Departamento de Informática e Matemática Aplicada – DIMAp

Introdução a Sistemas Distribuídos

Prof. Nélio Cacho

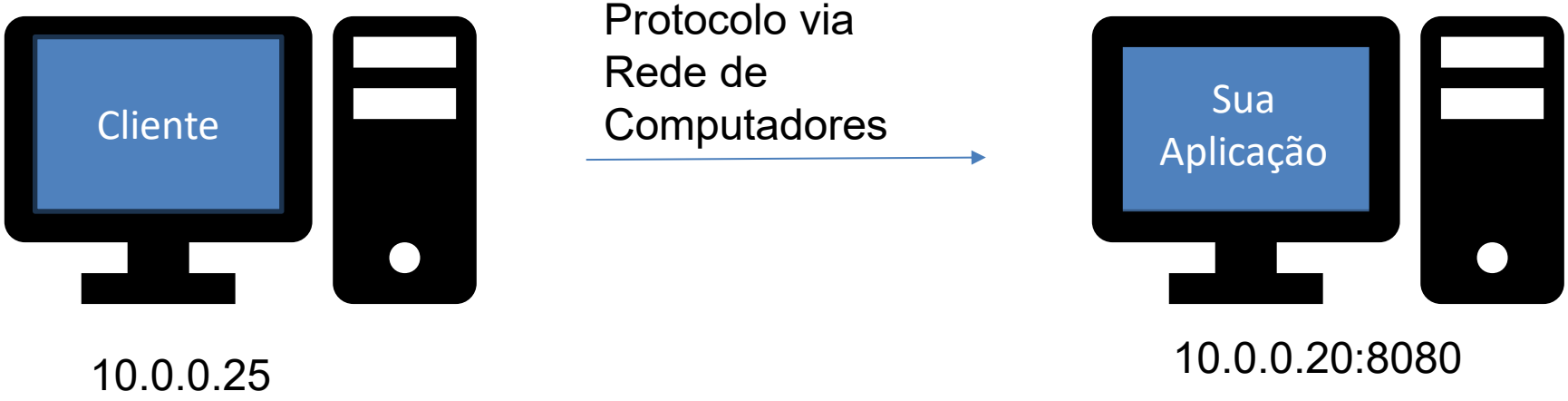
Universidade Federal do Rio Grande do Norte
neliocacho@dimap.ufrn.br

Hora de silenciar o celular...

- Manter o telefone celular sempre desligado/silencioso quando estiver em sala de aula;
- Nunca atender o celular na sala de aula.

Na aula passada, vimos que saímos de uma aplicação Standalone para ...

... um sistema distribuído formado por Cliente e Servidor



Anatomia de um Sistema Distribuído

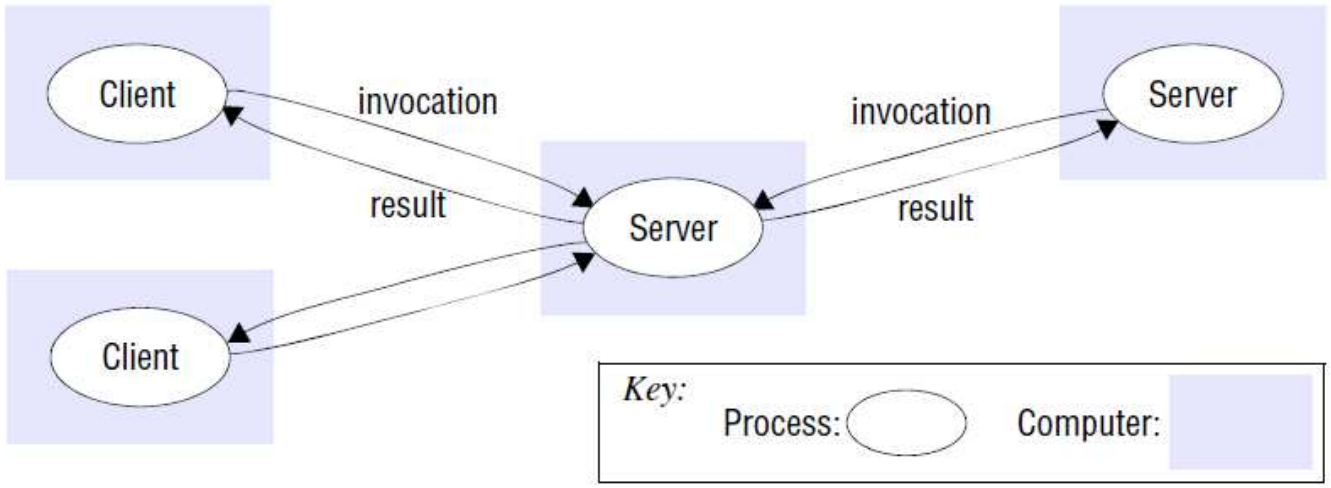
- Em sistemas distribuídos, **System Models** são modelos usados para descrever um sistema distribuído.
- Esses modelos ajudam a guiar o projeto, a escolha de algoritmos e a forma de avaliar desempenho e tolerância a falhas.
- Os principais **System Models** são:
  - Modelos de Arquitetura (Architectural Models)
  - Modelos de Interação (Interaction Models)
  - Modelos de Falha (Failure Models)

Modelos de Arquitetura (Architectural Models)

- Definem como os componentes do sistema distribuído estão organizados, como interagem e como os recursos são distribuídos.
- A escolha da arquitetura define:
  - Complexidade de coordenação.
  - Estratégias de balanceamento de carga.
  - Possibilidades de escalabilidade e resiliência.
- Principais variações:
  - Cliente–Servidor
  - Peer-to-Peer (P2P)
  - Modelo de filtros (*pipes-and-filters*)
  - Publicação–subscrição (Publish/Subscribe)
  - Microservices / Service-Oriented Architecture (SOA):

Modelo cliente-servidor (*client-server*)

- Dois tipos de entidades, **cliente** e **servidor**
  - Clientes **realizam requisições** a servidores solicitando **serviços** providos por estes
  - Servidores **respondem a requisições** feitas por clientes
- Servidores podem ser por ora **clientes de outros servidores**



Modelo cliente-servidor (*client-server*)

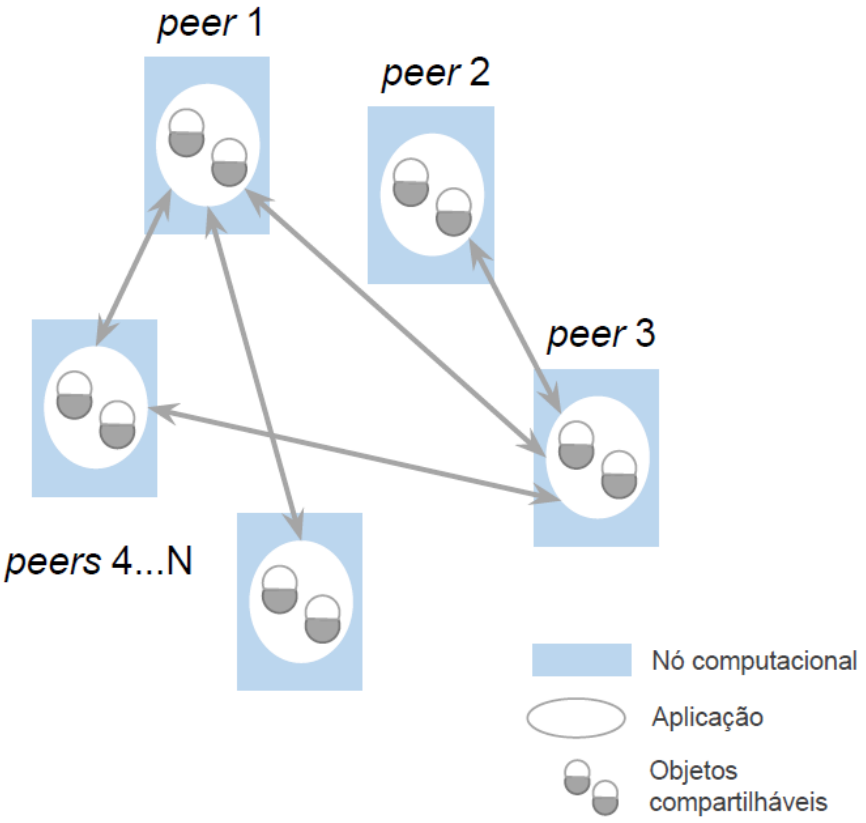
- O esquema de comunicação é baseado no paradigma **requisição-resposta** (*request-reply*)
- O cliente fica aguardando até que o servidor responda a sua requisição, podendo incorrer em desperdício de recursos computacionais e tempo
- **Transparência de localização**: clientes comunicam-se com servidores através da rede, porém sem conhecimento da localização destes
- O modelo cliente-servidor é bem indicado para aplicações distribuídas **não paralelas e síncronas**

Modelo cliente-servidor (*client-server*)

- Servidores podem ou não armazenar estado (***stateful servers*** *vs.* ***stateless servers***)
- **Serviços Stateless**: Não mantêm estado de sessão ou histórico do cliente. Cada requisição é processada de forma independente.
  - Vantagem: Fácil de escalar e recuperar de falhas, pois uma instância pode ser substituída a qualquer momento sem perda de dados. Exemplo: Uma API de consulta de preço que não armazena o histórico do usuário.
- **Serviços Stateful**: Mantêm informações sobre o estado da sessão entre as requisições. Requerem armazenamento persistente (banco de dados, cache em memória) para operar.
  - Desafio: A recuperação de falhas é complexa, pois a nova instância precisa recuperar o estado anterior. A falha pode resultar em perda de dados. Exemplo: Um serviço de carrinho de compras que armazena os itens de um cliente.

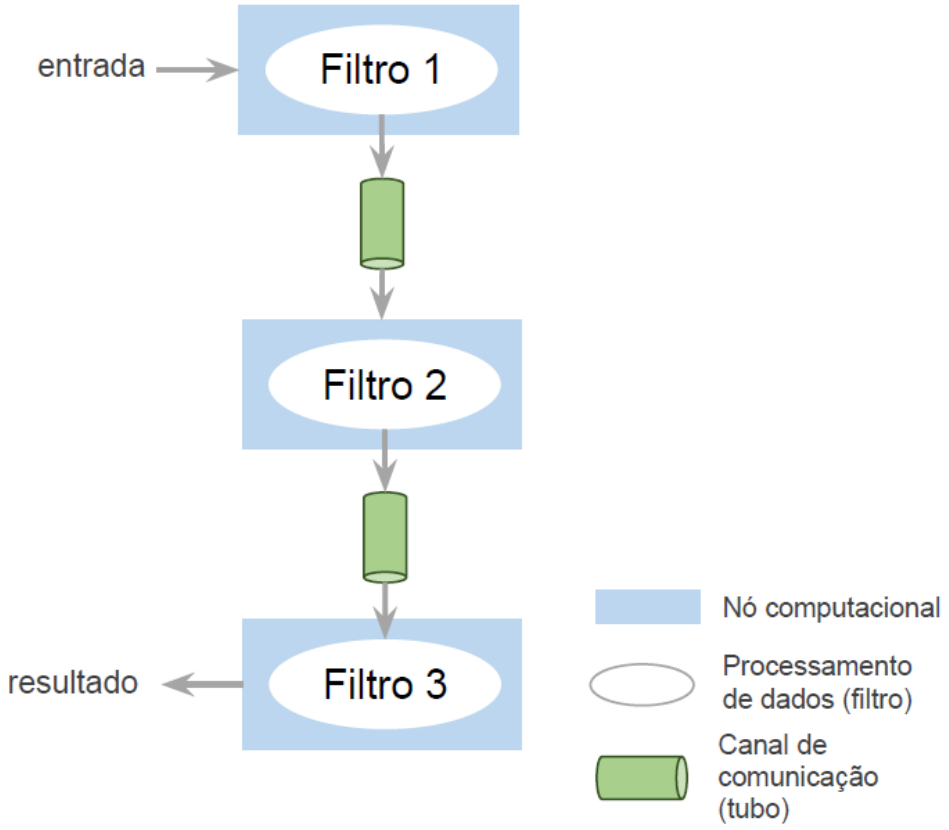
Modelo em pares (*peer-to-peer)*

- Alternativa ao modelo cliente-servidor na qual todas as entidades podem ser **clients ou servidoras umas das outras**
- Tipicamente envolve o **compartilhamento de objetos** entre as entidades
  - Todos os *peers* necessariamente proveem **os mesmos serviços** (simetria)



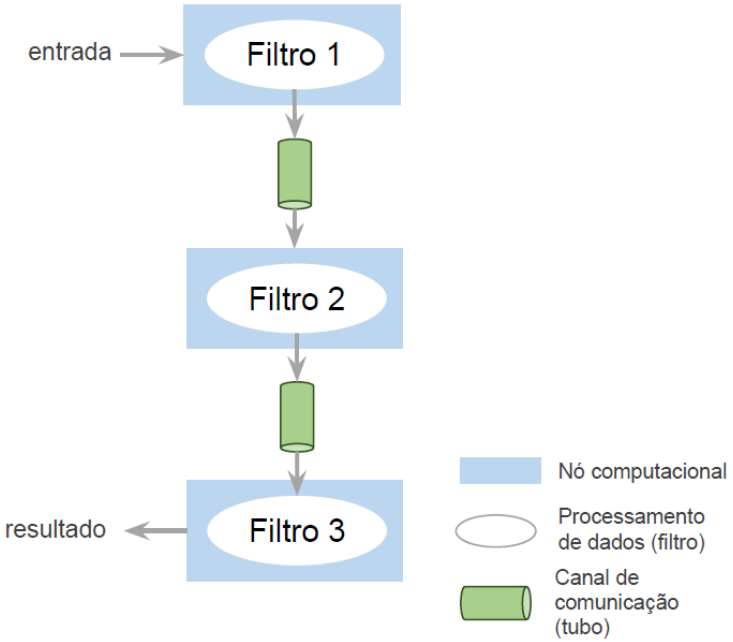
Modelo de filtros (*pipes-and-filters*)

- Modelo em que processos distribuídos realizam o processamento de uma ou mais entradas e dispõem o resultado em uma saída, havendo canais de comunicação entre eles



Modelo de filtros (*pipes-and-filters*)

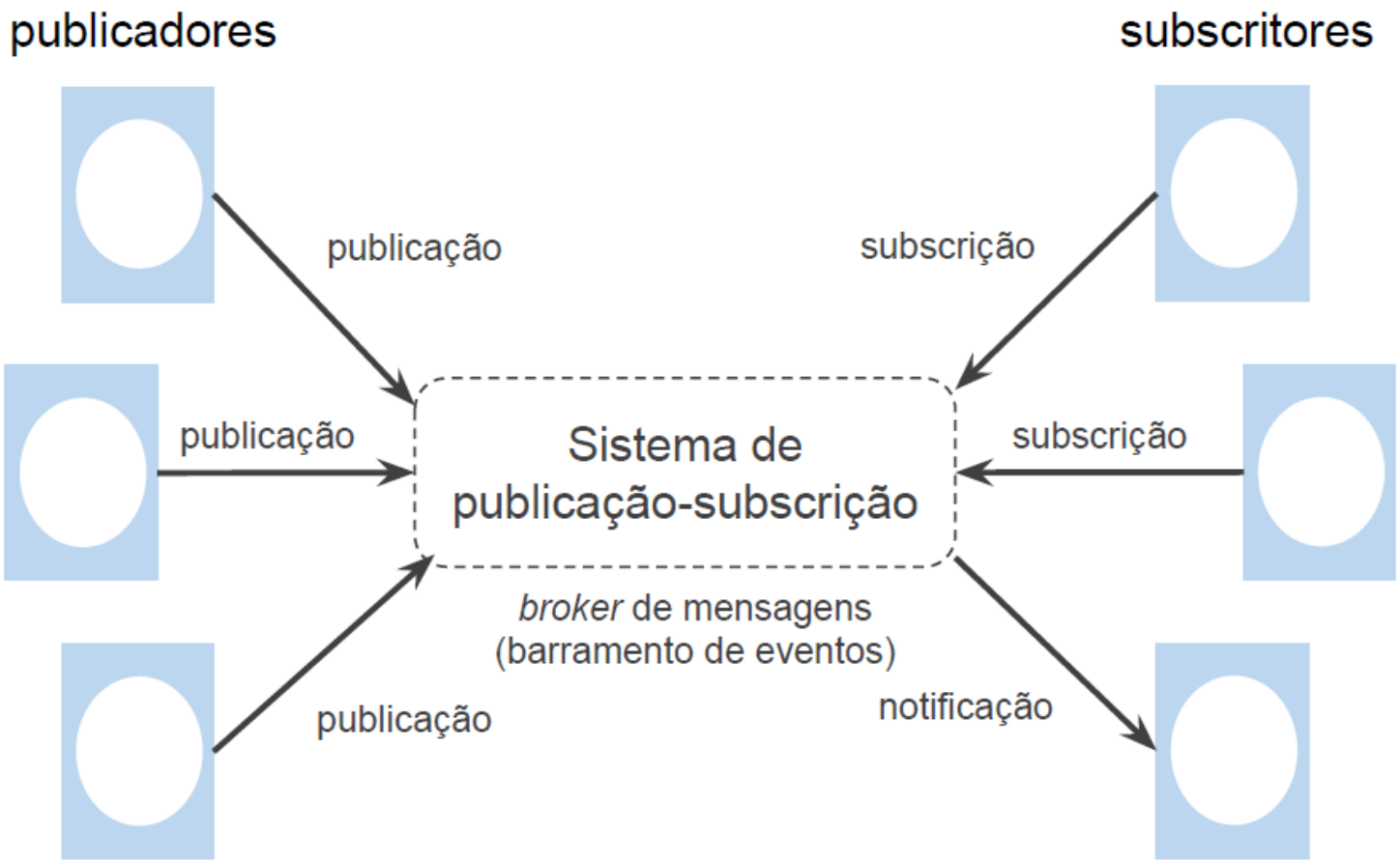
- Entidades:
  - **Filtro** (*filter*): componente que realiza um estágio do processamento, transformando os dados de entrada
    - Possibilita reuso, separação de responsabilidades (*separation of concerns*) e divisão da carga de trabalho
  - **Tubo** (*pipe*): conector através do qual dados são transportados de um filtro para outro



Modelo publicação-subscrição (*publish-subscribe*)

- Modelo de **comunicação assíncrona** composto por **publicadores** (*publishers*) e **subscritores** (*subscribers*)
- Publicadores enviam mensagens para serem consumidas por subscritores de acordo com seu **interesse**
  - A publicação de informação por parte de publicadores produz **notificações** (eventos)
  - **Subscrições** manifestam o interesse de subscritores com relação à(s) informação(ões) em questão
- Existência de um ***broker* de mensagens** (barramento de eventos) para gerenciar publicações, subscrições e notificações

Modelo publicação-subscrição (*publish-subscribe*)



Modelo publicação-subscrição (*publish-subscribe*)

- Subscrições são **indexadas** e as publicações que chegam são comparadas com as subscrições que foram armazenadas
  - As **subscrições são contínuas**: subscritores são notificados de publicações enquanto não manifestarem o cancelamento de seu interesse por publicações
- Publicadores **não possuem qualquer conhecimento** dos possíveis subscritores interessados em consumir as mensagens
  - **Desacoplamento espacial**: anonimato
  - **Desacoplamento temporal**: publicadores e subscritores possuem ciclos de vida independentes

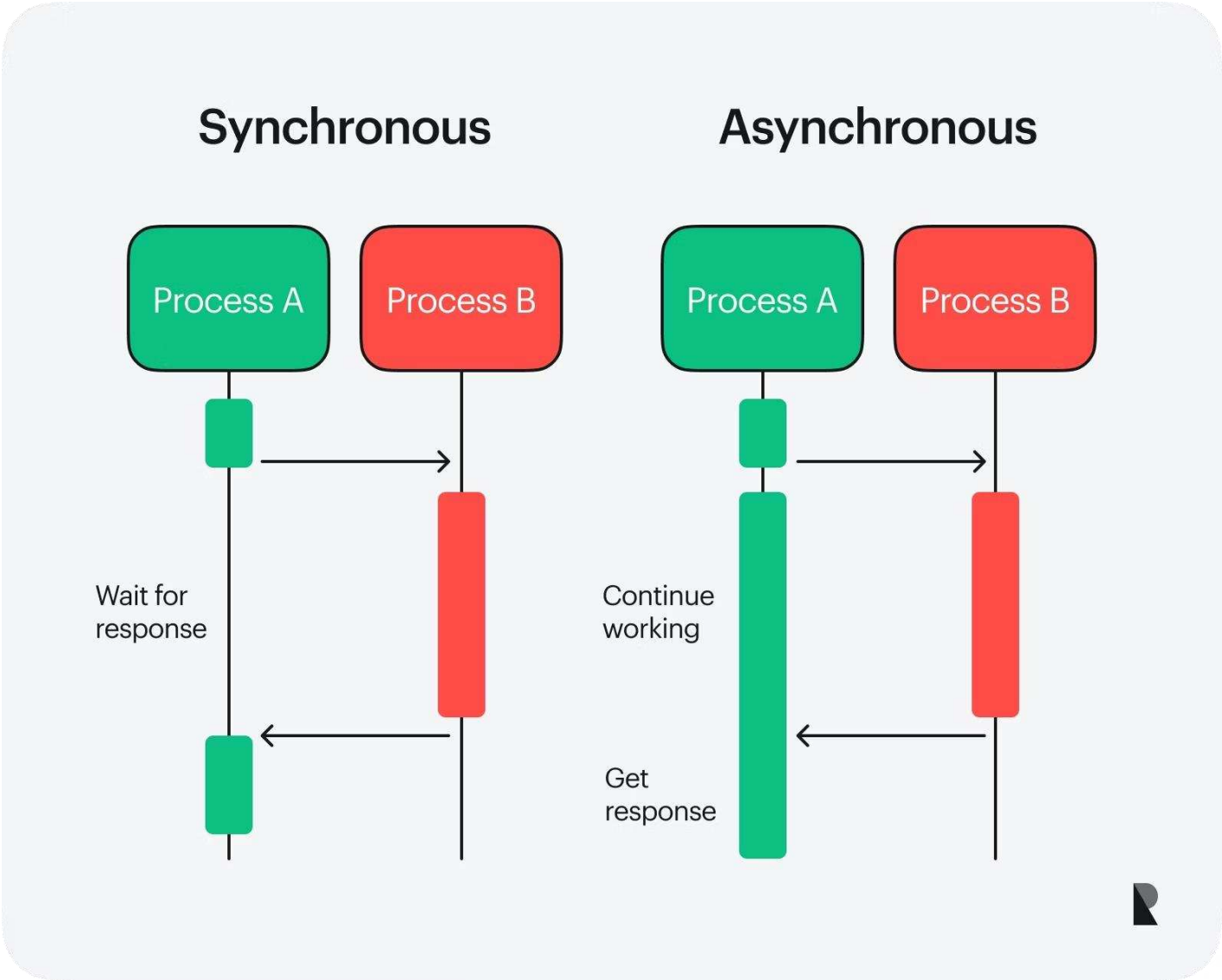
Microserviços

- A aplicação é dividida em serviços **independentes, pequenos e autocontidos**, cada um responsável por uma funcionalidade específica.
- Cada Microserviço é autocontido, sendo responsável por um **conjunto de funcionalidades bem definidas.**
- **Escalabilidade horizontal**: cada serviço pode ser replicado separadamente conforme demanda.
- **Gerenciamento distribuído**: exige orquestração (Kubernetes, Docker Swarm) e monitoramento.

Modelos de Interação (Interaction Models)

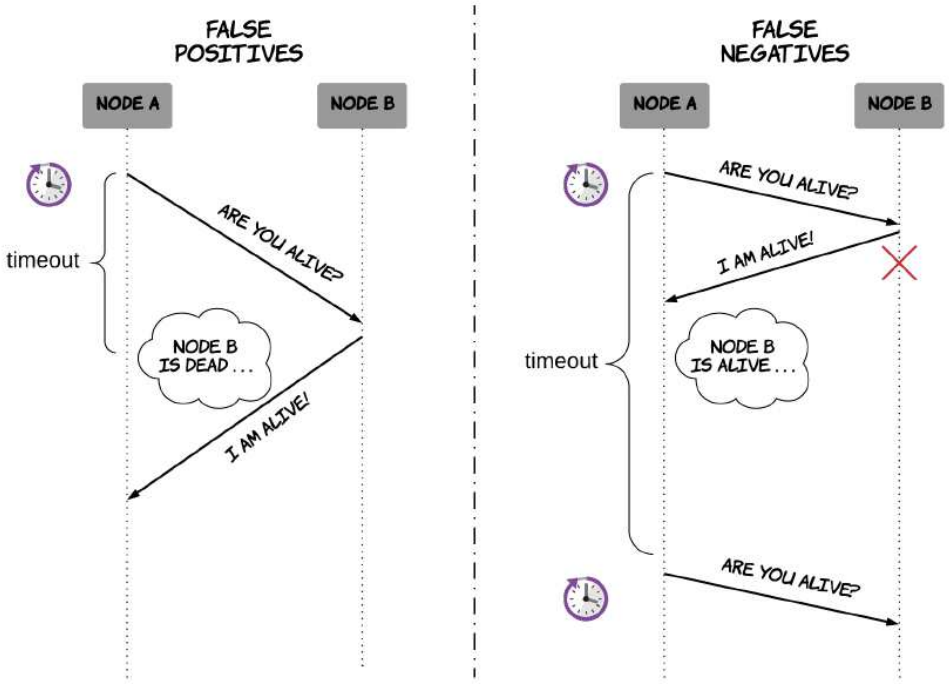
- Descrevem as restrições temporais e as garantias de entrega das mensagens, afetando diretamente os algoritmos usados.
- Três categorias principais:
  - **Modelo Sincrônico**: Tempos máximos de processamento e comunicação são conhecidos e garantidos. Clocks podem ser perfeitamente sincronizados.
  - **Modelo Assíncrono**: Sem garantias de tempo. Mensagens podem atrasar indefinidamente. Clocks podem divergir completamente.

Modelos de Comunicação



Falhas em Sistemas Distribuídos

- Sistemas distribuídos são de natureza assíncronos;
- Mensagens podem atrasar indefinidamente.
- Difícil distinguir entre:
  - Nó que travou (crash)
  - Nó que está apenas lento
- A detecção de falhas depende de tempos limites (timeouts) para tomar decisões.

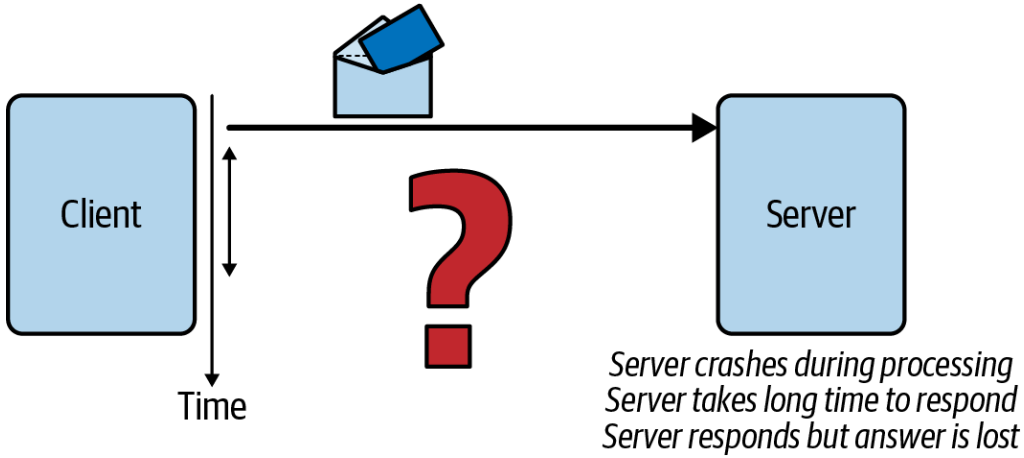


Falhas em Sistemas Distribuídos

- Quando um cliente deseja se conectar a um servidor e trocar mensagens, os seguintes resultados podem ocorrer:
  1. **A solicitação é bem-sucedida** e uma resposta rápida é recebida. Tudo está bem. (Na realidade, esse resultado ocorre na maioria das solicitações. *Quase* é a palavra-chave aqui.)
  2. **A consulta ao endereço IP de destino pode falhar.** Nesse caso, o cliente recebe rapidamente uma mensagem de erro e pode agir de acordo.
  3. **O endereço IP é válido**, mas o nó de destino ou o processo do servidor de destino falhou. O remetente receberá uma mensagem de erro por tempo excedido (*timeout*) e poderá informar o usuário.
  4. **A solicitação é recebida pelo servidor de destino**, mas este falha durante o processamento da solicitação e nenhuma resposta é enviada.
  5. **A solicitação é recebida pelo servidor de destino**, que está sobrecarregado. Ele processa a solicitação, mas leva muito tempo (por exemplo, 34 segundos) para responder.
  6. **A solicitação é recebida pelo servidor de destino** e uma resposta é enviada. No entanto, a resposta não é recebida pelo cliente devido a uma falha na rede.

Falhas em Sistemas Distribuídos

- Os três primeiros pontos são fáceis para o cliente lidar, pois uma resposta é recebida rapidamente. Um resultado do servidor ou uma mensagem de erro — ambos permitem que o cliente prossiga. Falhas que podem ser detectadas rapidamente são fáceis de tratar.
- O restante dos resultados representa um problema para o cliente. Eles não fornecem qualquer informação sobre o motivo pelo qual uma resposta não foi recebida. Do ponto de vista do cliente, esses três resultados parecem exatamente iguais. O cliente não pode saber, sem esperar (potencialmente para sempre), se a resposta eventualmente chegará ou nunca chegará; esperar para sempre não resulta em muito trabalho realizado.



Uso de Timeouts

- **Timeouts** impõem um limite artificial para a espera de respostas.
- Se um nó não responde dentro do timeout, assume-se que ele falhou.
- Evita que o sistema bloqueie indefinidamente esperando nós lentos ou inativos.
- Trade-off na escolha do timeout:
  - *Timeout pequeno*: Detecta falhas mais rápido. Pode marcar nós lentos como falhos.
  - *Timeout grande*: Mais tolerante a nós lentos. Detecta falhas mais devagar, podendo esperar nós já mortos.

Modelos de Falha (Failure Models)

- Classificam os tipos de falhas que podem ocorrer e que o sistema deve tolerar.
- Principais tipos:
  - Crash Failure
  - Omission Failure
  - Timing Failure
  - Response Failure
  - Byzantine Failure

Crash Failure

- O processo ou nó para de funcionar abruptamente e não responde mais.
- Precisamos manter dados e estado mesmo com falha de nós.
- Exemplo: Um servidor de banco de dados que trava e deixa de atender requisições.
- Ligação com os Padrões
  - Data Replication (write-ahead log): garante que dados não se percam.
  - Cluster Management (Consistent Core): detecta nós indisponíveis e mantém quorum.
  - Data Partitioning (Two-phase commit): coordena transações mesmo se um nó cair no meio do processo.

Omission Failure

- Definição: Mensagens ou respostas são perdidas ou não enviadas/recebidas.
- Tipos:
  - Send Omission: Mensagem não é enviada.
  - Receive Omission: Mensagem não é recebida.
- Exemplo: Pacotes TCP perdidos em uma rede congestionada.
- Impacto: Necessidade de retransmissão, confirmação de recebimento e logs de operação.
- Ligação com os Padrões
  - Distributed Time (Lamport Clock): detecta e ordena eventos mesmo com mensagens ausentes.
  - Cluster Management (Lease): controla validade de líderes quando mensagens se perdem.
  - Replication: retransmite dados quando réplicas não receberam atualizações.

Timing Failure

- Definição: Resposta chega fora do prazo esperado.
- Exemplo: Um serviço que responde depois de 10 segundos quando o cliente espera 1 segundo.
- Ligação com os Padrões**:**
  - Hybrid Clock: combina tempo lógico e físico para detectar atrasos.
  - Lease: expira liderança quando respostas atrasam demais.
  - Gossip Dissemination: propaga estado de forma resiliente a atrasos.

Response Failure

- **Definição:** Resposta errada é retornada devido a erro interno do processo.
- **Tipos:**
  - *Value Failure:* Valor incorreto retornado.
  - *State Transition Failure:* Sequência de operações inválida.
- **Exemplo:** Função de cálculo de saldo que retorna valor errado após uma atualização de registro.
- Ligação com os Padrões
  - Two-phase commit: evita commits inconsistentes.
  - State Watch: monitora consistência de estado.
  - Write-ahead log: mantém histórico para recuperação.

Byzantine Failure

- **Definição:** Comportamento arbitrário ou malicioso; nó pode enviar respostas contraditórias ou falsas.
- **Exemplo:** Nó comprometido em blockchain que envia transações inválidas para diferentes nós.
- **Impacto:** Requer protocolos de consenso tolerantes a falhas bizantinas (PBFT), quórum confiável e verificação de integridade das mensagens.
- Ligação com os Padrões:
  - Consistent Core: requer algoritmos tolerantes a bizantinos (ex.: PBFT).
  - Gossip Dissemination: detecta divergências de estado.
  - Replication com validação: compara múltiplas réplicas para detectar comportamento incorreto.

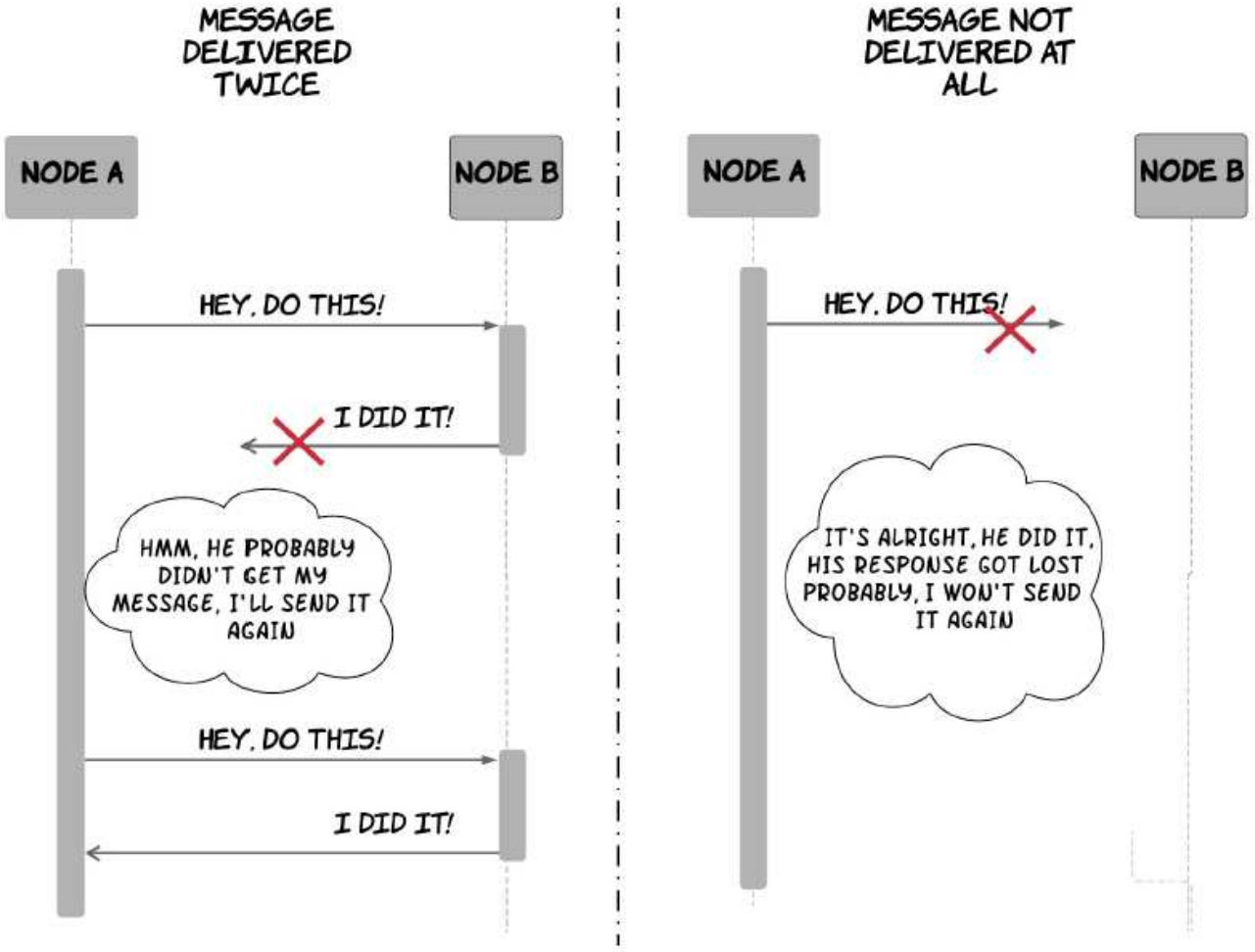
Modelos de Falha vs Padrões

Padrão	Crash	Omission	Timing	Response	Byzantine
Write-Ahead Log	1	0	0	0	0
Segmented Log	1	0	0	0	0
Low-Water Mark	1	1	0	0	0
Leader and Followers	1	1	0	0	0
HeartBeat	0	1	1	0	0
Majority Quórum	1	0	0	0	1
Generation Clock	0	1	1	0	0
High-Water Mark	0	1	0	0	0
Paxos	1	0	0	0	1
Replicated Log	1	1	0	0	0
Singular Update Queue	0	0	0	1	0
Request Waiting List	0	1	0	0	0
Idempotent Receiver	0	1	0	1	0
Follower Reads	1	1	0	0	0
Versioned Value	0	1	0	1	0
Version Vector	0	1	0	1	0
Fixed Partitions	1	0	0	0	0
Key-Range Partitions	1	0	0	0	0
Two-Phase Commit	1	1	0	0	0
Lamport Clock	0	1	0	1	0
Hybrid Clock	0	0	1	0	0
Clock-Bound Wait	0	0	1	0	0
Consistent Core	1	0	0	0	1
Lease	0	1	1	0	0
State Watch	0	0	0	1	0
Gossip Dissemination	0	1	0	0	1
Emergent Leader	1	1	1	0	0
Single-Socket Channel	0	1	0	0	0
Request Batch	0	1	0	0	0
Request Pipeline	0	1	1	0	0

Como fica a noção "*exactly-once semantics*"

- A rede não é perfeitamente confiável. Mensagens podem ser perdidas durante o transporte.
- Para lidar com a perda, o remetente reenvia a mensagem.
- Isso pode levar à entrega de mensagens duplicadas, pois o remetente não sabe se a mensagem original foi perdida ou apenas atrasada.

Como fica a noção "*exactly-once semantics*"



O que é Exactly-once Semantics?

- Garantia de que uma operação será processada exatamente uma vez, mesmo que a mensagem seja enviada várias vezes.
- Diferencia delivery (chegada da mensagem) de processing (processamento pelo software).
- Objetivo: garantir que a aplicação não processe mensagens duplicadas.

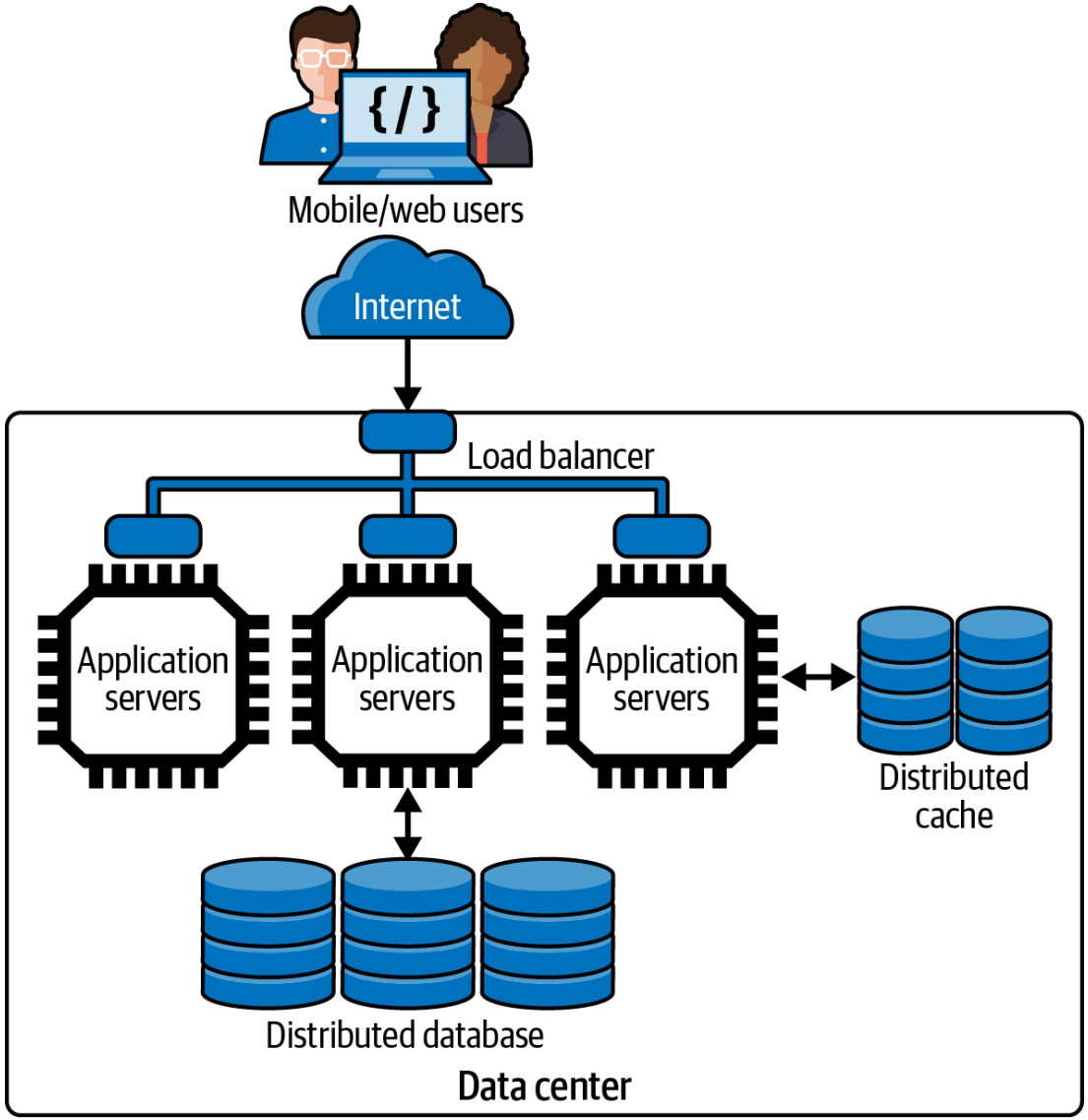
Como suportar Exactly-once Semantics?

- **Idempotent Operations:** Operações que podem ser aplicadas várias vezes **sem alterar o resultado** após a primeira aplicação.
- **De-duplication**: Cada mensagem recebe um identificador único. O destinatário armazena IDs já processados e ignora repetidos. Requer controle em ambos os lados (remetente e destinatário).

Como suportar Exactly-once Semantics?

- **Vários dos padrões de projetos ajudam a resolver esse problema:**
  - **Idempotência** (Idempotent Receiver, Versioned Value, Version Vector) garante que operações repetidas não mudem o resultado final.
  - **Controle de ordem e confirmação** (Replicated Log, High-Water Mark, Majority Quorum) garante que cada evento só seja aplicado quando seguro.
  - **Logs e commit atômico** (WAL, Segmented Log, 2PC) permitem recuperação confiável sem duplicar efeitos.

Replicação de Dados



Replicação de Dados

- Em um sistema distribuído, **dados precisam ser replicados** entre múltiplos nós para garantir **disponibilidade** e **tolerância a falhas**.
- O problema é que **as réplicas ficam em máquinas diferentes**, comunicando-se por redes não confiáveis e com latência variável.
- Isso leva a **inconsistência temporal**: uma réplica pode estar mais atualizada que outra.
- Esse cenário está no cerne do **problema de consistência** (CAP Theorem).
- **Exemplo:** Em um sistema de e-commerce, um item pode ser vendido simultaneamente para dois clientes diferentes se duas réplicas não tiverem sincronizado o estoque ainda.
- **Consistência vs Disponibilidade** (CAP Theorem).
- Sincronização eficiente das réplicas.
- Detecção e resolução de conflitos (eventual consistency, CRDTs).

Teorema CAP

- O **Teorema CAP** é um conceito fundamental em sistemas distribuídos que descreve as limitações inerentes ao design de sistemas distribuídos. CAP é um acrônimo para os três principais atributos que um sistema distribuído pode oferecer:
  - **Consistência (Consistency)**
  - **Disponibilidade (Availability)**
  - **Tolerância a Partições (Partition Tolerance)**

1. Consistência

- Definição: Em um sistema consistente, todas as leituras de um dado retornam o mesmo valor, garantindo que todos os nós tenham a mesma visão dos dados a qualquer momento. Se um dado é atualizado em um nó, todas as outras operações subsequentes devem ver essa atualização.
- Implicação: Consistência é sobre garantir que todos os nós do sistema tenham uma visão sincronizada e atualizada dos dados.

Modelos de Consistência

- **Consistência Forte (Strong Consistency)**
  - **Definição:** Em um sistema com consistência forte, uma operação de leitura sempre reflete a mais recente atualização confirmada em qualquer nó do sistema. Todos os nós têm uma visão consistente e atualizada dos dados a qualquer momento.
  - **Exemplo:** Sistemas de banco de dados relacionais tradicionais, como **Oracle** e **MySQL** (em modo de transação), garantem consistência forte.
  - **Vantagens:** Garante que todos os nós vejam os dados mais recentes.
  - **Desvantagens:** Pode reduzir a disponibilidade e a performance, especialmente em sistemas distribuídos e durante partições de rede.

Modelos de Consistência

- **Consistência Causal (Causal Consistency)**
  - **Definição:** Em um sistema com consistência causal, operações que estão causalmente relacionadas (ou seja, uma operação deve ocorrer após a outra) são vistas na mesma ordem por todos os nós. Entretanto, a ordem de operações não causalmente relacionadas pode variar entre os nós.
  - **Exemplo:** Sistemas de mensagens e algumas implementações de redes sociais utilizam consistência causal.
  - **Vantagens:** Garante que as relações de causa e efeito sejam preservadas.
  - **Desvantagens:** Menos rigorosa do que a consistência forte, mas mais complexa de implementar do que a consistência eventual.

Modelos de Consistência

- **Consistência Eventual (Eventual Consistency)**
  - **Definição:** Em um sistema com consistência eventual, as atualizações são eventualmente propagadas a todos os nós, e, após um certo tempo, todos os nós convergem para o mesmo estado. Enquanto isso, leituras podem retornar valores desatualizados.
  - **Exemplo: Amazon DynamoDB**, **Apache Cassandra** e **Riak** utilizam consistência eventual.
  - **Vantagens:** Oferece alta disponibilidade e tolerância a partições, com boa performance.
  - **Desvantagens:** Não garante que todas as leituras reflitam as atualizações mais recentes imediatamente.

2. Disponibilidade

- Definição: Um sistema disponível garante que cada solicitação recebida seja respondida, seja com sucesso ou com uma falha. Mesmo se alguns nós estiverem indisponíveis, o sistema deve continuar a funcionar e fornecer respostas aos clientes.
- Implicação: Disponibilidade é sobre garantir que o sistema esteja operacional e respondendo a solicitações, mesmo em caso de falhas.

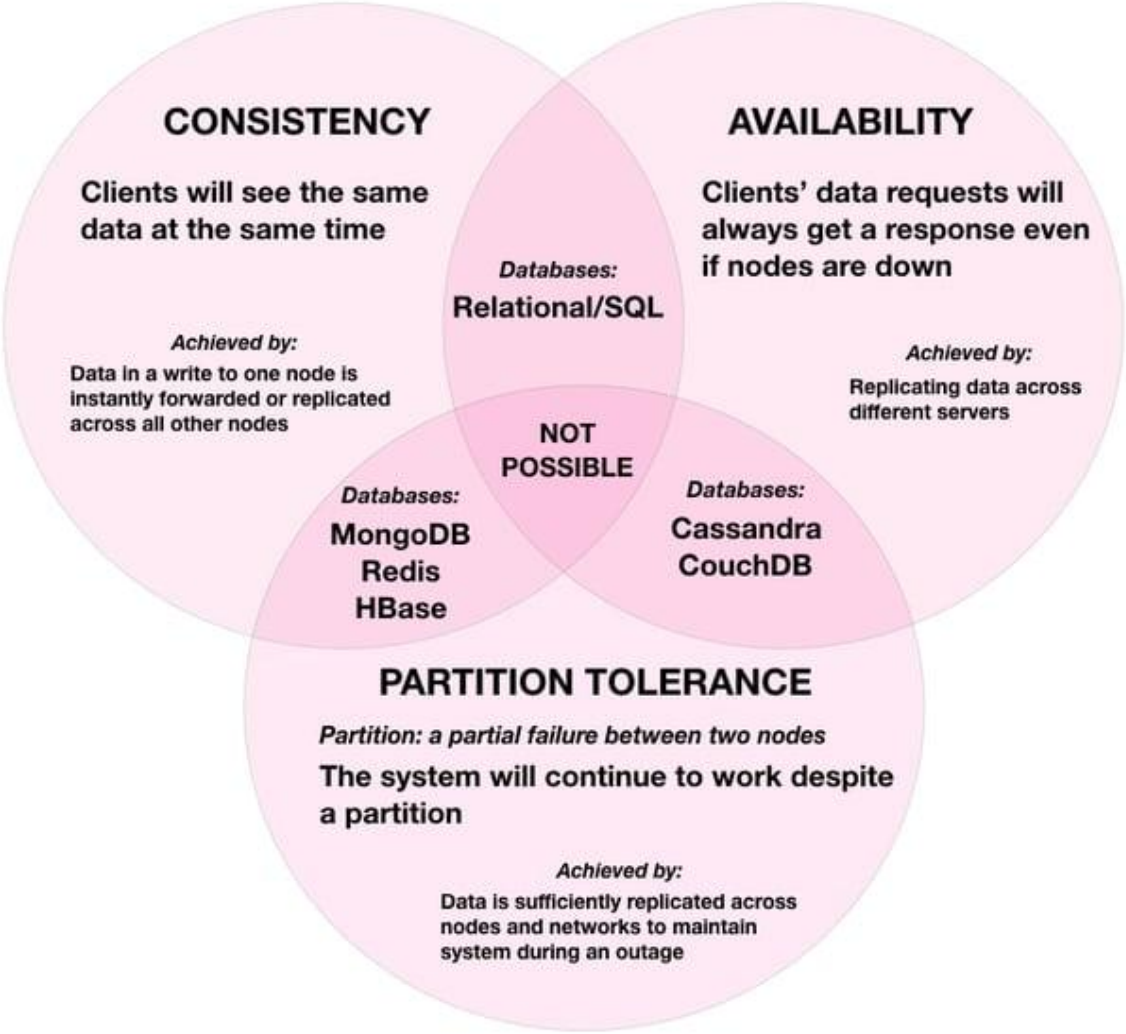
3.Tolerância a Partições

- Definição: Tolerância a partições significa que o sistema deve continuar a operar corretamente mesmo se houver uma falha de comunicação que divide a rede em partes desconectadas. O sistema deve ser capaz de lidar com a perda de comunicação entre alguns de seus nós e ainda fornecer serviços.
- Implicação: Tolerância a partições é sobre a capacidade do sistema de continuar funcionando apesar de falhas de rede que causam a separação de seus componentes.

Teorema CAP

- O Teorema CAP afirma que **é impossível** para um sistema distribuído garantir simultaneamente os três atributos. Em outras palavras, um sistema distribuído pode, no máximo, satisfazer dois desses três atributos em qualquer momento, mas não todos os três simultaneamente.

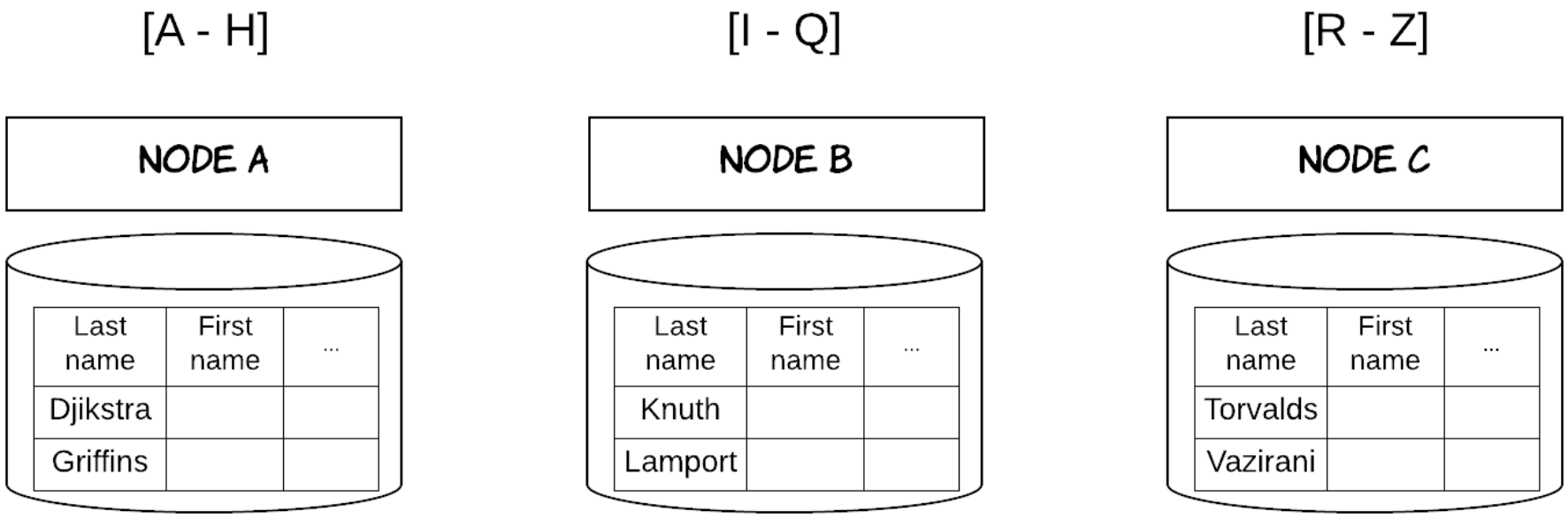
Teorema CAP


CONSISTENCY
Clients will see the same data at the same time
Achieved by:
Data in a write to one node is instantly forwarded or replicated across all other nodes
AVAILABILITY
Clients' data requests will always get a response even if nodes are down
Achieved by:
Replicating data across different servers
Databases:
Relational/SQL
NOT POSSIBLE
Databases:
MongoDB
Redis
HBase
Databases:
Cassandra
CouchDB
PARTITION TOLERANCE
Partition: a partial failure between two nodes
The system will continue to work despite a partition
Achieved by:
Data is sufficiently replicated across nodes and networks to maintain system during an outage

Teorema CAP

- **Consistência e Tolerância a Partições (CP):** Sistemas que priorizam consistência e tolerância a partições podem não estar sempre disponíveis. Por exemplo, MongoDB em um cluster com replica sets, é necessário um quórum para eleger um nó primário. Se a partição impede que um nó atinja o quórum, ele não aceita escritas. Assim, evita inconsistências, mas sacrifica disponibilidade temporariamente
- **Disponibilidade e Tolerância a Partições (AP):** Sistemas que priorizam disponibilidade e tolerância a partições podem sacrificar a consistência. Por exemplo, sistemas de **eventual consistency** como **Cassandra** garantem que todos os dados eventualmente se tornem consistentes, mas podem responder a solicitações mesmo quando a rede está particionada, podendo retornar dados desatualizados temporariamente. **CouchDB** mantém cópias independentes e resolve conflitos usando *version vectors*.

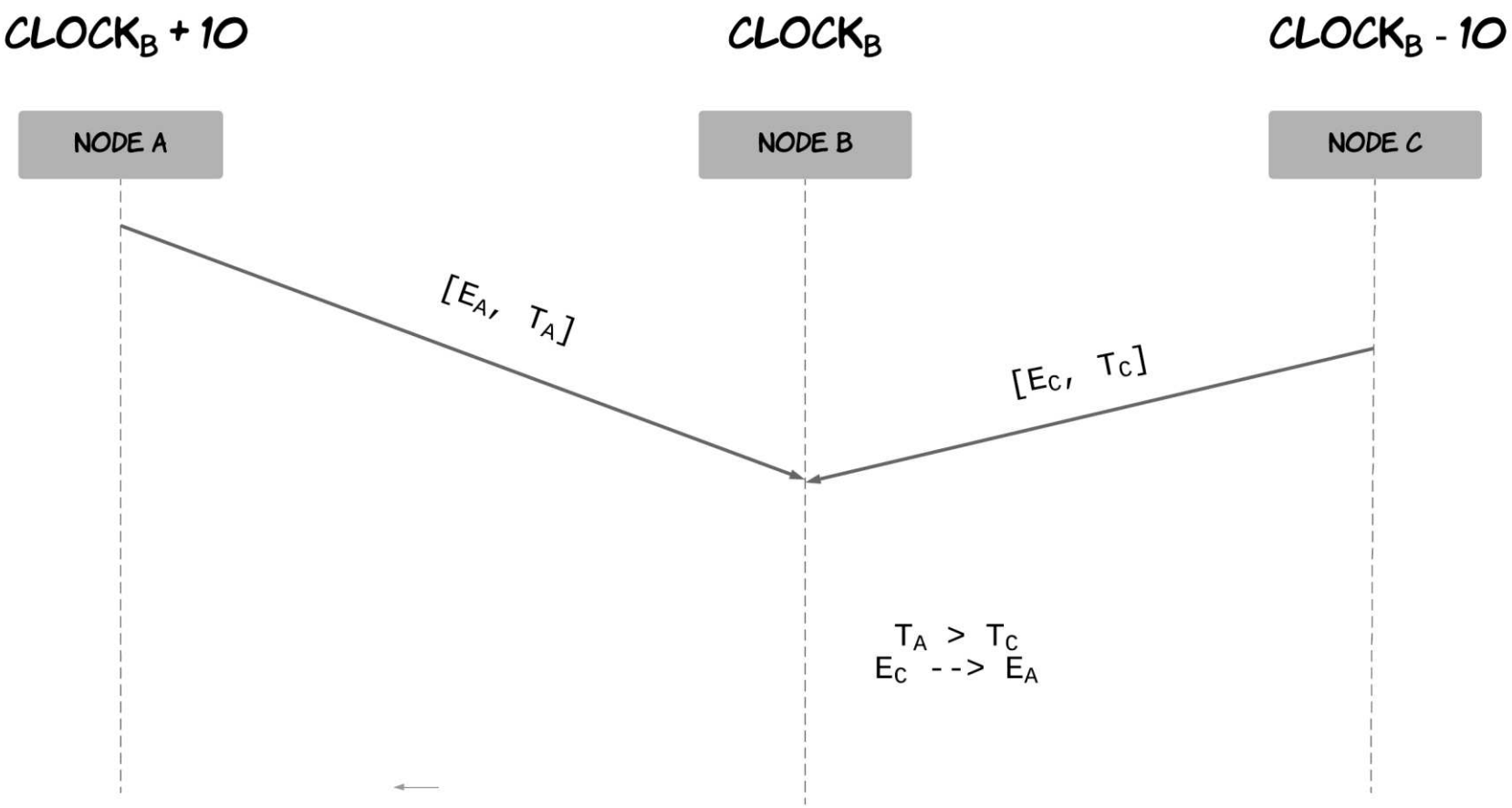
Partição de Dados



Partição de Dados

- Grandes volumes de dados não cabem em um único nó (limites de **armazenamento**, **CPU** e **rede**).
- A solução é **particionar (shard) os dados** entre nós — cada partição guarda uma parte do todo.
- Isso cria o problema de **coordenação entre partições**: operações que afetam dados em múltiplas partições exigem protocolos complexos (como **Two-Phase Commit**) e sofrem com falhas parciais.
- **Exemplo:** Um banco de dados distribuído com *Fixed Partition* ou *Key-Range Partition* precisa lidar com o que acontece se uma partição falha no meio de uma transação.
- **Principais desafios:**
  - Localizar rapidamente qual partição contém o dado.
  - Rebalancear partições quando nós entram/saem.
  - Garantir atomicidade em transações multi-partição.

Tempo em Sistemas Distribuídos



Tempo em Sistemas Distribuídos

- Em um sistema distribuído **não há relógio global perfeito**.
- Cada nó tem seu próprio relógio, que pode estar **desincronizado** devido a:
  - Drift de hardware.
  - Latência de rede ao tentar sincronizar.
- Como consequência, **a ordem de eventos pode ser ambígua**.
- Isso é crítico para garantir **ordenação de operações**, **detecção de conflitos** e **causalidade**.
- **Exemplo:** Dois usuários editam o mesmo documento quase simultaneamente em servidores diferentes. Quem editou "primeiro" depende do ponto de vista do relógio.
- **Principais desafios:**
  - Definir **ordem lógica** (Lamport Clock, Vector Clock).
  - Usar relógios híbridos (Hybrid Logical Clock) para equilibrar tempo físico e lógico.
  - Evitar assumir que timestamps físicos são sempre corretos.

Por isso que Sistemas Distribuídos são Complexos

- **Replicação** precisa de **Time** para decidir qual versão é a mais nova.
- **Partição** agrava problemas de **Replicação** porque as réplicas ficam espalhadas.
- **Time** influencia protocolos de consenso e commit usados para manter consistência entre **partições** e **replicas**.

Atividade

- Estudar o padrão de projeto sorteado