiteturais: organização lógica dos componentes de software em um sistema.
Formulado em termos de componentes, conexões entre eles, dados, trocas e configuração. Componentes: unidades modulares com interfaces bem definidas, comunicam-se por conectores, mecanismos para medição entre componentes (RPC, streaming de dados) partagem de montagens
- Arquitetura em Camadas: organiza componentes em camadas, permitindo chamadas apenas para camadas sub-jacentes. Implicitamente usada em sistemas cliente-servidor e aplicações distribuídas como motores de busca, corredores de ação e office suites.
- Arquitetura Baseada em Objetos: cada objeto é um componente, conectado por chamadas de procedimento remoto (método). Facilita a encapsulação de dados e operações, fundamentalmente arquiteturas orientadas a serviços (SOAs)
- (1) • Proxy e Skeleton: realizam a interpretação de mensagens, serialização de seu conteúdo (marshalling) e envio/recepção desses dados entre objetos remotos em um SD.
- Arquitetura Baseada em Eventos: processos se comunicam através de um repositório comum, por eventos propagados, geralmente associados a sistemas de publish/subscribe. Vantagem na redução de acoplamento fraco entre processos.
 - Event bus: propaga os eventos gerados por publishers para todos subscribers inscritos para aqueles eventos.
- (2) • Arquiteturas baseadas em Receptores/Dados: processos comunicam-se por meio de um repositório comum. Desacoplamento no tempo pelos processos; visualização do SD como uma coleção de recursos gerenciados por componentes.
- Arquiteturas de sistema: localização física dos componentes de software (onde não colabora)
 - Centralizadas: organização em termos de clientes que solicitam serviços de servidores, gerenciando a complexidade dos SDs. Modelos: arquiteturas cliente-servidor e em camadas (layered).
 - Descentralizadas: crescimento notável em sistemas peer-to-peer (P2P):
 - Estruturadas: organiza os nós em uma rede overlay estruturada (hierárquico ou anel lógico); não espalh. → ID sistema privado/abre a operação Look UP (KEP) que eficientemente sistema requisita para o nó associado (Ex: CDN)
 - Não estruturadas: organiza os nós em random overlay → dois nós ligados com probabilidade p. Bucos não-determinísticos, recorre a técnicas como: Flooding (anúncio de consulta lookup a todos os registradores) e random walk (seleciona um registro aleatoriamente, se não tiver a resposta, seleciona aleatoriamente um dos seus registradores)
 - Híbrida: combinação de características arquiteturais por muitos SDs, como cliente-servidor com P2P (descentralizado)
 - Edge-serv: servidores na borda da rede, como ISP. Usado para redes de entrega de conteúdo (CDN).
 - Sistema Colaborativo Dist.: transição de um modelo cliente-servidor para colaborações totalmente descentralizadas, permitindo a distribuição paralela de partes de aplicação entre os nós de uma "nuvem".
 - * Overlay Networks: predominam, roteando dados entre nós, com endereço lógico em nível de aplicação, são a base para implementações de sistemas colaborativos distribuídos e redes privadas virtuais (VPNs)
 - Arquitetura vs. Middleware: atua como uma camada entre aplicações e plataformas distribuídas, buscando pela transparência de distribuição: de dados, processamento e controle das aplicações.
Defeito/Problema: adaptação dinâmica de middleware para estilos arquiteturais específicos, frequentemente facilitado por interpretadores.

(1) mechanismo responsável pela execução de um método implementado em um objeto remoto. Skeleton: responsável pela manipulação dos métodos localmente no objeto remoto. Proxy: responsável pela interface com o objeto cliente (chamada/caller)

(2) servo: não pede liga e serve rule; ele pede an centralizado ou distribuído; diferentes componentes de SD podem trazer dados

Unidade 03: Processos

Processos: papel crucial em SD; embora processos sejam blocos fundamentais em SDs, threads, com granularidade mais fina, são frequentemente delegadas para melhorar o desempenho.

Modelos de Implementação:

	Espaço do Usuário (M:1)	Espaço do Kernel (1:1)	Modelo Híbrido (M:N)
Definição	Múltiplas threads de usuário não mapeadas para uma única thread do kernel	Cada thread do usuário é mapeada diretamente para uma thread do kernel	Múltiplas threads de usuário não mapeadas para um nº menor de threads do kernel
Eficiência	Eficiência em termos de troca de contexto de thread	Menor controle e suporte do kernel para gerenciamento de threads	Combina eficiência e controle do espaço do usuário com suporte do kernel
Características	Gerenciadas sem intervenção do kernel	Troca de contexto mais custosas	Oferece um equilíbrio entre eficiência e flexibilidade
Uso em SD	Inadequado para sistemas distribuídos onde a eficiência de troca de contexto é crucial.	Menor comum em SDs devido à sobrecarga das trocas de contexto.	Implantado amplamente em SDs para otimizar desempenho e recursos.

Threads em Sistemas não Distribuídos:

- Benefícios: independência de contextos de execução dentro de um processo e exploração de paralelismo em syst. multiprocessador
- Uso eficaz para economizar recursos do sistema, evitando sobrecarga de IPC causada por processos cooperativos
- Estruturação mais simples para muitas aplicações como IDE's, jogos, CUI's, editores de texto e planilhas

Threads em SDs:

- Permitir que chamadas de sistema bloqueantes ocorram sem que todo o processo seja bloqueado
- Facilitam expressão de comunicação em SD, mantendo múltiplas conexões lógicas simultâneas
- Explora o paralelismo em sistemas multiprocessador, separa elas do lado do cliente quanto do lado do servidor

Clientes Multithreaded:

- Facilitar latência de comunicação em sistemas WAN, facilitando a expressão de comunicação ao manter várias conexões lógicas simultâneas. Para a ocultação de latência, uma das formas comuns é: (latência de comunicação)
 - Iniciar a comunicação e ao mesmo tempo executar alguma outra atividade até a chegada da resposta
- Ex: navegadores da web

Servidores Multithreaded:

- Benefícios não apenas na simplificação do código do servidor, mas também na exploração de paralelismo para alto desempenho.
- Não é adequado para cenários em que o número de clientes simultâneos é elevado (ex: 10 mil ou mais) e uma thread é criada para servir cada cliente. Ex.: servidor de requisições
 - 10K → problema de congestionamento de rede para lidar além de 10K conexões simultâneas / cliente

• Modelos

- Threaded: Parallelismo, chamadas de sistema bloqueante. É um fluxo de execução interno a um processo.
- Single Threaded: Sem parallelismo, chamadas de sistemas bloqueantes. Única thread principal de exec. e torrar req.
- Finite State Machine (FSM): Paralelismo. Não pode usar chamadas de sistemas bloqueantes, não suporta múltiplos fluxos de execução independentes, atendimento de múltiplas requisições concorrentemente.

• Organizações de Clientes e Servidores

- Cliente: Foco em GUI. Modo de operação: software no lado do cliente interagindo com serviço remoto protocolado. Adaptado para transp. de dat.: aviso, localização/migração, replicação e de falha. Tipos de clientes:
 - Thick (rich): usa recursos locais (CPU, Memória, Armazenamento).
 - Thin: executado inteiramente no lado do servidor, não usando recursos da máquina cliente.
- Servidor: Um processo implementando um serviço específico em nome de uma coleção de clientes. Tipos comuns: de arquivos, web, banco de dados, aplicativos, comunicação, segurança, nome/diretório, metadados/att/etc. Geralmente organizados para esperar por uma requisição do cliente e garantir que seja atendida, aguardando a próxima.
 - Iterativo: lidam com um cliente de cada vez. Tratam apenas uma requisição de cada vez.
 - Concorrente: vários clientes simultaneamente, podem tratar várias requisições (independente do cliente).

→ Contato com servidor: end points

- Onde os clientes contatam o servidor.
- Clientes enviam requisições para um end point (porta) na máquina onde o servidor está em execução. Como saber qual: atribuir globalmente end points para servidores conhecidos ou usar "nome servir": DNS
- Serviços com end points pré-atribuídos
 - Alguns servidores não requerem, como um timer de hora do dia sua end point aberta dinamicamente pelo SO local.
 - Soluções:
 - Daemon (processo especial) → roda em cada máq. que roda o servidor, curvando o end point atual
 - Supervisor (supervisor) → em syst. Unix, quando vários end. points e criando processos para manipular solicitações.
- Interrupção de um servidor
 - Importante se considerar como um servidor pode ser interrompido. Interrupções: interromper abruptamente o cliente ou enviar dados fora de banda (out-of-band) para processamento urgente pelo servidor.

• Stateless

Eg: servidores web para pag. estáticas

- Não mantêm informações sobre o estado dos clientes.
- Processam ^{reduz.} requisições com base nas info. fornecidas, sem depender de requisições anteriores.
- Cada requisição encapsula todas as info. do cliente necessárias para o servidor executar essa requisição.
- Stateful
 - Processam requisições com base em info. de requisições atuais e anteriores.
 - Enviam aviso a info. de estado durante o processamento.
 - Clientes não podem trocar de servidor a cada requisição, apenas em cada conexão.
 - Pode processar todas as requisições associadas às mesmas info. de estado ou compartilhar info. de estado entre servidores.
 - Uso maior recursos de memória principal do que a versão Stateless. Pode ser implementado como servidor do tipo FSM.

FSM: tradicionalmente (sem arquitetura multithread) gera erros paralelos nem.

Unidade 04: Comunicação

5,6 e 7

Capada de Middleware

- Middleware fornece serviços e protocolos comuns para várias aplicações. Serviços geralmente encontrados nas camadas 6,7 do modelo OSI.
- Exemplos: chamadas de procedimento remoto (RPC) - L7; marshaling/demarshaling de dados - L6; protocolos de nomenclatura (naming protocol) - L7; e sincronização - L5.

Tipos de comunicação

- Transiente: mensagem armazenada pelo middleware apenas se ambos remetente e destinatário (sender e receiver) estiverem em execução (exige aderência no tempo). Desvantagem se a entrega não for possível devido a uma interrupção na transmissão ou destino morto.
- Persistentes: mensagem armazenada até ser entregue ao destinatário (adecapamento no tempo não é necessário), middleware armazena a mensagem em instalações de armazenamento.
- Síncronas: remetente bloqueado até que o destinatário seja ativo. Três pontos de sincronização: no início, durante a entrega e após o processamento.
- Asíncronas: remetente continua imediatamente após enviar a mensagem. Mensagem temporaneamente armazenada pelo middleware.
- Combinação de Persistentes e Síncronos: persistência com sincronização na submissão da solicitação (requisição) e comunicação transitória com sincronização após o processamento da requisição → correspondente ao RPC
- Middleware Orientado a Mensagens → objetivo: alcançar comunicação assíncrona persistente de alto nível
Processos enviam mensagens, que não enfileiradas, permitindo que o remetente realize outras tarefas.
Middleware frequentemente garante tolerância a falhas
- Computação Cliente/Servidor: geralmente baseada em comunicação síncrona transitória.
Problema: cliente esperando por uma resposta, falhas devem ser tratadas imediatamente (páginas e cliente só esperando), inadequado para certas aplicações (email, notícias)

Mecanismos de Comunicação

- Chamada de Procedimento Remoto (RPC): protocolo e mecanismo para comunicação cliente-servidor (comun. síncrona)
- Objetivo: fazer com que uma chamada de procedimento remoto pareça o máximo possível com uma chamada local.
- Uso: em etapas envolvendo stubs do cliente e do servidor, troca de mensagens e passagem de parâmetros (wrapping de parâm. em uma mensagem) considerando encoding/decoding dos valores dos parâmetros dos procedimentos durante uma chamada remota (encoding/decoding).
- Dificuldades: garantir a compatibilidade das diferentes representações concretas de dados entre os vários níveis de computação envolvidos (máquinas de cliente e servidor), como considerando a ordem de bytes (endianness) de uma palavra na memória do computador (little e big endian) e serialização (valor em uma sequência de bytes).
- Stub: código que implementa a interface de todos os procedimentos processados remotamente
- Função: intercepta chamada local
converte os valores dos argumentos de entrada em uma representação abstrata
empacota esses valores e os envia para o servidor
desempacota no servidor
converte a representação abstrata em uma concreta
chama o procedimento local no servidor
recebe a resposta e segue as mesmas etapas na mesma ordem (do servidor para o cliente).

- Comunicação Multicarte: enviar a mensagem para vários nós
- Multicast em Nível de Aplicação: Técnica de multicast em nível de aplicação interligada com a tecnologia P2P e através rede de sobreposição (overlay network) elaboradas como: multicast em nível de aplicação baseado em árvores e redes de malha.
- Problemas de performance em rede overlay: não é simples a construção de árvores eficientes, para tal são necessários métodos de desempenho:
 - Estresse de link (sink stress): indica quantas vezes um link expende e utilizada para transmitir informações. Busca minimizar a carga em links individuais, distribuindo o tráfego de forma mais equitativa.
 - Penalidade de Atraso Relativo (Relative Delay Penalty - RDP): mede a razão entre o atraso entre dois nós na árvore overlay e o atraso correspondente na rede física subjacente. Objetiva minimizar diferença entre os atrasos na sobreposição e na rede física.
 - Custo de nó (tree cost): refere-se à métrica global relacionada à menorização dos custos agregados dos links na árvore de sobreposição. Visa otimizar a estrutura da árvore, considerando os custos adicionais associados aos links interligados.
- Multicast baseado em Flooding: rede overlay por grupo de multicast para evitar infiltrações equivalentes à transmissão de uma mensagem para um grupo.
- Multicast baseado em Gosseling: baseado em comportamento epidêmico e informações locais. Propagam informações rapidamente dentro de uma larga colônia de nós, usando apenas info. locais. Modelos de propagação populares: anti-entropia. Modelos de disseminação de infec., incluindo abordagens de push (rápida no começo, lenta final), pull (lenta no começo rápida final (mais nós infectados)) e push-pull (qualidades de ambos).

Gabarito

01.	C) <input checked="" type="checkbox"/>	X ✓	2,5
02.	A - Upcall, Downcall; B - Proxy, Skeleton; C - Publisher, Subscriber; D - Active, Passive reporting	✓	5
03.	B) ✓		5
04.	E) ✓		5
05.	D) <input checked="" type="checkbox"/>	X ✓	2,5
06.	A) C) ✓		5
07.	A) ✓		5
08.	B) D) X ✓		2,5
09.	A) ✓		5
10.	C) D) ✓		5
11.	B) C) ✓		5
12.	C) ?		2,5

Unidade 05: Naming

machine friendly
Identificação para uma entidade

- Permitir que entidades sejam conhecidas e usadas independentemente dos seus endereços, para que processos acassem serviços remotos
- Nomes: string de caracteres usados p/ referenciar uma entidade. Endereços: access point das entidades. Identificadores: string de bits
- Identificadores individuais são independentes de localização (location independent) e referencia única a uma entidade (true identifier name)
- ARP • Eflat Naming: redução de identificadores (nomes simples, não estruturados, não contendo nenhuma info sobre como localizar endereço da entidade), para endereços. Ex: nome do serviço (www, ftp, rpc). Ex: protocolo ARP

- ① → Broad/Multicasting usada em redes LAN. Host envia uma msg de broadcast (p/ todos da rede) com uma query p/ receber um dado nome. Host que tem esse nome responde diretamente p/ o requestante com seu endereço. Ex: redução IP → MAC, via protocolo Address Resolution Protocol (ARP), em que um host deseja obter o MAC address para um dado IP address
- Name Location: cada host móvel usa um endereço IP fixo. Comunicação é inicialmente direcionada ao home agent (HA) do host móvel. HA encaminha o tráfego p/ o endereço temporário (care-of-address) do host móvel. → Fairwading Pointers: quando uma entidade move de um host A p/ um host B, esta dava uma referência no host A p/ o seu novo endereço (B), de forma que todas as consultas de serviços destinadas a entidade no host A retornariam p/ seu novo destino. Nesse método, a redução → processo recorrente.
- DHT: baseado em peer-to-peer. Poxi lookup service. Entidades respondem p/查字 (Key) em um espaço de identificadores, e cada nó na rede é responsável por uma faixa de identificadores. Ex: Chord, cada entidade com key K é gerenciada pelo nó com menor identificador id > K. Chord: usa um espaço de identificadores de m bits para atribuir identificadores aleatórios aos nós e查字 de entidade. Cada nó mantém uma tabela de dados (finger table) para rotas consultar. A consulta por uma查字 K é direcionada ao nó q com o índice k na finger Table do nó p. Ele devolve uma redução de查字 eficiente em O(log N), onde N = nº de nós na rede.
- DNS • Structured naming: human friendly. Organização hierárquica de nomes em namespaces. Iba-se um grafo direcionado com nós de diretório e nós-folha p/ representar entidades e diretórios. Redução de nomes inicia-se a partir de um ponto de fechamento conhecido. Paths e routing de namespaces permitem a combinação de diferentes namespaces. Ex: path de arg. em file server (ftp://www01/..pdf)
- Namespaces: grafo direcionado com nós de diretório e folhas. Cada caminho no grafo é referido por uma sequência de rótulos de rotas.
- Resolução de nomes: processo de lookup de info arms. em um nó referido por um nome. Processo de fechamento para iniciar redução (recursivo)
- ↳ Recurso: cliente solicita a consulta para o servidor de nomes raiz, que devolve o nome consultado ou serv. de nomes em sua hierarquia de forma recursiva. Processo continua até que o nome seja resolvido ou uma mensagem de "desconhecido" seja retornada. Vantagem: remove a carga de processar consultas de resolução, para o serv. de nomes de envelope direto; simplifica o processo para o usuário, que recebe a resposta do servidor de nomes raiz consultado. Iterativa: cliente consulta o servidor de nomes raiz, que não pode devolver diretamente o nome, mas indica um dos servidores de nomes imediatamente informando que o nome ~~é resoluto~~ é desconhecido. O usuário só consulta este servidor, e o processo continua até que o nome seja resolvido ou uma msg de "desconhecido" seja retornada.
- Vantagem: reduz a carga do servidor de nomes, distribuindo-a entre os diferentes clientes (resolv).

- Sintax: aliases permitem referenciar uma entidade por vários nomes. Host list: múltiplos caminhos abertos para referir a mesma nó folha num grafo de nomes. Type, Symbolic link: representam uma entidade através de um nó folha, mas em vez de em vez armazenar o endereço ou nome dessa entidade, o nó armazena um nome de um caminho aberto para outro nó.
- Metting: permite a combinação de diferentes namespaces de forma transparente.
- Planejamento nos sistemas de nomes: baseados em níveis. DNS: papel → de middleware de redução de nomes no lado do cliente. Funcionamento → o usuário realiza as consultas no servidor DNS, por meio de mensagens. Nota: base como implementa cache dos nomes resolvidos p/ fins de otimização das futuras consultas. Cache no servidor sincroniza com o servidor de nomes e as consultas geradas de forma iterativa ou recursiva. Todas localidades em uma rede devem manter sua parte na cache enquanto for necessário.
- DNS: Structured naming, namespaces hierárquicos e suporta lookup iterativo

- ① Drawbacks: se torna inflexível a medida que a rede cresce; largura de banda desperdiçada com msgs de request; muitos hosts interrompidos por request;
- ② cache de ponteiros se torna pode se tornar tão longa que encontra ~~uma~~ entidade se torna problemático. Ponteiro perdido → entidade não pode ser alcançada localizada

LDAP

- Atributo based naming: tb conhecido como servos de diretório, complementam estruturas de nomeação. Traduzem diretamente o buscando entidades com base em atributos associados → Entidades não vinculadas a atributos como localidade, org, CN, dc, country, orgname. Pode usar em combinação com base em valores específicos de atributos, retornando result correspond. Combinatio / dnSearch: LDAP, melhora a gerenciamento de servos em SD. Ex: objeto LDAP (C = BR, O = UFU, OU = FACOM, CN = ServPonto01)

Unidade 06: Sincronização

↳ Lightweight Directory Access Protocol

- Relógios físicos (hardware): conjunto de componentes físicos, mecanicos e eletrônicos que geram, contam e armazem clock ticks (gerados por cada oscilação de um cristal de quartzo que descreve um contador associado).
- Relógio de software: relógios e estruturas de dados que, por meio de interrupções (IRQs) geradas pelo hardware clock, mantêm a info de data e hora atualizada e disponibiliz p/ os processos de aplicação; sua operação ocorre no contexto do SO.
- Relógio lógico: tb implementado via software pelo middleware p/suporte aos processos distribuídos, em especial p/suportar a ordenação
- Algoritmos de sinc.: tem obj de manter preciso interna (sincro. entre os nós) e externa (sincro. com relógios físicos)
 - de Chreston: clientes contatam um servidor de tempo, informando o atraso de msg p/calcular a sincronização
 - de Berkeley: um servidor de tempo ativo consulta periodicamente os nós p/calcula uma média e ajustar os relógios dos outros nós.
 - Protócolo de Tempo de Pêndulo (NTP): servidores passivos informam o atraso e a sincro. entre si, dividindo-os em stratos para evitar同步化
 - baseada em uma hierarquia de serv., onde cada nível da hierarquia → Stratum (normalmente 4), onde cada serv. da Stratum 1 (1 nível) obtém sua hora por meio de servs de UTC receivers; - cada passo de serv. NTP, em mesmo stratum ou diferente, trazem info de sincro.. eventualmente, computam o offset entre os nós por meio do alg. de Chreston, resultando no valor. O (total), o qual serve p/ induzir a defasagem entre os relógios de ambos os servidores; - eles calculam o delta, que indica o delay médio entre os servidores, o que serve p/ ajustar os timestamps durante a sincro. entre elas. A posição na hierarquia determinará qual servidor ajustará o seu relógio ao do outro, onde o servidor de Stratum maior sempre ajusta seu relógio no servidor de Stratum menor.
 - Rel. lógicos de Lamport: como maneira de capturar a ordem dos eventos em SD. São essencialmente contadores de eventos Happens-before indica ordem entre eventos. Pode ser observada diretamente em eventos internos de um processo ou na trilha de msg entre processos
 - Algoritmo: cada processo mantém um contador local que é incrementado antes de executar um evento. Ao enviar uma msg, o emissor associa o valor atual (timestamp) de seu contador à msg. I cada trilha de msg, caso haja uma defasagem entre os relógios, de forma que o relógio do processo destinatário tenha um "tempo lógico" anterior ao do emissor, então o relógio lógico do destinatário é atualizado p/ o valor Timestamp + 1
 - Relógios internais: com os de Lamport, n/ podemos determinar a relação causal entre dois eventos apenas comparando seus tempos (relógios lógicos) São uma extensão dos de Lamport que captura essa relação causal. Cada processo mantém um vetor de contadores, onde cada ítem do vetor é o timestamp de eventos em um processo específico; Relógios internais permitem discernir precedência e dependência causal entre eventos.
 - Exclusões mutua: p/garantir que múltiplos processos n/ usarem simultaneamente os mesmos recursos; fundamental em SD. Abordagens:
 - Permissão-based (centralized): um processo é eleito coordenador. Quando um processo deseja usar um recurso compartilhado, envia uma msg de request ao coord., o qual concede permissão se nenhum outro o estiver usando. Após o término do uso do recurso, o processo informa ao coord. p/ liberá-lo. Dessr: coord. é SPOF, e pode se tornar bottleneck de desempenho em sistemas grandes.
 - Paxos & Agrawala (distribuída): usa relóg. lóg. de Lamport p/ ordenar os eventos. Quando um proc. deseja usar um rec., envia request p/ todos outros processos, os quais respondem com "OK" se n/ estiverem usando o rec. ou comparando os timestamps p/ determinar qual request tem prioridade, garantindo exclusão mutua sem deadlock ou starvation. Dessr: pode ser afetado por N SPOF's; ineficiente em grandes grupos de proc.
 - Token-ring: processos formam um anel e um Token circula entre elas; apenas proc. com o token pode usar recurso compartilhado. Após seu uso, passa para o token p/ o próximo na ordem do anel. Dessr: o perda do token se torna um problema → é preciso um mecanismo p/ detecção de token perdido, tal como o Timeout, contendo suspeição e falhas persistentes. Vant: determinismo na obtenção do recurso compartilhado.

- Fully decentralized (distributed): envolve algoritmo de votação; cada recurso compact. é replicado N vezes, com cada réplica tendo seu próprio coord. de exclusão mútua. Um proc. precisa obter a maioria de votos de $m > \frac{N}{2}$ coord. p/ um voto o rec. Desse: se uma qtdade f de recursos, haverá uma votação na instância quando um $n^o m \geq N/2$ coord. concordarem em liberar o recurso identificado $\rightarrow f+1 = \text{coord.}$
- Alg. de Eleição: todos p/ eleger um proc. especial (coord.) em um grupo de proc. Cada processo tem uma identificação única (Objetivo: garantir q todos proc. concordem sobre quem será o próximo coord.). Alg. definem na forma como localizam o coord., tds proc. se correm.
 - Bulky: quando um proc. P_k ($\text{id}(P_k) = k$) percebe que o coord. não está mais respondendo:
 - 1 - ele inicia uma eleição enviando uma msg ELECTION p/ todos proc. com $\text{id} \neq k$.
 - 2 - se nenhum proc. com $\text{id} \neq k$ responder, o proc. iniciador vence a eleição e se torna coord.
 - 3 - se um com $\text{id} \neq k$, ele assume papel de coord., eleição termina; Múltiplas rodadas \rightarrow menor eficiência.
 - Ring: baseado em um anel lógico (ring consistency). Mais eficiente que Bulky \rightarrow uma roda.
 - Quando um proc. detecta falha do coord. ele inicia uma eleição enviando uma msg ELECTION p/ os outros no anel. Se o sucessor estiver inativo, o processo remetente envia a msg p/ próximo membro ativo do anel. Em cada passo, o remetente add seu próprio id à msg, tornando-se um candidato à eleição. Quando a msg retorna ao processo iniciador, ele a reverte e a transforma em um msg COORDINATOR informando a todos quem é o novo coord. Após a circulação dessa msg, todos voltam ao trabalho.

Unidade 07: Consistência e Replicação - Replicação \rightarrow p/ aumentar disponibilidade e desempenho, parâm - prob. de consistência entre cópias

- Data-center Consistency Models: fornecem uma visão consistente do data-store p/ todos os processos do sistema. Data-store: armazenamento de dados distribuído em operações de leitura e escrita não realizadas; cada proc. assume-se que tem uma cópia local de store; → Consist. Sequential: as op. de read/write em dados são precedidas como se estivessem ocorrendo em uma ordem sequencial. Orden das op. não importa, desde que a ordem seja a mesma em todos os proc.; a mesma sequência sequencia em um P deve ser escrita no outro P.
- Data-store seq. consist.: o result. de qualquer execução é o mesmo,不管 como se as operações (read e write) sobre o data-store, realizadas por todos os processos distribuídos, forem executadas na mesma ordem por todos os proc. igualmente; n. há ref. ao tempo ou à op. de write → Consist. causal: tempo + forma da anterior. base em considerações causalidade entre eventos, aqueles que são potencialmente observados causalmente devem ser precedidos na mesma ordem por todos os proc. O p. considerar em dados não relacionados causalmente podem ser produzidos em ordens diferentes por diferentes proc. Se evento b é causado ou influenciado por um evento anterior a, causa. Isto significa que $a \rightarrow b$ → Consist. x Causa (Melds): Consist \rightarrow comportamento esperado de um conj. de dados replicados quando múltiplos proc. operam sobre eles. Causa: comport. esperado de um único item de dados replicado, no qual é garantido q as diff. cópias obedecem às regras.
- Client-centered Consistency models: fornecem garantias p/ um único proc. cliente em relações à consist. do modelo de consistência atingido. de acesso a um data-store; Têm um grau relativamente alto de inconsistência entre acessos consecutivos por diferentes clientes, especialmente iter em situações onde a consistência é limitada ou onde a tolerância à inconsistência é maior.
- Monotonic Reads: op. sucessivas de reads seguem a mesma ordem em que foram emitidas. Isso é garantido q, se um proc. lida um valor de um item de dados x, qualquer leitura subsequente de x pelo mesmo proc. sempre retornará o mesmo valor ou um valor mais recente. {A operação de write em x por um proc. é concluída antes de qualquer operação de write subsequente em x pelo mesmo processo. \rightarrow Monotonic Writes}
- Read Your Writes: garantindo q uma op. de write em data item x por um proc., após será sempre visto por uma op. de read subsequente em x pelo mesmo proc. Uma op. de write é sempre concluída antes de uma op. de read subsequente pelo mesmo proc., independentemente de onde essa op. de read ocorre. Writes Follow Reads: garantindo q op. de write em um data item x por um proc., após um read anterior em x pelo mesmo proc., será replicada no mesmo valor de x que foi lido ou em um valor mais recente. Qualquer op. de write subsequente em x por um proc. será replicada em uma cópia de x que esteja atualizada com o valor mais recentemente lido por esse processo.

- Gerenciamento de Réplicas: Possuem: onde, quando e por quem colocar réplica é usual em SD c/ replicação. Percebam: de Senadores de Réplicas e Conteúdo: encontro entre os melhores localizações p/ os usuários hospedarem o armazenamento de dados; Se aí não é útil p/ todos devesse informar Encontrando a melhor local. dos usuários; antecipa, agir com adianto de grande data centers distribuídos. Algoritmos usados buscam a otimização na seleção das melhores localidades de replicação. Tipos de replicação:

- Réplicas permanentes: são aquelas definidas a priori, ou seja, antes do início do funcionamento do sistema. Essas réplicas ficam mais distantes dos usuários. Geralmente usadas p/ fins de balançoamento de carga e alta disponibilidade e estão todas no mesmo data center.
- Réplicas criadas pelo servidor: criadas dinamicamente e se localizam em posição intermediária em relação às réplicas permanentes (mais distantes do cliente) e criadas pelo cliente (mais próximas do cliente).
- Réplicas criadas pelo cliente (cache): criadas dinamicamente, são as mais próximas do usuário; seu objetivo é performance, ou seja, reduzir o tempo de acesso do cliente aos dados da réplica.

→ Distribuição de Conteúdo.

- Notificação: protocolos de validações informam outras réplicas sobre atualizações, economizando largura de banda.
- Transferência de Dados: útil quando a relação read/write é alta; alterações são frequentemente agregadas em logs.
- Propagação de Alterações: réplicas executam operações de atualizações recebidas, monitorando custo de largura de banda; requer mais processamento, especialmente para operações complexas.

- Push: atualizações enviadas sem voluntade; usado p/ garantir forte consistência em ambientes com alta taxa de reads; eficiente, garantindo disponibilidade imediata de dados. Pull: atualizações solicitadas quanto necessário; eficiente em ambientes com baixa relação read/write; pode ter aumento no tempo de resposta em caso de falhas de cache.
- Combinação de ambos → uma forma hibrida de propagação de atualizações com base conexões (leaves), que prometem atualizações p/ um cliente por um período específico:

- Baseadas na idade: concedem leaves duradouros n/ dados que não foram modificados por um longo tempo
- Baseadas na freq. dezenas: concedem leaves longas p/ clientes cujos caches frequentemente recebem as atualizações
- Baseadas no estat: reduzem o tempo de expiração das leaves à medida que o servidor se torna sobrecarregado, esperando alunos workloads.
- Protocolos de Consistência → BASEADOS EM PRIMÁRIO de escrita em X

→ Consistência Seg.: cada data item x no data store tem um servidor primário associado, o qual é responsável por coordenar operações

- Protocolo de escrita remota: todos os op. de write precisam ser encaminhados p/ um único servidor fixo. As op. de read podem ser realizadas localmente. Ex. de escopo: primary-backup → No. q. quer fazer write em data item x, encaminha-se primay no x * continuação: primay realiza atualização em sua cópia local de x, e subsequentemente encaminha a atualização para os nexts de backup.
- Protocolo de escrita local: escopo de primary-backup onde a cópia primária migra entre prim. que devem realizar op. de write.

* continuação: primary realiza atualização em sua cópia local de x, e subsequentemente encaminha a atualização para os nexts de backup.

Gatilhos: multi. op. de write em X podem ser realizadas localmente, enquanto op. de read ainda podem obter sua cópia local.

→ Consist. Seg.: Protocolo Barcodes em Réplica: as op. de write podem ser realizadas em várias réplicas em vez de apenas uma; assim, as op. de write são executadas nos diferentes servidores onde se encontram as réplicas do item de dados atualizado.

- Ativa: requisições de atualização são enviadas a todos os réplicas. Requerem (central coord.) → ordenação total → ? → porque os serv. têm que falar na mesma ordem em Todo lugar
- (Quorum-based) barcodes em rotas de máscara, as op. de atual. são enviadas apenas p/ um subconjunto total de serv. de réplicas.

7 Shopify: Manda reads. Para conseguir isso → direcionar op. de reads p/ o mesmo servidor, garantindo consist. p/ cliente específico

8 DFS's: Tectonic file system (Meta); Neggfs file system (Netflix); Hadoop DFS; GlusterFS; Lustre FS; Amazon Elastic FS (EFS)

7 Amazon RDS: introduz read replicas p/ distribuir workloads, aumentando capacidade de read aggregate. São de tipo miúdos pelo servidor, elas monitoram automaticamente as alterações do DB principal, oferecendo flexibilidade na distribuição de carga. Melhor combinadas com implantações Multi-AZ p/ disponibilidade de geografia, elas permitem usar clusters flexíveis e escaláveis.

Modelo de consut client-centre: Read-your-writes, no qual garante consistência entre read e write, essencial p/ op. criticas

Unidade 08: Sistemas de Arquivos Distribuídos permitem que multip. pces. compartilhem dados de várias localizações; nem todos os sist. são distribuídos, contudo em cima deles, podem serem adicionados de comunicação, sincronização, replicação e consistência. Ex: NFS e AFs.

- Avg. Cliente-Servidor: cada serv. de avg. dist. fornece uma versão padronizada de seu syst. de avg. local; cliente executa op. por RPC
 - Arq. remoto: o conteúdo do avg. reside no serv. de avg. e todas op. sobre ele (pelos clientes) são feitas por meio de cham. de pce. remoto
 - Upload/Download: uma cópia do avg. é enviada do servidor p/ o cliente operar sobre ela localmente. Após a conclusão das op., no cliente, a cópia atualizada do arquivo é enviada p/ o servidor a fim de substituir a cópia original (old) pelo a atualizada (new).
 - NFS: baseado na avg. de acesso remoto que utiliza RPC (stubs cliente + servidor), marshalling, chacking, etc.) p/ a execução remota de op. Aplique-se no cliente executa op. de arquivos (read/write, open, seek, etc) como se os arq. estivessem armaz. localmente. O SO, por meio do VFS (através comando abraço do system call), identifica se a op. chamada é n/um avg. local ou remoto. Local: o fluxo segue pelos subsistemas do Kernel que lidam com op. no storage local. Remoto: um módulo NFS client é carregado p/ se comunicar com o serv. No rem., a mesma abordagem é seguida até o novo básc. os arquivos.
- DFS Baseado em Cluster utilizados p/ aplicações paralelas, onde um avg. é distribuído em múltiplas máquinas n/núcleo paralelo.
 - GFS: baseado no modelo upload/download da avg. cliente-servidor, proposto p/ operar em ambientes de cluster de computadores, dando alta-disponibilidade e performance voltada para workloads voltados p/ grandes arquivos, os quais são "fatios" (file stripes) e as partes resultantes são distribuídas em diff. nós do cluster de serv. de avg. Dessa forma, é possível usar diff. partes do avg. de forma paralela. Um nó Master fornece info de onde estão as diff. partes de um avg., assim, clientes acessam no Master p/ obter as serv. em que estão encontradas as partes do avg. que precisam; posteriormente, os clientes acessam os serv. de chunks p/ obter os dados rest. de avg. de interesse.
 - Processos uniu diff. → diferentes tipos de proc. cooperantes, como serv. de avg. e gerenciadores de avg.
 - Stateless (V2 - V3): simples, não requer recuperação de estados se o servidor falhar, mas não oferece garantias quanto ao cliente de que sua request for concluída; essa abordagem nem sempre pode ser plenamente seguida nas implementações práticas de NFS
 - Funcionamento: a cada op. sobre um avg., o cliente deve enviar a id. do avg. a seu servidor, bem como a posição onde a op. será feita e demais info que permitam o serv. a validar a op. e executá-la, tal como o ID de usuário, entre outros. Como consequência, o volume de dados de controle entre cliente e servidor é maior.
 - Stateful (V4): serv. mantém info mínima sobre os clientes. Adotado na v4 para o NFSv4 fb é esperado funcionar em WAN, e p/ tal, requer um protocolo eficiente de const. de cache; oferece garantias de acesso exclusivo a avg. com base em contratos de locação (lease)
 - Funcionamento: o serv. mantém info de estados de um avg. ativo e, a partir da sua duração, op. sobre o avg. (read, write, seek) poderão ser realizadas sem a necessidade de identificar o arquivo, a posição em que a op. terá efeito, como ocorre na abordagem anterior.
 - Comunicação entre DFS não baseado em RPC, quando tornar o sistema independ. dos SO subjetivos, das redes e dos prot. de Transp.
 - NFSv3 - 4 op.: Request LOOKUP → Response lookup name; Request READ → Response read file data;
 - NFSv4 - 2 op., procedimento composto: Request LOOKUP OPEN READ → Response lookup name open file read file data;
 - ↳ Suporte a callback: permite serv. fazer chamada de RPC p/ o cliente; exige que o serv. mantenha controle dos seus clientes, ← callback → RPC2: oferece RPCs confiáveis sobre UDP (não confiáveis). - Quando um pce. remoto é chamado, o cliente envia uma solicitação p/ o serv. e aguarda uma resposta. Serv. atualiza o cliente sobre o progresso da request e, se falhar (servidor), cliente detecta e solicita falha.
 - Suporte a side effects p/ comunicação cliente-servidor com protocolos específicos de aplicação. Por exemplo, quando um cliente faz request de acesso a avg. de um serv. de vídeo, eles estabelecem um fluxo contínuo e sincrono garantindo uma transmissão segura.
 - Permite multicasting: permite que serv. partilhem clientes com cópias locais de avg. e os notifiquem mult. sobre modificações avg., garantindo eficiência.
 - Mouting: nome geralmente organizados em um namespace hierárquico. No NFS, os clientes têm acesso transparente a um sistema de arquivos remoto, permitindo montar um syst. de avg. remoto em seu próprio syst. de avg. local. Esse NFS mounting pode ser feito diretamente através dos serviços seu próprio namespace. Porém, compartilhamento nesse modelo pode se tornar difícil se os usuários nomearem o mesmo avg. diferentes! Solução: fornecer cada usuário um nome space local que está parcialmente padronizado, subsequentemente montando syst. de avg. remoto igualmente n/ cada usuário

- Mounting/Exporting: serv. NFS exporta partes de seu rest. de arq. local, permitindo que clientes montem e usem essas partes localmente. A montagem no cliente associa uma parte do rest. de arq. remoto ao seu próprio rest. Quando o cliente usa arq. nesse namespace montado, não feta chamadas remotas ao serv. p/ acessar esse arq. Esse proc. pode ser aninhado, com partes do namespace montado tb serem montadas de outros serv.
 - Automounting: montar um namespace remoto sob demanda, i.e., quando é necessário no cliente e não na inicialização do sistema.
 - Sincronização nos SD, a remota (regras de especif. de UNIX, de serviço, de arq. imutáveis ou de transações) do compartilhamento de arq. apresenta desafios de sincronização para o desempenho. Quando vários usuários compartilham um arq. simultaneamente, é essencial definir com precisão a sequência de read e write p/ evitar problemas. Em UNIX, por exemplo, a remota geralmente estipula que uma op. de read após uma de write retorna o valor recentemente escrito. Vários writes em seguida e 1 read → valor lido é o último write.
 - File locking: em arq. cliente-servidor stateless lock manager é necessário. NFS v4 distingue read lock de write lock: vários clientes podem escrever simultaneamente a mesma parte de um arq., desde q' reja ress. de dados. Write lock necessário p/ obter acesso exclusivo n/ uma parte de um arq.
- ①
- Leases (NFSv4): tempo concedido pelo servidor p/ um lock estiver válido. Por seu término, caso não seja renovado (ex. timeout), o lock é removido pelo servidor. Esta abordagem ajuda na recuperação apóis falhas, tb evita p/ outros usuários peneirar por写入.
 - Consist. e Replicação: importante papel em DFS operando em wide-area networks (WAN).
 - Client-side caching

NFSv3: caching levado mais de forma do protocolo e geralmente não garante consistência.

NFSv4: a consistência ainda é tratada de forma dependente da implementação, c/ estratégias diferentes p/ lidar com a consist. dos dados e atributos de arquivos, como:

- 1 - cache de dados lidos do serv. 2 - cache de dados escritos no cliente e desassociados no serv. apesar quando o arq. é fechado. Vários processos no cliente podem compartilhar o cache. O cache pode ser mantido no cliente, mesmo depois q' o arq. é fechado; essa cache deve ser renovaada quando o arq. for aberto novamente no cliente.

- Segurança: tratada pela autenticação e controle de acesso nos servidores

• Funcionalidades: uso de Secure RPC p/ a comunicação entre cliente e servidor; controle de usuários, normalmente, via ACL's (Access-control list) com tipos de usuário (owner, group, everyone, etc); autenticação descentralizada, combinando o uso de Secure File System (SFS) com um repositório de autenticação (ex. LDAP)

- ②
- Ex. (stateful): serv. de arq.-dist. pode associar um lease (concessão) a cada arq. que entrega a um cliente/host, permitindo que esse cliente use exclusivamente de read e write, por um período de tempo, i.e., até q' o mesmo expire ou seja renovaado/actualizado caso não haja outro host requerendo esse arquivo, na fila. "Lembre de diferenças de forma usada depois, da de file locking"

Unidade 09: Transações Distribuídas (DT)

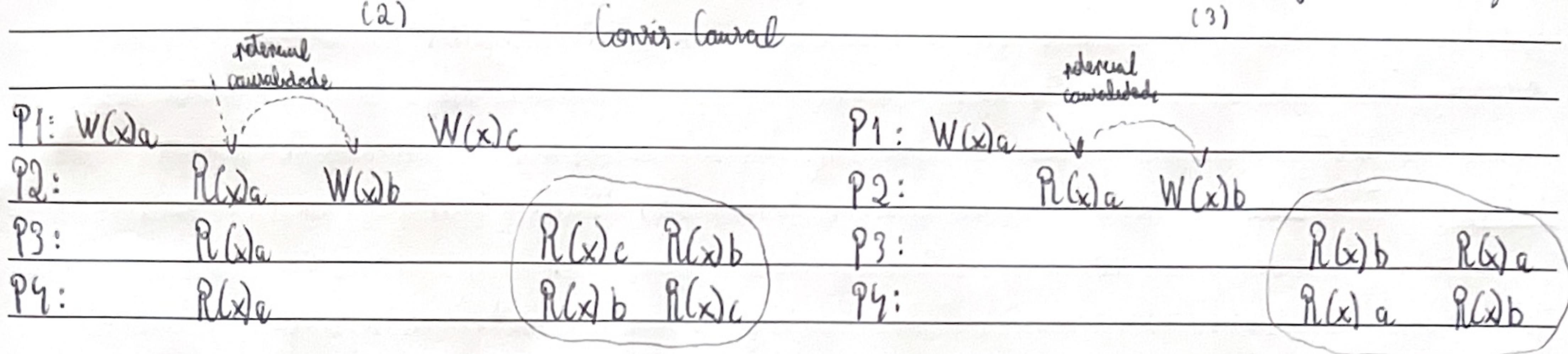
- Transações FLAT: clientes fazem request p/ mais de um serv., onde cada transação opera os objetos do serv. respectivamente, isto ocorre pois uma transação de um cliente flat completa cada um dos seus requests antes de se p/ o próximo. Concorrência é limitada. Quando serv. usa locking, uma transação pode operar sobre um objeto de cada reg., Abort se um dos serv. não puder confirmar transação.
- Transações NESTED: as transações de nível superior podem gerar subtransações ("filhos")/ gerando ainda mais outros filhos até qualquer nível de profundidade, que são executadas em paralelo (concorrente) pelas subtransações de mesmo nível, em requisições diferentes, p/ obter ganho de desempenho ou simplificar a programação. Proporciona arm., mas flexibilidade do que as Trans. planas (flat).
- COMMIT Protocols garantem q' se todos os ops. de uma transação são realizados ou nenhuma delas (0 ou 1)
- Atomos (one-phase commit): inadequado → não permite q' um serv. aborte unilateralmente uma transação quando cliente adverte commit. Problemas de controle de concorrência podem levar ao aberto sem q' o cliente saiba.

commit = confirmar

- Atomicus (two-phase commit) permite que qualquer participante abra sua parte da transação, faça do commit:
- 1 - Participantes votam para transação ser confirmada ou abortada, se um deles confirmar, não pode mais abortar. Falha estátua em node storage
 - 2 - Todos os participantes da trans. executam a decisão conjunta: - Se um abort, devem e' abortar a transação; - Se todos confirmam votarem em confirmar, ent. devem e' confirmar.
- Durante a execução de uma transação, se uma parte da transação for abortada, toda a transação deve ser abortada; exemplo: node X retorna resposta negativa na primeira fase \Rightarrow na segunda fase, coord. deve avisar a todos os outros participantes p/ não completarem a transação.
- Concurrency Control: cada serv. gerencia conf. de obj e é responsável por garantir sua consistência durante transações concorrentes. Número de uma coleção de serv. de DT não conjuntamente responsáveis por garantir que elas são realizadas de uma maneira realmente equivalente.
 - Locking: em DT, os locks em um objeto são mantidos localmente (no mesmo serv.), utilizando o local lock manager, no qual:
 - decide entre garantir/forçar um lock ou fazer a transação solitada esperar
 - porém, não pode liberar qualquer lock até saber que a transação foi confirmada, ou abortada por todos os serv. envolvidos na Transação
 - quando serv. p/ controle de concorrência, os obj. permanecem bloqueados/transados e indisponíveis p/ outras trans. durante período de commit Átomos.
 - Deadlocks Distribuidos: local \Rightarrow quando locking é usado p/ controle de concorrência: deadlocks podem surgir, sór devem prevenir-los
 - $\xrightarrow{\text{SD}}$ global wait-for graph: pode ser construído a partir de grafos locais em SD envolvendo muitos serv. sendo executados por muitas transações
 - \hookrightarrow caso haja pode haver um ciclo no grafo global que não está em nenhum grafo local, i.e., deadlock distribuído
 - \hookrightarrow há deadlock se, e somente se, existe um ciclo no wait-for graph* \Rightarrow é um grafo direcionado no qual nós representam transações e objetos, arcos representam os obj. mantido por uma transação ou uma transação esperando por um obj.
 - Detectão: requer encontrar um ciclo no global transaction wait-for graph, que é dist. entre os serv. envolvidos na Transação
 - Detectão centralizada de deadlocks \Rightarrow serv. toma papel de detector global de deadlocks \Rightarrow de tempo em tempo cada serv. envia p/ o ^{detectar} processador global sua versão mais recente do seu grafo wait-for local, e qual combinar com as demais versões recebidas, formando o grafo wait-for global, a fim de encontrar esperas cíclicas no mesmo. Detecção sum \Rightarrow duração de um único serv.
 - Phantom deadlocks: falso positivo, por causa da desatualização das info de grafos wait-for no detector global \Rightarrow Transm. de info requer um certo tempo de coleta \Rightarrow chance de que uma das trans. que detém um lock e libere enquanto isso, caso em q deadlock não encontra-se.
 - Edge changing: não exerce papel de detectar global \Rightarrow é feito por meio dos local lock managers. Algoritmo em três etapas:
 - Iniciação: um serv. envia probe (ondas) p/ o serv. de um objeto onde uma transação está bloqueada. caso contrário, encaminha-o
 - Detecção: recebe as ondas e decidem por meio de seu processamento se foi identificado ciclo no wait-for graph, ou seja, deadlock
 - Resolução/Finalização: uma Transação é abortada p/ quebrar o deadlock, combate em prioridade de Transações
 - Recuperação de Transações: não afetará das Transações \Rightarrow todos os efeitos das Transações confirmadas e nenhum dos efeitos das Transações confirmadas incompletas/abortadas refletam nos obj. mantidos por elas.
 - Durabilidade requer que os obj. sejam salvos em armaz. perm. e estarem disponíveis indefinidamente. Isto é, uma confirmação (ack) da subtransação de commit de um cliente implica que todos os efeitos da trans. foram registrados em armaz. perm., assim como objetos salvo no storage remoto.
 - Recovery file: serv. mantém um arq. de recuperação p/ registrar obj. confirmados, sendo que os obj. são mantidos na memória local. Em caso de falha (crash), o serv. pode restaurar seu obj. a partir das versões confirmadas mais recentes armazenadas em armaz. perm.
 - Recovery manager: garante durabilidade e tolerância de falhas. Tarifas: salvar obj. em armaz. perm. (com um recovery file) p/ transações confirmadas; restaurar os obj. do serv. após uma falha; Reorganizar recovery file p/ melhorar o desempenho da recuperação; recuperar espaço de armaz. (no recovery file). Deve ser resiliente a falhas de mídia (recovery file perdido \Rightarrow existir outra cópia).

- Intentions list garante que um transação trazendo os estados prepared / committed ou aborted, consistentemente. Ele mantém uma lista p/ todos transações atualmente ativas, no qual contém referências e valores de objetos alterados por cada transação.
- Transação commitada → recovery manager usa a lista p/ identif. obj. afetados e registrar suas versões confirmadas no recovery file
 - Transação abortada → recovery manager usa a lista p/ remover versões tentativas de obj.

// Isto garante que as alterações confirmadas sejam registradas e persistem com precisão, mesmo em caso de falhas de Transações //



Leitura permitida em store causalmente consistente, mas não com uma regionalmente consistente.

Utilização de um store causalmente consistente

(4)

Monotonic Reads

P1:	$W(x)_a$
P2:	$W(x)_b$
P3:	$R(x)_b$ $R(x)_a$
P4:	$R(x)_a$ $R(x)_b$

Leitura de escrita correta em store causalmente consistente

$L_1: W_1(x_1)$	$R_1(x_1)$
$L_2: W_2(x_1; x_2)$	$R_1(x_2)$

(Op. de read realizadas por único processo P em dois stores diferentes cópias locais do mesmo data-store

Monotonic Writes

Read Your Writes

$L_1: W_1(x_1)$	$R_1(x_1)$
$L_2: W_2(x_1; x_2)$	$W_1(x_2; x_3)$

(Op. de write realizadas em duas diferentes cópias locais do mesmo data-store

Data-store que promove consistência Read your writes

Writes Follow Reads

$L_1: W_1(x_1)$	$R_2(x_1)$
$L_2: W_3(x_1; x_2)$	$W_2(x_2; x_3)$

Data-store que promove consistência writes follow reads

apenas de bairros rendidos. Nós temos que não podemos p/ dividir, e a ferramenta permite "repairs" p/ depuração e desenvolvimento de novos recursos. O buffering adaptativo aumenta significativamente o desempenho em comparação com buffering fixo, beneficiando especialmente workflows envolvendo a latência.

⑧ Tectonic promove melhor utilização de recursos, simplificando armazenamento e reduzindo complexidade operacional. Isto é, abordagem antiga: Evaluar p/ eradicador, simplifica o relacionamento de desempenho entre localização e escrita estabelecidas específicas p/ cada um. Assim, sua aderência → redução significativa no n.º de clusters de armaz. MegFS → montar obj em núvem como arq. locais via FUSE no sítio de armaz. local, sem ocupar espaço em disco, e oferecer recursos como montagem de multip. obj, cache regional e buffering adaptativo. → razão: streaming