Processos e Threads

Quando pensamos num programa informático, imaginamo-lo como um conjunto estático de instruções gravadas no disco. É apenas uma receita. O **processo** surge quando essa receita é colocada em prática: o sistema operativo aloja na memória o código do programa, reserva regiões para dados e pilha, e associa a tudo isso um estado de execução (registos da CPU, contador de instruções, canais de I/O, etc.). É esse conjunto de recursos e de estado que define o processo.

A simultaneidade que vemos no ecrã, com várias aplicações aparentemente a correr em paralelo, é muitas vezes conseguida através da **multiprogramação**, onde o sistema operativo alterna rapidamente entre diferentes processos - cada com o seu "contexto" - dando a ilusão de execução simultânea. Essa alternância acontece por preempção: um temporizador obriga regularmente o sistema a interromper o processo ativo para, se for caso disso, passar a execução a outro. Para o programador, esse mecanismo está invisível, mas é fundamental para a eficiência do sistema.

Enquanto um processo encapsula o espaço de endereçamento completo e todos os seus recursos, a **thread** é uma "linha de execução" dentro desse processo. Podemos imaginar um processo como uma casa (com todos os móveis, divisão para arrumos, cozinha e sala), e cada thread como um residente que usa a casa: todos partilham a mesma estrutura, mas cada um tem o seu caminho de deslocamento (pilha própria), decide onde entra e sai, e pode aceder às divisões partilhadas. Ao criar várias threads dentro de um mesmo processo, evitamos o custo de duplicar todo o espaço de memória e de recursos; basta duplicar a pilha e os registos.

Existem dois modelos principais de implementação de threads:

1. **No espaço do utilizador (user-level threads)**: toda a gestão é feita por bibliotecas no processo, sem intervenção do kernel. É extremamente portátil e rápido criar ou bloquear uma thread, mas se uma delas fizer uma chamada de sistema bloqueante, bloqueia todo o processo.
2. **No espaço do kernel (kernel-level threads)**: cada thread corresponde a uma entidade gerida pelo sistema operativo. Permite bloqueio e escalonamento independentes, suportando real paralelismo em processadores múltiplos (SMP), mas com maior overhead de criação e comutação.

Na prática, muitas linguagens e runtimes, como a Java Virtual Machine, usam uma combinação híbrida ("múltiplas threads de utilizador mapeadas em múltiplas threads de kernel"), garantindo portabilidade e ao mesmo tempo aproveitando o paralelismo real do hardware.

Estados de um Processo e de uma Thread

Um **processo** transita classicamente pelos estados de novo, pronto, em execução, bloqueado e terminado. Quando é criado, está "novo"; logo que o sistema operativo o prepara para a CPU, passa a "pronto". O despacho (dispatch) coloca-o em "em execução". Se o processo precisa de I/O, passa a "bloqueado" até o evento terminar, regressando depois a "pronto". Finalmente, ao concluir, move-se para "terminado".

As **threads** seguem um diagrama semelhante, mas com nuances: ao terminarem as instruções na sua pilha, ficam em estado "terminated" sem afetar necessariamente o processo como um todo; ou podem ficar em "waiting" ou "timed waiting" ao aguardarem notificações de outras threads ou a expiração de temporizadores.

Compreender estes estados ajuda a diagnosticar problemas de performance: por que razão uma thread está constantemente bloqueada? Onde ocorre espera ativa ineficiente? Como podemos reduzir o tempo de comutação de contexto para melhorar o débito global?

Sincronização e Coordenação

Quando várias threads acedem aos mesmos dados partilhados, surgem as temidas **condições de corrida** (race conditions), onde o resultado depende da ordem imprevisível de interleaving das operações. Para evitar tal caos, definimos **regiões críticas**, secções de código que apenas uma thread pode executar de cada vez.

O instrumento tradicional para proteger uma região crítica é o **mutex** (ou lock), que uma thread deve "entrar" (lock) antes de entrar na secção e "sair" (unlock) quando termina. A garantia de exclusão mútua vem acompanhada de potenciais armadilhas: se duas threads ficarem cada uma à espera de um recurso que a outra detém, instaura-se um **deadlock**. São necessárias quatro condições para que um deadlock ocorra simultaneamente: exclusão mútua, hold–and–wait, sem preempção e espera circular. Para o evitar, podemos impor uma ordem rígida de aquisição de locks ou exigir que as threads peçam todos os recursos de uma só vez.

Além dos locks simples, surgem abstrações de maior nível, como **semáforos** e **monitores**.

O **semáforo** é um contador atómico associado a uma fila de espera; a operação P (down) decrementa o contador ou bloqueia a thread se for zero, e V (up) incrementa e eventualmente desbloqueia uma thread. Isso permite modelos de sincronização

mais flexíveis, como o controlo de acesso a múltiplas instâncias de um recurso (por exemplo, n cadeiras numa sala de espera).

O **monitor** combina dados privados e operações de acesso protegidas automaticamente por um lock interno. Dentro dele, as **variáveis de condição** permitem que uma thread faça wait() - libertando o lock e aguardando um signal() de outra - coordenando-se de forma estruturada. Em Java, o uso de synchronized num método ou bloco e de wait()/notify() ilustra o padrão de monitor clássico.

Para além destes, as bibliotecas modernas oferecem mecanismos avançados: barreiras de sincronização (CyclicBarrier), locks reentrantes com múltiplas filas de condição (ReentrantLock com newCondition()), variáveis atómicas (AtomicInteger, AtomicReference), e coleções concorrentes que eliminam quase por completo a necessidade de locks externos.

Conceitos Introdutórios

A arquitetura de sistemas distribuídos assenta na ideia de que múltiplos computadores autónomos, interligados por rede, cooperam de modo a apresentar-se ao utilizador como uma única entidade coerente. Esse paradigma visa tanto acelerar a execução de aplicações (via paralelização dos cálculos) como garantir elevada disponibilidade de serviços, abstraindo complexidades subjacentes como heterogeneidade de hardware, falhas pontuais ou variações de carga.

Num primeiro plano, distingue-se claramente o ambiente de **computação paralela**, em que componentes rígidos de um único sistema trabalham em proximidade tight-coupled, do **sistema distribuído**, em que cada nó tem autonomia e apenas partilha informações por **passagem de mensagens**. No modelo distribuído, não há pressupostos sobre topologia de rede ou uniformidade de plataformas: os nós podem correr sistemas operativos distintos, usar redes heterogéneas e executar software desenvolvido em linguagens variadas. O middleware, camada intermédia entre aplicação e SO, uniformiza essas diferenças, criando uma "máquina virtual" programável que oculta esta heterogeneidade.

Para que o utilizador não perceba as origens físicas dos recursos, fala-se em **transparência** em várias vertentes:

- **Acesso**: interfaces idênticas para recursos locais e remotos.
- **Localização**: independência da localização física ou na rede.
- **Replicação**: múltiplas cópias de um mesmo recurso sem alterar a perceção do utilizador.

- **Concorrência** e **falhas**: operações paralelas ou tolerância a falhas mascaradas, sem intervenção direta do utilizador.

Uma plataforma é considerada **aberta** quando permite fácil extensão e incorporação de novos serviços, desde que estes sigam APIs e padrões publicados, promovendo interoperabilidade entre fornecedores diversos.

A **segurança da informação** distribui-se em três grandes objetivos: confidencialidade (impedir acessos não autorizados), integridade (evitar corrupção ou modificações indesejadas) e disponibilidade (assegurar acesso contínuo). As soluções vão desde firewalls e encriptação de mensagens até assinaturas digitais; contudo, ataques de negação de serviço e a execução de código móvel representam desafios que exigem abordagens específicas, como deteção de anomalias e sandboxing rigoroso.

Escalabilidade é a capacidade do sistema crescer sem degradação substancial de desempenho - idealmente, um aumento linear de utilizadores não deveria tornar o overhead maior que O(log n). Isso implica evitar gargalos únicos e planear recursos de modo a antever expansões futuras, mantendo a latência e débito em patamares aceitáveis.

No domínio de **tratamento de falhas**, as estratégias principiais são mascaramento (retry automático de mensagens perdidas, replicação de dados), recuperação (checkpoints periódicos e restauro a partir do último estado consistente) e tolerância (aceitação de falhas menores com fallback de degradação de serviço, avisando o utilizador sem interromper completamente a operação).

Embora a necessidade de **concorrência** seja intrínseca - vários nós ou threads a aceder simultaneamente a recursos partilhados -, a ausência de um relógio global força a coordenação por protocolos de passagem de mensagens, algoritmos de eleição de líder e mecanismos de exclusão mútua distribuída, onde tokens ou votações asseguram que apenas uma entidade realiza alterações críticas de cada vez.

Finalmente, os emergentes paradigmas de **computação ubíqua** (Internet das Coisas) e **cloud computing** elevam estes conceitos ao extremo: no primeiro, dispositivos inteligentes comunicam de forma automática e sensível ao contexto - desde eletrodomésticos em casas conectadas até veículos autónomos em redes de tráfego cooperativo; no segundo, ganha força o modelo “serviços sob demanda”, em camadas SaaS, PaaS e IaaS, onde os utilizadores acedem a aplicações, plataformas ou infraestruturas via protocolos padronizados, vendo elasticidade quase ilimitada.

Modelos Arquiteturais

Antes de migrar para um ambiente distribuído, partimos de uma **solução concorrente** válida num sistema monoprocessador, onde múltiplos processos cooperam sobre um recurso partilhado protegido por um monitor (ou semáforos), garantindo exclusão mútua e condição de sincronização por meio de variáveis de condição. Na migração, esse recurso e os processos passam a viver em nós distintos, comunicando exclusivamente por troca de mensagens.

1. **Cliente-Servidor**

   Neste modelo, o recurso partilhado transforma-se num serviço provido por um processo servidor, enquanto os processos que o usam são clientes. A cada pedido de operação, o cliente envia uma mensagem ("request") ao servidor, que a executa localmente sobre o recurso e devolve um "reply" com o resultado.

   - O **thread base** do servidor inicializa o recurso, cria o canal de comunicação e aguarda ligações. A cada pedido, lança uma **service proxy agent** (thread) que processa a mensagem, executa a operação e responde ao cliente, depois de fechar o canal.
   - Existem três variantes principais:
     1. **Request serialization**: só um agente ativo de cada vez, simplicidade ao custo de ineficiência pela ausência de concorrência simultânea.
     2. **Server replication**: vários agentes podem correr em paralelo, usando um monitor para garantir exclusão mútua, reduzindo latências e aproveitando tempos mortos de interação.
     3. **Resource replication**: múltiplos servidores, cada um com a sua cópia do recurso, melhoram disponibilidade e escalabilidade, mas exigem mecanismos para manter a consistência das réplicas.

2. **Peer-to-Peer**
   No modelo peer-to-peer, não há distinção fixa entre clientes e servidores: cada nó (peer) possui uma réplica do serviço e pode tanto servir pedidos como solicitá-los. Para lidar com mudanças de estado global ou coordenação, costuma-se recorrer a algoritmos de eleição de líder, garantindo transições ordeiras entre estados estáveis.

3. **Publisher-Subscriber**

Aqui, múltiplos produtores de informação (publishers) e consumidores (subscribers) comunicam-se indiretamente através de um intermediário (broker). Os publishers enviam dados sobre tópicos que o broker armazena e distribui apenas aos subscribers inscritos nesses tópicos. Esse desacoplamento evita interações síncronas diretas, escalando bem em cenários de difusão de eventos (por exemplo, plataformas de trading ou serviços de notificação).

Primitivas de Comunicação

A comunicação em sistemas distribuídos apoia-se em operações básicas de **send/receive**. Normalmente, os dados são primeiro copiados para buffers do kernel antes de irem para a rede, e vice-versa. As primitivas distinguem-se em:

- **Síncronas vs. Assíncronas**: na síncrona, o send só retorna quando se garante que o receive correspondente já ocorreu; na assíncrona, o send conclui logo que os dados saem do buffer de usuário.
- **Bloqueantes vs. Não-bloqueantes**: no bloqueante, o controlo só regressa ao processo invocador após conclusão da operação; no não-bloqueante, regressa imediatamente, deixando a conclusão pendente em segundo plano.

Modelos Fundamentais

1. **Interação**

   O desempenho de um canal de comunicação é avaliado por **latência** (atraso do início do envio até o início do recebimento), **largura de banda** (quantidade total de dados por unidade de tempo) e **jitter** (variação nos tempos de entrega). Em sistemas síncronos definem-se limites conhecidos para tempos de computação, entrega de mensagens e deriva dos relógios locais; em sistemas assíncronos, esses limites existem, mas são arbitrários e não fixos.

2. **Tratamento de Falhas**

   Tanto processos como canais podem falhar. As falhas dividem-se em:

   - **Omissão**: ações que não ocorrem (por exemplo, um send que não coloca mensagem no buffer)
   - **Timing**: ações que violam prazos (por exemplo, entrega de mensagem para além do limite)

- **Arbitrárias (Byzantine)**: comportamento imprevisível, incluindo envios erráticos ou corrupções de dados
  A classificação formal destas falhas ajuda a conceber protocolos de tolerância e a definir pressupostos mínimos de confiabilidade.

3. **Segurança**

   Para proteger processos e canais, define-se um modelo onde cada operação invocada leva associada a identidade de um **principal** (utilizador ou processo) e direitos de acesso diferenciados. O servidor autentica o principal e verifica autorizações antes de executar a operação, enquanto o cliente deve confirmar a identidade do servidor para evitar respostas fraudulentas. O adversário pode eavesdroppar, modificar ou injetar mensagens, exigindo técnicas como encriptação, assinaturas digitais e sandboxing para garantir **confidencialidade**, **integridade** e **disponibilidade**.

A comunicação por passagem de mensagens em sistemas distribuídos procura oferecer ao programador um modelo simples e uniforme, ocultando as complexidades das redes físicas subjacentes. Essa abstração é tipicamente organizada em camadas, onde cada uma recebe dados da camada superior, os encapsula num formato próprio e os envia à camada inferior, revertendo o processo no destinatário para reconstituir a mensagem original.

O desempenho de um canal de comunicação baseia-se essencialmente em três métricas: **latência**, que mede o atraso desde o início do envio até ao início do recebimento; **taxa de transferência de dados**, correspondente à velocidade efetiva de transmissão; e **largura de banda**, ou seja, o volume máximo de dados que pode ser enviado por unidade de tempo. Na prática, o tempo total de transmissão de uma mensagem cumpre a fórmula

Tempo de Transmissão = Latência + (Tamanho da Mensagem / Taxa de Transferência).

Para aplicações de fluxo contínuo (por exemplo, vídeo ou voz sobre IP), exige-se que a latência se mantenha abaixo de um limite superior e a largura de banda acima de um mínimo, de modo a satisfazer os **requisitos de Qualidade de Serviço** (QoS) e evitar interrupções ou degradação percetível.

Embora as camadas inferiores (como Ethernet ou IP) incorporem já mecanismos de deteção e correção de erros - por exemplo, através de CRC ou ARQ - a filosofia do “argumento fim-a-fim” recomenda que a **confiabilidade** e a **correção de falhas** sejam, sempre que possível, geridas pelas próprias aplicações, pois estas dispõem

do contexto semântico necessário para decidir se e como reenviar, descartar ou recuperar dados.

Internamente, o software de rede está estruturado segundo o **modelo OSI**, composto por sete camadas que vão desde a aplicação até ao meio físico. Cada camada expõe uma interface de comunicação para a superior - por exemplo, a camada de transporte (TCP/UDP) para a sessão, e esta para a apresentação - garantindo independência funcional e permitindo otimizar ou substituir módulos sem afetar globalmente a aplicação.

No nível de programação, o middleware fornece o conceito de **socket** (ou endpoint), identificado por um par (endereço IP, porto). O processo abre um socket para enviar ou receber dados, operando sempre em termos de chamadas send e receive, que podem ser bloqueantes ou não, e síncronas ou assíncronas, consoante se espere pela confirmação de receção ou não.

Existem dois protocolos de transporte dominantes:

- **TCP (Transmission Control Protocol)** estabelece um canal virtual antes de qualquer troca de dados, suportando comunicação bidirecional e ordenada, ideal para aplicações que não toleram perda de pacotes ou reordenação (ex.: HTTP, SSH). Um servidor TCP cria primeiro um **listening socket** para aguardar pedidos, e a cada conexão aceita gera um **communication socket** exclusivo para aquela sessão, o que permite a cada thread ou proxy agent tratar isoladamente cada cliente.
- **UDP (User Datagram Protocol)** é "connectionless": não há estabelecimento de canal, pelo que cada mensagem (datagrama) é enviada isoladamente, sem garantia de entrega nem ordenação, mas com latência menor e overhead reduzido, adequado a aplicações de tempo real ou multicast (ex.: DNS, streaming de vídeo). O recetor cria um **receiving socket** que fica à escuta num porto, enquanto o emissor usa um **sending socket** para despachar datagramas a um ou vários destinos.

Num **fluxo TCP típico**, o cliente abre um socket, invoca connect(server_ip, port), obtém streams de entrada/saída, escreve o pedido e lê a resposta antes de fechar tudo. No servidor, o thread-base liga o listening socket, bloqueia em accept(), e ao aceitar gera um novo communication socket que, entregue a um service proxy agent, processa o pedido e envia de volta a resposta, libertando os recursos no fim.

No caso **UDP**, o emissor converte a mensagem numa matriz de bytes, embala-a num datagrama com o endereço destino e faz send(). O receptor, num ciclo de receive(), extrai os bytes, constrói a mensagem e processa-a. Não há handshaking nem controlo de fluxo, exigindo à aplicação que implemente, se necessário, mecanismos adicionais de fiabilidade ou retransmissão.

Quando um cliente quer usar um objeto remoto, em vez de obter uma referência direta (como ocorreria em memória partilhada), ele obtém primeiro um **remote reference** através de um serviço de naming. Essa referência inclui o endereço de rede do servidor, o porto onde o serviço está a escutar, as assinaturas dos métodos disponíveis e informação sobre os tipos de dados a enviar e receber. O cliente utiliza esta referência para instanciar localmente um **stub** - um proxy gerado automaticamente pelo middleware - que expõe exatamente as mesmas interfaces do objeto real, mas, internamente, converte cada invocação de método numa mensagem apropriada.

No momento da chamada a objetoRemoto.metodo(arg1, arg2), o stub serializa (marshaliza) os parâmetros, abre um canal de comunicação (por exemplo, um socket TCP), embala num "request message" o identificador do método, os parâmetros e dados de controlo, e envia para o servidor. A seguir, bloqueia (ou, em versões assíncronas, regista um callback) até receber o "reply message", que contém o resultado da invocação ou uma exceção serializada. Finalmente, o stub desserializa (unmarshaliza) o resultado e devolve-o ao programa cliente, como se fosse um retorno local.

No lado do **servidor**, existe um componente complementar, o **skeleton**, também gerado pelo ambiente de execução. O skeleton corre num thread independente do código de aplicação principal, fica à escuta no porto associado e, sempre que chega um pedido, despacha-o para um **service proxy agent**: um thread ou objeto responsável por ler a mensagem, interpretar o identificador do método, desserializar os parâmetros, e invocar o método real sobre o objeto que encapsula o recurso. Quando o método conclui, o proxy agent serializa o valor de retorno (ou a exceção), devolve o reply ao cliente e termina.

Este modelo segue três diferenças cruciais face a chamadas locais:

1. **Possibilidade de falha**: Mesmo com o código correto, a chamada pode falhar por indisponibilidade do servidor, falha de rede ou timeout. O programador deve tratar exceções de comunicação, reconectar ou reintentar conforme a política da aplicação.
2. **Passagem por valor**: Não existe partilha de memória; todos os parâmetros e retornos são clonados via marshal/unmarshal. Tipos complexos precisam implementar mecanismos de serialização (por exemplo, Serializable em Java) para garantir correção na conversão.
3. **Latência acrescida**: Cada invocação inclui tempo de rede e processamento no middleware, tornando o custo significativamente maior do que numa chamada local.

Para encontrar o objeto remoto, usa-se um **Naming Service**, que atua como diretório: o programador conhece apenas um nome symbolic (ex., "BancoDeDadosReplica1"), e o middleware pergunta ao naming service onde esse serviço está ativo, recebendo então a remote reference para criar o stub. Este desacoplamento entre nome lógico e localização física é essencial para escalabilidade e mobilidade de serviços.

Migração de Código

Um passo além de RPC é permitir que não só os dados, mas também o código se desloque entre nós: a **code migration**. Imagine querer balancear carga ou reagir a falhas: podemos serializar o estado de execução de um objeto (pilha, variáveis, referências) e transferi-lo para outro nó onde haja mais recursos disponíveis. As formas de migrar código distinguem-se por:

- **Código executável**: envia-se o binário já compilado. Rápido a executar, mas exige homogeneidade de hardware/OS.
- **Código-fonte**: envia-se o código em texto, o nó de destino compila localmente. Altamente portátil, mas mais lento e exposto a riscos de segurança se o código não for verificado.
- **Bytecode/intermediate code**: envia-se código numa forma intermediária (por exemplo, Java bytecode ou.NET IL), interpretado ou JIT compilado no destino. Bom compromisso entre portabilidade e desempenho, mas requer a máquina virtual no destino.

Para suportar a migração é preciso tratar do **estado** (serializar o contexto de execução), dos **recursos externos** (rebinding de referências, reabertura de sockets, acesso a ficheiros) e da **continuidade de execução**, garantindo consistência e evitando duplicação de efeitos. Em Java, por exemplo, o java.rmi.activation framework oferece parte destas capacidades.

Segurança

Executar código ou atender chamadas vindas de outros nós expõe-nos a ameaças: um stub ou código migrado pode tentar aceder a ficheiros sensíveis, consumir recursos ou injetar comportamentos maliciosos. A solução típica passa por inserir um **Security Manager** no servidor de objetos remotos, que intermedeia cada pedido de operação e valida, segundo políticas definidas, se a ação é permitida. Em plataformas maduras:

- Sandboxing limita chamadas a API seguras.

- Sistemas baseados em capabilities concedem explicitamente apenas as permissões mínimas necessárias.
- Assinaturas digitais e verificações de integridade garantem que só código autenticado é executado.

Nos sistemas distribuídos, a noção de tempo transcende o simples valor numérico de um relógio de cada máquina: ela sustenta a correção e a coerência de toda a comunicação e coordenação entre nós. Quando confiamos em relógios físicos (baseados em cristais de quartzo ou em sinais de satélite), deparamo-nos com dois tipos de erros inevitáveis: o **offset**, que é o desvio fixo entre o relógio local e o tempo "verdadeiro", e a **deriva** (drift), que resulta de pequenas variações na frequência do oscilador de cada máquina. Reconhecendo que não existe garantia de que uma mensagem chegue dentro de um limite máximo de atraso - afinal, em qualquer rede pode ocorrer congestionamento ou falha momentânea -, precisamos de algoritmos que ajustem os relógios de forma que, pelo menos internamente, eles permaneçam suficientemente sincronizados para que a ordem dos eventos faça sentido.

O método de Cristian é provavelmente o mais direto: cada cliente pergunta periodicamente a um servidor de tempo (considerada "referência UTC") qual é o instante atual. Ao receber a resposta, o cliente calcula o desvio médio entre o seu próprio relógio e o horário indicado, levando em conta metade do tempo de ida e volta da mensagem. Se esse intervalo for pequeno e simétrico, o ajuste deixa o relógio local bem aproximado do UTC, mantendo a monotonicidade (ou seja, nunca retrocedendo no tempo). Contudo, se a rede estiver instável, o erro de estimativa cresce e o cliente deve simplesmente ignorar essa ronda e tentar novamente mais tarde.

Quando não há um servidor UTC confiável à disposição, recorre-se ao algoritmo de Berkeley. Nele, um dos nós assume a função de "mestre" e periodicamente interroga todos os demais, incluindo a si próprio, sobre os valores atuais dos seus relógios. Após coletar essas leituras (e estimar os atrasos de cada ida e volta), o mestre calcula a média dos desvios e envia a cada nó um ajuste diferencial a ser aplicado. Como essa correção é relativa, o atraso na entrega da mensagem de volta não adiciona incerteza extra. Assim, todos os relógios do conjunto convergem para um tempo interno comum, embora possam continuar em desacordo com o UTC real.

Para lidar com a Internet como um todo, o Network Time Protocol (NTP) organiza servidores em estratos hierárquicos: no topo, servem relógios atômicos ou recetores GPS; abaixo, cada estrato sincroniza-se com o imediatamente superior. Numa troca típica, dois nós A e B trocam quatro carimbos de tempo - instantes em

que cada mensagem foi enviada ou recebida - e calculam tanto o desvio dos seus relógios quanto a assimetria do caminho. Filtrando estatisticamente várias dessas estimativas, cada cliente consegue manter o seu relógio alinhado dentro de poucas dezenas de milissegundos, mesmo atravessando várias camadas de servidores e redes heterogêneas.

Mas na maior parte das tecnologias de sistemas distribuídos a precisão do tempo físico chega a ser secundária: o que realmente importa é a ordenação lógica dos eventos. Foi Leslie Lamport quem, em 1978, observou que processos que não trocam mensagens não precisam de ter os seus relógios sincronizados de forma observável - basta que, sempre que haja comunicação, todos concordem sobre qual evento ocorreu primeiro. Com base nisso, definiu-se a relação "happened-before" ($e < e'$), que engloba três regras: a ordem sequencial dentro de cada processo, a causalidade de envio-receção de mensagens e a transitividade.

Para tornar essa relação numérica, cada processo mantém um contador local - o **relógio lógico escalar** - que avança um passo a cada evento interno e sempre que envia ou recebe uma mensagem. Ao enviar, ele anexa o valor do seu contador à mensagem; ao receber, ele atualiza o seu contador para o máximo entre o próprio e o timestamp recebido, e só depois incrementa. Assim, se um evento e precede causalmente outro e', garantimos que o timestamp de e seja menor que o de e'. A convenção de estender o timestamp com o identificador do processo (formando pares lexicográficos) resolve empates e permite ordenar totalitariamente todos os eventos que envolvam troca de mensagens.

Apesar dessa elegância, os relógios escalares não capturam totalmente a noção de concorrência: se dois eventos em processos distintos nunca trocaram informação, nada impede que acabem por ter timestamps comparáveis, sugerindo falsamente uma relação causal. A solução são os **relógios vetoriais**, onde cada processo pi guarda um vetor de N entradas - uma para cada processo do sistema. No seu próprio evento, pi incrementa apenas a sua componente; ao enviar, anexa o vetor inteiro; ao receber, funde componente a componente o vetor local com o recebido (tomando o máximo), e só então incrementa a sua própria posição. Deste modo, um vetor V é estritamente anterior a outro V' se, e só se, todos os componentes de V forem menores ou iguais aos de V' e existir pelo menos um estritamente menor. Isso significa que podemos decidir com segurança se $V < V'$ (e precedeu e'), ou se são concurrentes (incapazes de comparação), capturando integralmente a causalidade do sistema.

Em conjunto, esses mecanismos-tanto de sincronização de relógios físicos quanto de definições de tempo e ordenação lógica-formam a espinha dorsal dos sistemas distribuídos, garantindo que, mesmo sem um relógio global perfeito, possamos coordenar, replicar e tolerar falhas de forma consistente e previsível.

Nos sistemas distribuídos, a comunicação em grupo assume um papel central sempre que vários processos independentes devem cooperar para aceder a recursos partilhados ou tomar decisões coordenadas. Diferentemente do modelo cliente-servidor, onde existe uma entidade central com privilégios especiais, aqui todos os participantes são pares que se comunicam por troca de mensagens sem partilharem memória comum, o que obriga a protocolos específicos para garantir exclusão mútua, ordenação de eventos e eleição de líderes.

Em primeiro lugar, a forma mais simples de controlar o acesso partilhado consiste em delegar esse papel a um "guardião" central. Cada processo que deseja aceder ao objeto envia-lhe uma mensagem de pedido, espera o consentimento e, depois de usar o recurso, envia uma mensagem de libertação. Embora fácil de implementar, este método sofre de elevado overhead de mensagens (três interações por acesso) e torna o sistema vulnerável a falhas do guardião, que se transforma num ponto único de falha.

Para eliminar esse ponto único, podemos dispor os processos em anel lógico e usar a passagem de um "token". Só o processo que detém o token pode entrar na região crítica, e ao terminar passa-o ao vizinho seguinte no anel. Esse mecanismo reduz a troca de mensagens a uma por fase de token (independentemente de uso), mas em grupos grandes pode gerar atrasos significativos até que o token percorra todo o anel, mesmo quando ninguém precisa usar o recurso.

Uma abordagem mais sofisticada, proposta por Ricart e Agrawala, baseia-se na total ordering de pedidos usando relógios lógicos de Lamport. Cada pedido é multicastado com um timestamp; todos os processos respondem imediatamente com a sua autorização, exceto se estiverem na sua seção crítica ou aguardando um pedido de ordem anterior, caso em que enfileiram as solicitações. Só após receber N–1 autorizações é que o processo entra na região crítica. Este protocolo exige 2(N–1) mensagens por acesso, sendo eficiente em grupos pequenos, mas pouco escalável quando N cresce demasiado.

Para cortar ainda mais no número de mensagens, Maekawa introduziu o modelo de votação por subgrupos. Cada processo pertence a um pequeno conjunto de eleitores e só precisa de recolher permissões desse subconjunto para aceder ao recurso. Escolhendo os grupos de modo que todos se intersectem (por exemplo, utilizando uma disposição matricial $\sqrt{N}\times\sqrt{N}$), o número de mensagens desce para $O(\sqrt{N})$ por acesso, o que melhora substancialmente o desempenho em grandes sistemas. Contudo, como cada voto é exclusivo, surgem riscos de deadlock em cenários de acesso concorrente intenso, que só se resolvem introduzindo ordenação de pedidos (usando timestamps) ou mecanismos de fallback centralizado/token.

Além da exclusão mútua, a comunicação em grupo inclui procedimentos para eleição de um líder quando uma tarefa única precisa de coordenação exclusiva. No anel lógico, o algoritmo de Chang-Roberts permite que qualquer processo inicie a eleição enviando o seu ID ao vizinho. Cada nó, ao receber um ID menor que o seu, o reenvia; se for maior, substitui pelo seu. Quando uma mensagem retorna com o mesmo ID do iniciador, este torna-se líder e propaga uma mensagem de anúncio até fechar o ciclo. Em ambientes com falhas, é preciso adicionar timeouts, esquemas de reconhecimento (ACK) e saltos sobre nós caídos para manter o anel funcional.

Em grupos não estruturados, onde não há topologia rígida, as eleições seguem a ideia de Garcia-Molina: um processo deteta a falta de líder ou inicia voluntariamente, envia mensagens de evento "start" para processos de menor ID, espera acknowledgments e, se nenhum responder num prazo definido, considera-se o de menor ID ativo e assume o papel. Depois, anuncia o resultado a todos. Para lidar com perda de mensagens e falhas, introduzem-se ACKs, retransmissões, números de ronda para evitar eleições duplicadas e serviços de "membership" que mantêm todos atualizados sobre quem está vivo no sistema.

Em suma, a comunicação em grupo em sistemas distribuídos equilibra três dimensões: complexidade de mensagens, tolerância a falhas e rapidez de resposta. Desde soluções centralizadas, passando por anéis de token e ordering total com relógios lógicos, até protocolos de voto em subgrupos e sofisticados algoritmos de eleição, cada abordagem pondera diferentes trade-offs entre sobrecarga de comunicação, pontos de falha e capacidade de escalar aos milhares de nós. Compreender estas técnicas é essencial para projetar sistemas robustos, eficientes e consistentes que funcionem em ambientes heterogéneos e sujeitos a falhas.

Para compreender como sistemas distribuídos mantêm dados replicados de forma correta e eficiente, é essencial começar por ver uma "região de armazenamento distribuído" como uma memória ou base de dados que existe em várias máquinas ao mesmo tempo. Cada nó de processamento tem acesso direto à sua cópia local desse conjunto de dados e pode executar operações de escrita (que modificam valores) ou de leitura (que apenas consultam valores). Quando um processo faz uma escrita, essa alteração tem de chegar a todas as réplicas; já as leituras podem ou não ver imediatamente as escritas feitas noutros nós, consoante o modelo de consistência pretendido.

O objetivo de cada modelo de consistência é definir um "contrato" que garanta que as leituras retornem valores plausíveis face às escritas ocorridas, mesmo quando estas ocorrem em paralelo. Para isso, imaginamos sempre uma sequência global

de operações - virtual, pois jamais efetivamente formada - que intercale todas as escritas e leituras dos diversos processos. Se, para um dado conjunto de acesso concorrente, não existir nenhuma dessas sequências canónicas que explique os resultados observados, dizemos que o modelo foi violado.

Consistência Estrita

No modelo ideal de consistência estrita, cada leitura devolve sempre o valor gerado pela escrita mais recente em todo o sistema. Isto pressuporia um único relógio global, onde cada evento tivesse um instante bem definido e comparável. Porém, em sistemas distribuídos não é possível sincronizar perfeitamente relógios físicos nem propagar sinais com latência nula, tornando a noção de "mais recente" ambígua. Consequentemente, a consistência estrita só existe teoricamente em sistemas monoprocessador e não é praticável em arquiteturas distribuídas.

Linearizabilidade

A linearizabilidade aproxima-se do ideal da consistência estrita ao assumir que existe um mecanismo de sincronização (através de relógios lógicos, por exemplo) que permite ordenar atomically todas as operações. Cada operação é vista como se ocorresse instantaneamente num ponto único entre a sua invocação e conclusão, e esse ponto define a ordem global. Assim, qualquer leitura sempre vê o valor mais atual, de acordo com essa ordenação que respeita a ordem real percebida localmente pelos processos. Para implementar esta propriedade, costuma usar-se um protocolo de difusão e confirmação de operações por todos os nós, apoiado em relógios lógicos de Lamport, garantindo que nenhum evento seja efetivado sem antes ter a certeza de que todos os outros nós concordam na sua posição na sequência global.

Sistemas que exigem forte consistência e visibilidade imediata - como plataformas financeiras, bases de dados transacionais ou controlos de infraestruturas críticas - beneficiam da linearizabilidade, apesar do seu custo em desempenho e complexidade de coordenação.

Consistência Sequencial

Mais fraco que a linearizabilidade, o modelo de consistência sequencial não se preocupa com tempos reais nem com pontos atómicos, apenas exige que exista alguma sequência global de operações onde as escritas e leituras de cada processo mantenham necessariamente a ordem local em que foram emitidas. Em outras palavras, não há necessidade de respeitar o instante cronológico em que cada operação ocorreu, apenas que os efeitos observados por todos os nós possam ser explicados por uma interleaving onde as operações de cada processo aparecem por ordem. Para o conseguir, basta garantir que todas as escritas sejam difundidas e confirmadas antes de se tornarem visíveis; as leituras podem servir-se

de cópias locais sem coordenação adicional. Plataformas de comércio eletrónico, sistemas de reserva e edição colaborativa usam frequentemente esta forma de consistência, que equilibra razoável coerência global com menor sobrecarga de sincronização.

Consistência Causal

Entre modelos que reconhecem dependências sem exigir total ordenação, a consistência causal define que quaisquer operações que sejam diretamente relacionadas - seja porque foram emitidas pelo mesmo processo em sequência, seja porque uma leitura viu o valor de uma escrita anterior - devem manter essa relação de causa-efeito em todas as réplicas. Por outro lado, operações que não têm nenhum vínculo causal podem ser vistas em qualquer ordem. A implementação recorre a relógios vetoriais: cada nó mantém um vector de contadores (um por cada processo), que serve para incluir nas mensagens a informação necessária para que cada receptor só torne visíveis escritas cujas causas já foram incorporadas localmente. Este modelo é ideal para chats, redes sociais ou editores colaborativos, onde importa sobretudo que as respostas ou comentários apareçam depois das mensagens a que se referem.

Consistência FIFO

Finalmente, o modelo mais fraco fala apenas de garantias de ordenação por emissor: cada processo vê as escritas de qualquer outro processo sempre na ordem em que foram enviadas, mas sem qualquer obrigatoriedade de ordem entre escritas emitidas por processos distintos. Não há sequer uma sequência global canónica; basta que, para cada par emissor-receptor, as mensagens cheguem conforme a sequência numérica atribuída pelo emissor. Este método é simples de implementar e útil em cenários como distribuição de notícias, dados meteorológicos ou streaming multimédia, onde é aceitável ver atualizações de fontes diferentes em ordens ligeiramente discrepantes.

Compreender estes diferentes níveis de consistência é crucial para desenhar sistemas distribuídos que equilibrem corretude, desempenho e disponibilidade, escolhendo o modelo que melhor se adapta às necessidades da aplicação.

Quando falamos de transações em sistemas distribuídos, imaginamos um conjunto de operações de leitura e escrita - por exemplo, “ler o saldo da conta X” ou “debitar 50 € de Y” - agrupadas num único bloco lógico que deve ser tratado pelo sistema como atómico: ou todas as operações desse bloco ocorrem com sucesso, ou nenhuma delas se concretiza. Do lado do servidor, há sempre um coordenador que, ao abrir uma nova transação, atribui-lhe um identificador único e passa a registar cada operação associada. Se, em algum momento, uma das operações

falhar - seja por erro local, violação de invariantes ou timeout - o servidor aborta toda a transação e reverte quaisquer alterações intermédias, assegurando o princípio "tudo ou nada" (atomicidade).

Além da atomicidade, temos outras propriedades críticas, resumidas pelo acrónimo ACID. A consistência ("C") garante que passar de um estado válido do sistema para outro também será sempre válido - por exemplo, não podemos deixar que o saldo de uma conta fique negativo se isso violar políticas internas. A isolação ("I") assegura que duas transações concorrentes não vejam os estados intermédios uma da outra, para que cada cliente perceba o sistema como se fosse unicamente seu. Por fim, a durabilidade ("D") implica que, uma vez confirmada a transação, os seus efeitos persistam mesmo que haja falhas de energia ou crashes do servidor.

Quando o sistema se alarga a vários nós, cada parte dos dados pode residir num servidor diferente. A grande questão passa então a ser: como coordenar um commit que envolva fragmentos de dados espalhados? A solução clássica é o protocolo de duas fases (2PC). Na primeira fase, o coordenador envia um pedido de voto a cada participante, que apenas responde "posso confirmar" (voteCommit) ou "não posso" (voteAbort). Só depois de recolher votos favoráveis de todos é que, na segunda fase, o coordenador envia a decisão final de commit. Caso contrário, manda abortar globalmente. Cada participante, ao receber essa decisão, executa o respetivo commit local ou rollback.

Este mecanismo, embora simples de explicar, deixa os participantes num estado de bloqueio se o coordenador falhar no momento entre o voto e a decisão final: sem saber se deve confirmar ou abortar, um servidor fica à espera indefinidamente, retendo bloqueios sobre recursos. Para contornar este problema surgiu o protocolo de três fases (3PC). Depois do voto inicial, há uma fase intermédia de "pré-commit" em que o coordenador informa os participantes de que todos aceitaram e pede-lhes que entrem num estado preparado, mas ainda não confirmem. Só após receber respostas afirmativas de todos é enviada a ordem de commit definitivo. Graças a este passo extra, mesmo que o coordenador falhe, os participantes podem consultar uns aos outros e decidir sem bloqueios permanentes - embora isto acrescente complexidade e overhead, razão pela qual o 3PC é raro em ambientes de produção.

Outra abordagem para lidar com falhas e melhorar modularidade são as transações aninhadas. Aqui, a transação principal dispara subtransações que podem executar-se em máquinas distintas, cada qual com o seu commit local tentativo. Se a transação de nível superior falhar, todas as subtransações são abortadas em cascata; se for bem-sucedida, então todos os commits locais tornam-se definitivos. Isto permite paralelizar e isolar partes de um mesmo fluxo de trabalho sem comprometer a coerência global do sistema.

Para evitar conflitos de concorrência - como duas transações tentarem escrever simultaneamente no mesmo registo - usam-se esquemas de bloqueio: antes de uma operação de escrita, obtém-se um lock exclusivo; para leitura, pode-se usar locks partilhados. Só depois de confirmado o commit é que esses bloqueios são libertados, garantindo serializabilidade e impedindo atualizações perdidas ou leituras sujas.

Compreender estes mecanismos é fundamental para desenhar sistemas distribuídos que equilibrem desempenho, escalabilidade e correção. Conforme as necessidades da aplicação - seja forte consistência imediata, alta disponibilidade sem bloqueios, ou suporte a longas cadeias de operações complexas - escolheremos o protocolo e o modelo de transação mais adequado.