



Bacharelados em  
Computação

**UniEVANGÉLICA**  
UNIVERSIDADE EVANGÉLICA DE GOIÁS

# ENGENHARIA DE SOFTWARE

## SISTEMAS DISTRIBUÍDOS


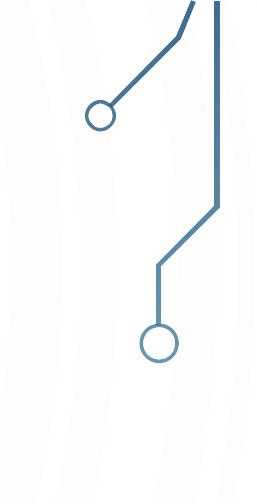
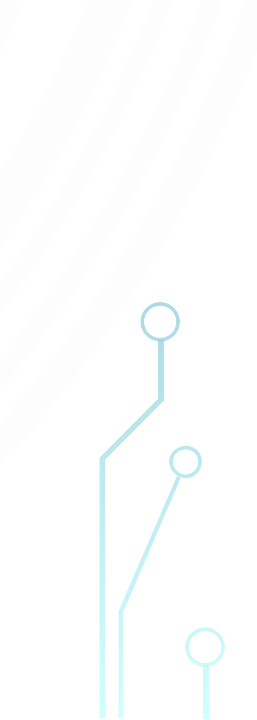
### AULA 10 – CONSISTÊNCIA, REPLICAÇÃO E SISTEMAS DE ARQUIVOS DISTRIBUÍDOS

PROF. ÁLVARO LOPES BASTOS

ANÁPOLIS – 2025.1

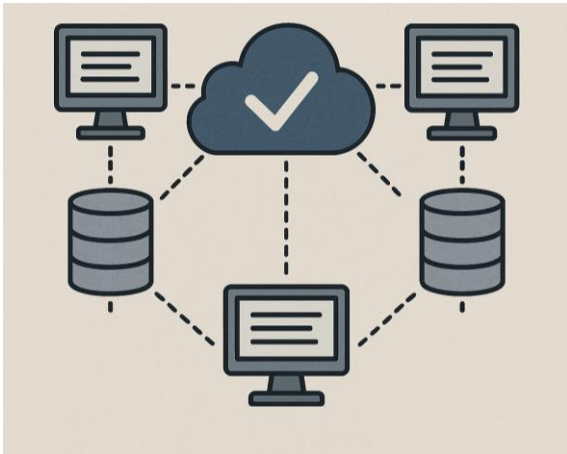


# OBJETIVOS DA AULA

- Compreender os desafios da sincronização temporal em sistemas distribuídos
  - Estudar algoritmos clássicos: Cristian, Berkeley, NTP
  - Entender o problema da exclusão mútua distribuída
- 
- 
- 

# CONSISTÊNCIA

- **Consistência** refere-se à visibilidade uniforme dos dados.
- Em um sistema com múltiplos nós, é necessário garantir que todos vejam a mesma informação, especialmente após atualizações.
- Se um cliente realiza uma alteração em um dado, essa mudança deve ser refletida corretamente em todas as réplicas.

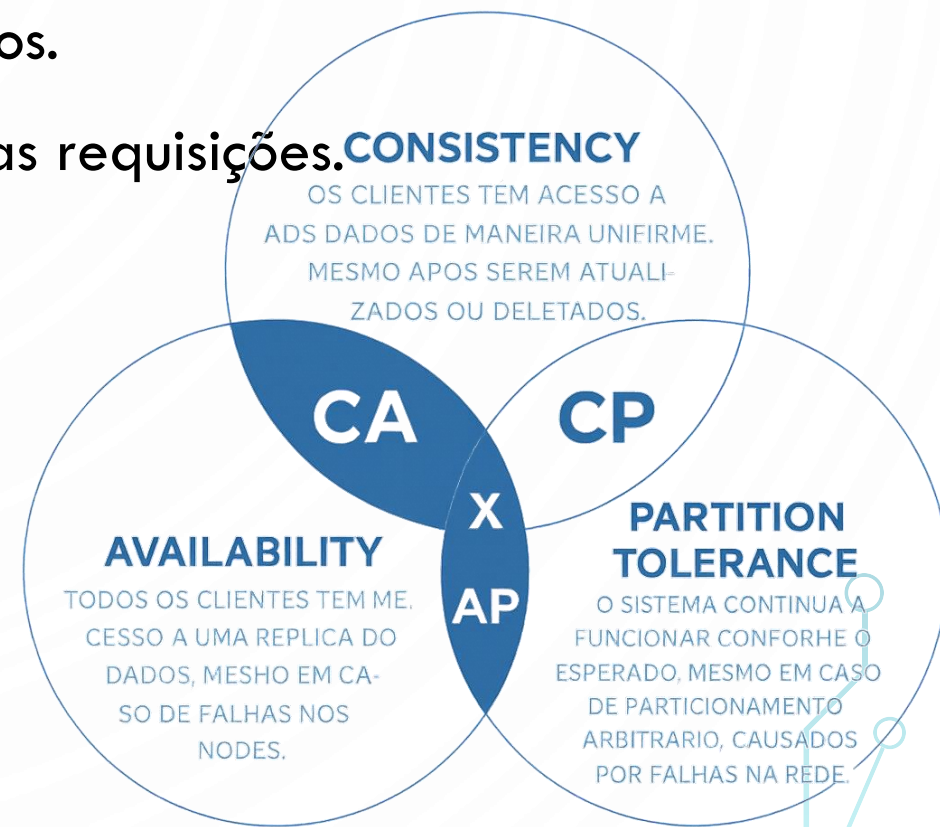


# TEOREMA CAP

O teorema CAP afirma que, em um sistema distribuído, só é possível garantir **duas de três propriedades**:

- **Consistência (C)**: todos os nós têm os mesmos dados.
- **Disponibilidade (A)**: o sistema responde a todas as requisições.
- **Tolerância à partição (P)**: o sistema continua funcionando mesmo com falhas na comunicação entre os nós.

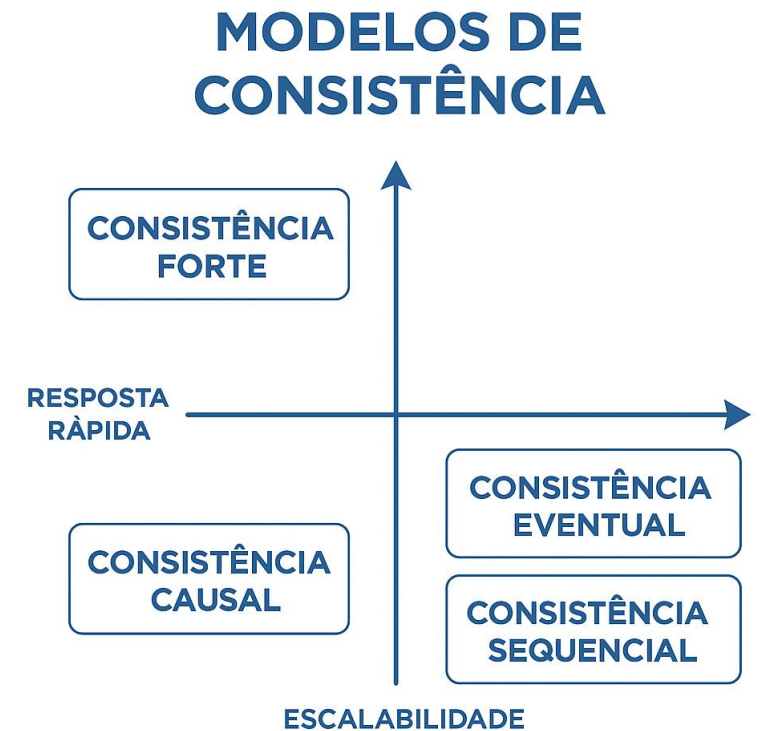
Isso significa que todo sistema distribuído precisa fazer escolhas, priorizando algumas propriedades em detrimento de outras.



# MODELOS DE CONSISTÊNCIA

Os sistemas distribuídos podem seguir diferentes modelos de consistência:

- **Forte (ou Linearizável):** as leituras sempre retornam o valor mais recente.
- **Sequencial:** as operações são vistas na mesma ordem por todos, mas não precisam refletir o tempo real.
- **Causal:** garante a ordem entre operações relacionadas causalmente.
- **Eventual:** as réplicas convergem com o tempo, mas podem divergir temporariamente.



# CONSISTÊNCIA FORTE

- Todas as leituras após uma escrita refletem o valor atualizado.
- Modelo mais rigoroso e o mais custoso em termos de desempenho, pois exige sincronização constante.
- Utilizado em sistemas onde integridade dos dados é prioritária.
- **Utilização:**
  - Sistemas bancários
  - E-commerces

# CONSISTÊNCIA FORTE

TODOS OS NODES VÊEM  
OS DADOS ATUALIZADOS  
AO MESMO TEMPO

# CONSISTÊNCIA CAUSAL

- Preserva a ordem entre eventos que estão logicamente relacionados.
- É um meio-termo entre forte e eventual
- Oferece mais flexibilidade com alguma garantia de ordem.

## CONSISTÊNCIA CASUAL

OS NODES VÊEM OS  
DADOS NA MESMA  
ORDEM EM QUEM  
ATUALIZADOS



# CONSISTÊNCIA EVENTUAL

- O sistema permite leituras inconsistentes temporárias
- Garante que, eventualmente, todos os nós terão os mesmos dados.
- Reduz a latência e aumenta a escalabilidade.
- **Utilização:**
  - Redes Sociais
  - Caches DNS
  - Sistemas de mensageria

## CONSISTÊNCIA EVENTUAL

SE NENHUMA NOVA  
ATUALIZAÇÃO FOR  
REALIZADA, EVENTUALMENTE  
TODOS OS NODES  
CONVERGEM



# COMPARATIVO ENTRE MODELOS

Modelo	Latência	Escalabilidade	Ordem das operações
Forte	Alta	Baixa	Total
Sequencial	Média	Média	Preservada
Causal	Média	Alta	Parcial
Eventual	Baixa	Muito Alta	Nenhuma

# O QUE É REPLICAÇÃO?

A replicação consiste em manter **cópias dos mesmos dados** em vários computadores para:

- **Melhorar o desempenho**
- **Aumentar a disponibilidade**
- **Garantir tolerância a falhas**

Ela é amplamente usada, por exemplo:

- Em caches de navegadores e proxies web.
- No DNS, que mantém mapeamentos replicados de nomes de domínio.

# VANTAGENS DA REPLICAÇÃO

- Alta disponibilidade: mesmo se um servidor falhar, outros têm a informação.
- Melhor desempenho: acesso ao dado pode ser mais rápido.
- Tolerância a partições: continuidade mesmo com falhas de rede.

- **Desafios:**

- **Transparência**


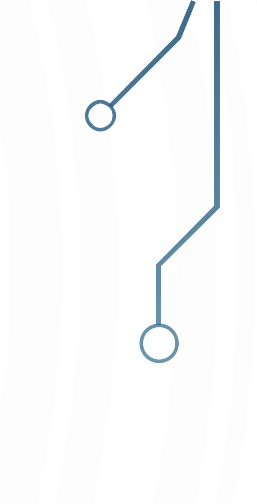
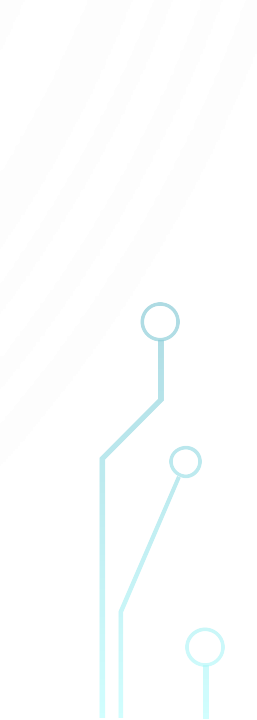
- O local de replicação deve ser transparente para o cliente

- **Consistência**

- Manter os dados replicados em cada nó atualizados.

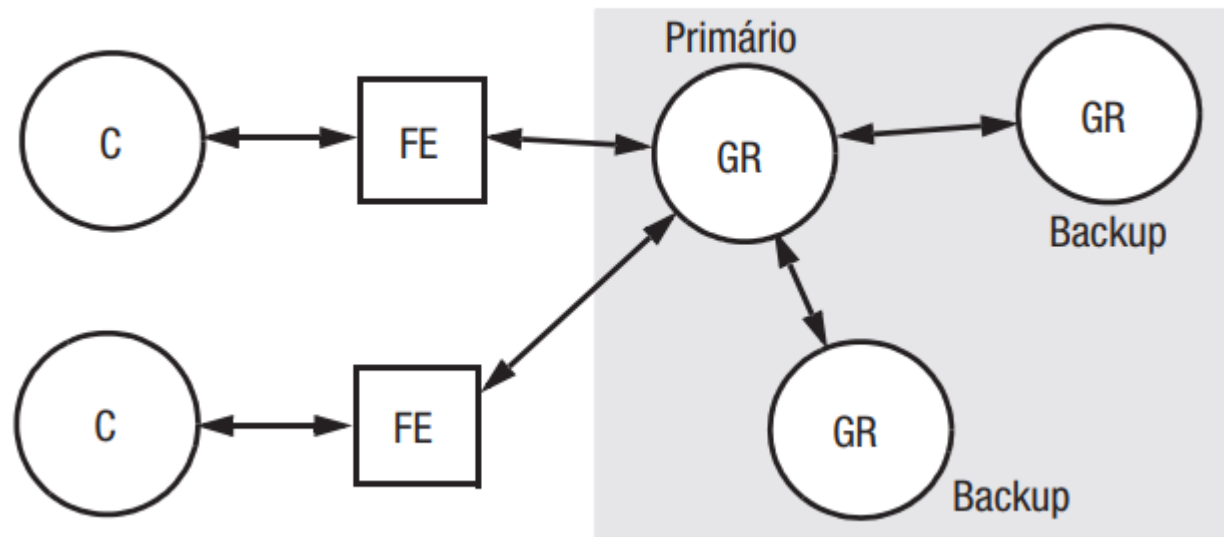


# MODELOS DE REPLICAÇÃO

- **Replicação Passiva:** Um nó primário é responsável por processar todas as operações de escrita.
  - **Replicação Ativa:** Todos os nós executam as operações simultaneamente.
- 
- 
- 

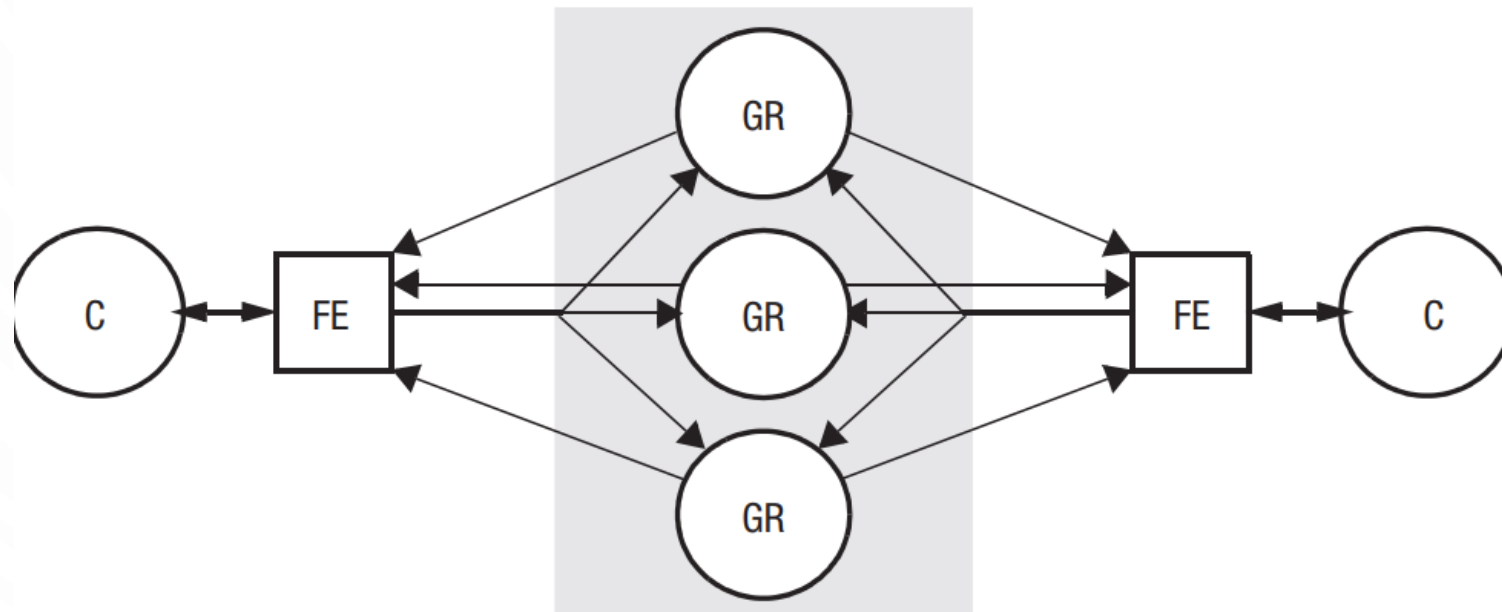
# REPLICAÇÃO PASSIVA

- **Um nó primário é responsável por processar todas as operações de escrita.**
- As atualizações são propagadas para réplicas secundárias.
- É mais simples, mas se o nó primário falhar, o sistema pode se tornar indisponível até que um novo líder seja escolhido.



# REPLICAÇÃO ATIVA

- **Todos os nós executam as operações simultaneamente.**
- Isso exige protocolos de ordenação e determinismo.
- É mais robusta a falhas, porém complexa de manter e sincronizar, especialmente sob alta carga.



# SISTEMAS DE ARQUIVOS DISTRIBUÍDOS (SAD)

- Permite acesso a arquivos de diferentes nós de forma transparente.
- Como se o arquivo estivesse local.

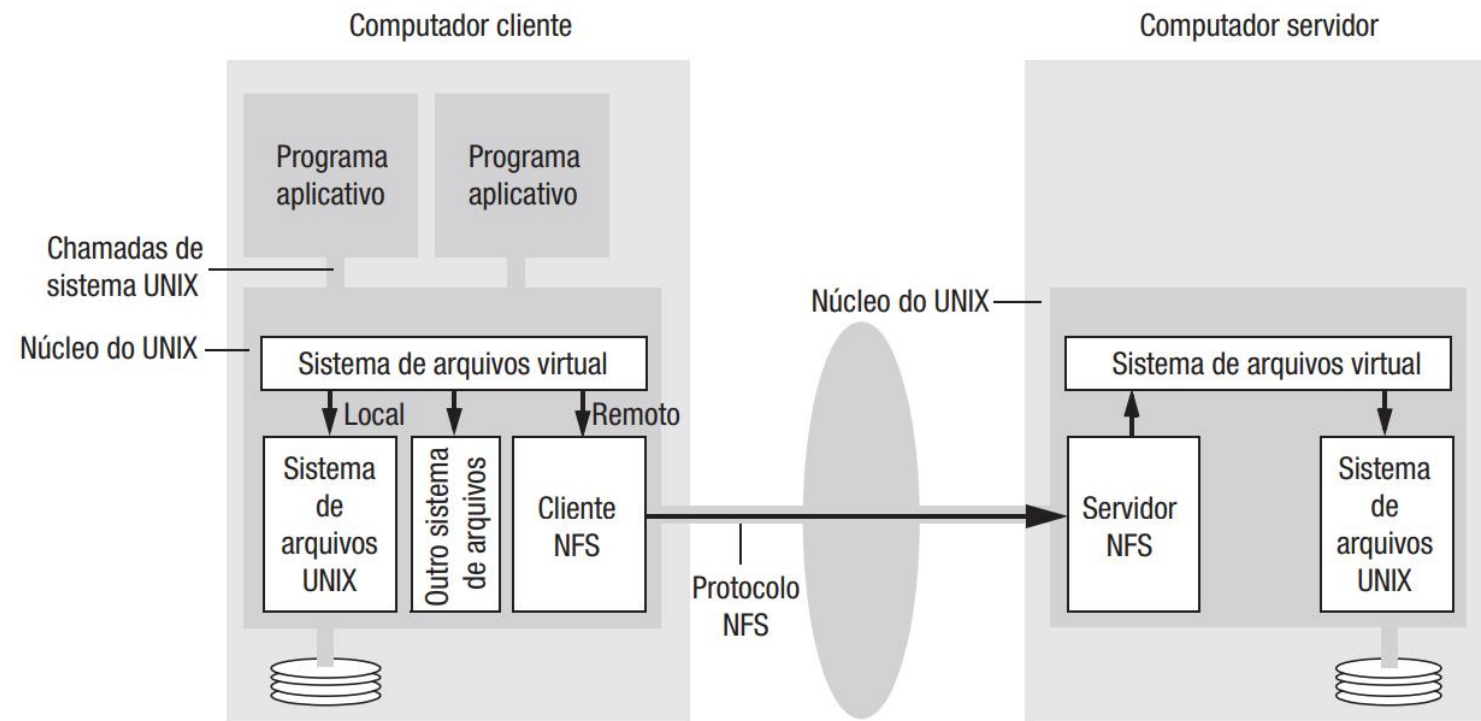
## **Funcionalidades Esperadas:**

- Transparência de localização
- Replicação
- Gerenciamento de cache
- Tolerância a falhas



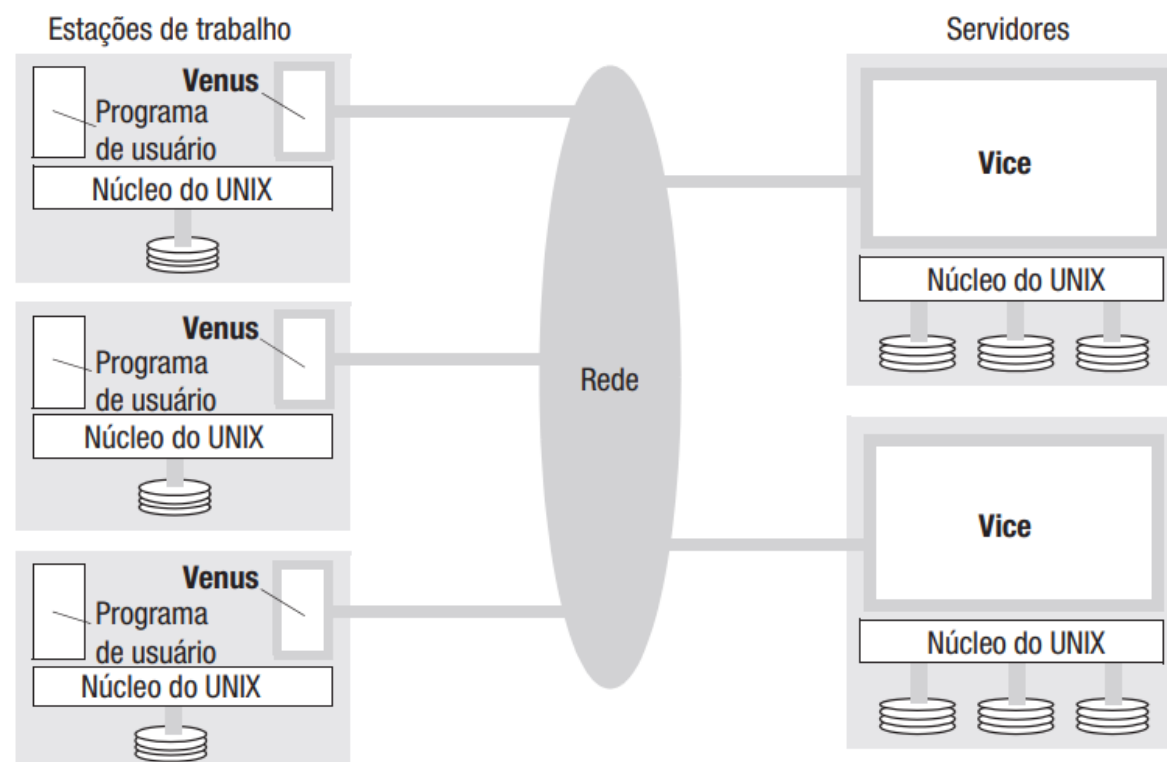
# NFS (NETWORK FILE SYSTEM)

- Arquitetura cliente-servidor.
- Baixa escalabilidade.
- Usado em redes locais.



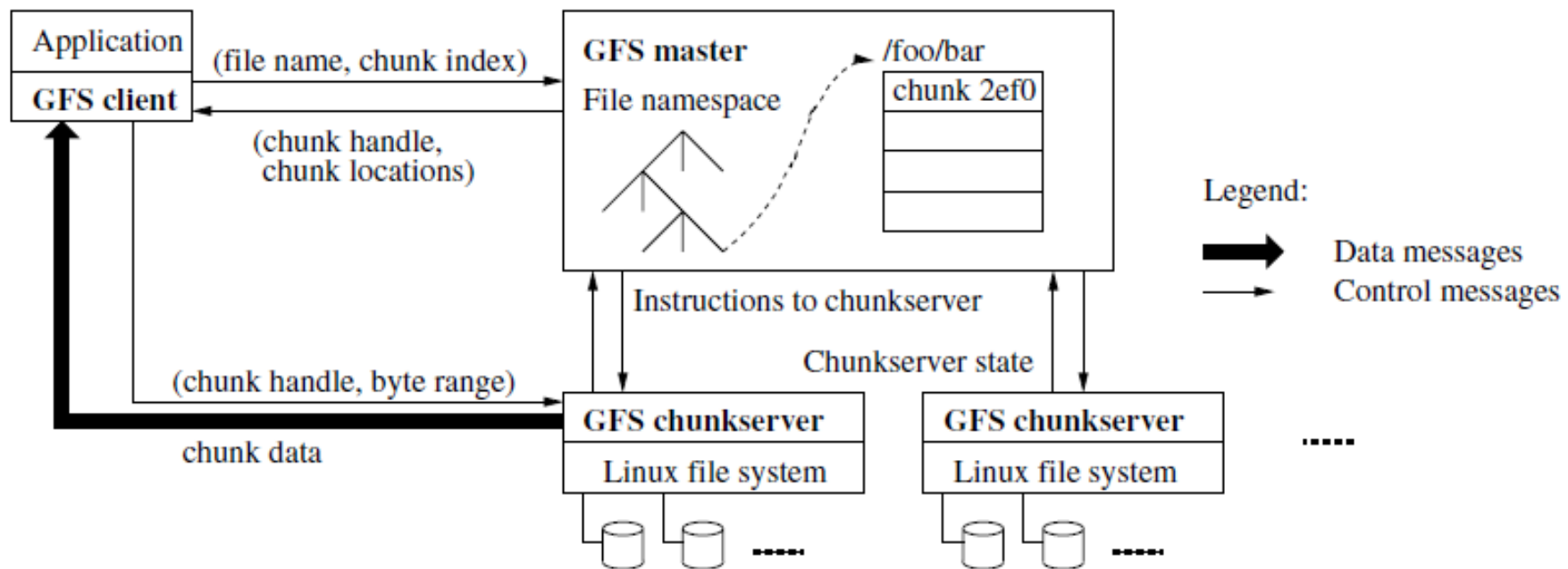
# AFS (ANDREW FILE SYSTEM)

- Usa cache local extensivo.
- Validação de dados por tempo ou eventos.
- Boa escalabilidade.



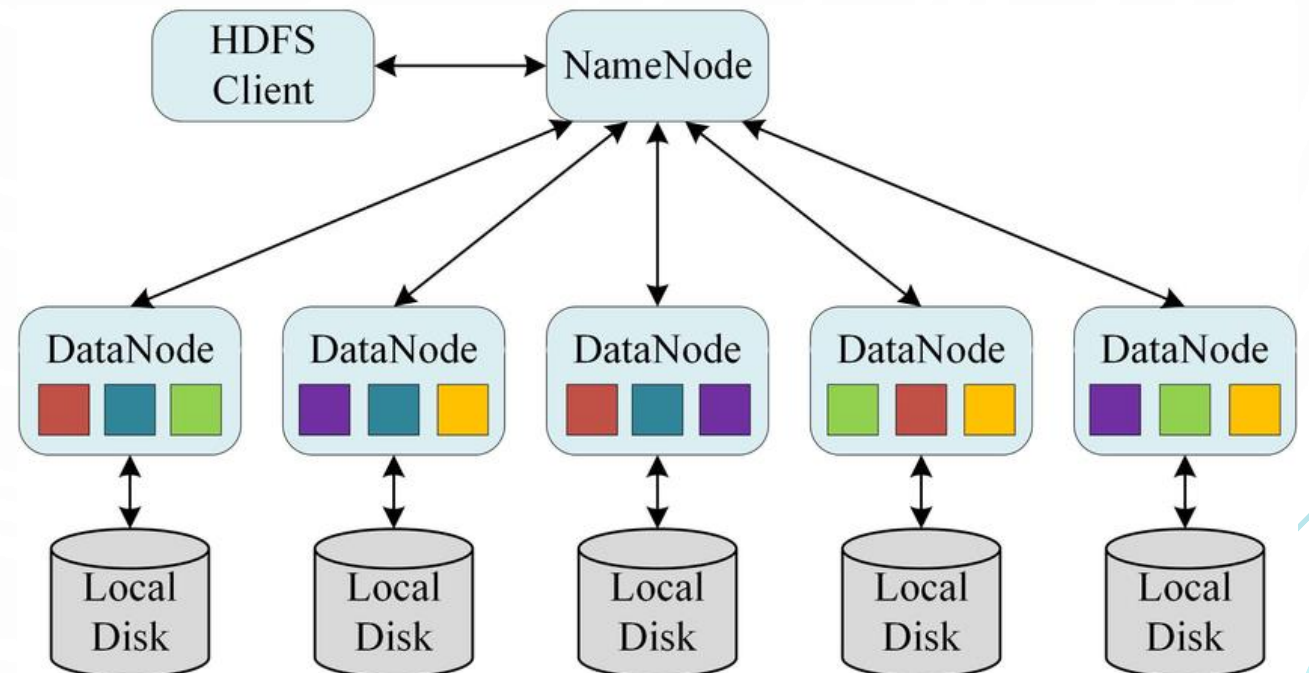
# GFS (GOOGLE FILE SYSTEM)

- Criado para armazenar grandes arquivos.
- Arquitetura: 1 master + vários chunkservers.
- Replicação automática (3 cópias padrão).



# HDFS (HADOOP DISTRIBUTED FILE SYSTEM)

- Inspirado no GFS.
- NameNode (metadados) + DataNodes (dados).
- Suporta petabytes de dados.
- Projetado para MapReduce.



# COMPARATIVO ENTRE SAD'S

Sistema	Arquitetura	Escalabilidade	Tolerância a falhas
NFS	Centralizado	Baixa	Baixa
AFS	Cache local	Média	Média
GFS	Master + Chunks	Alta	Alta
HDFS	NameNode/DataNode	Muito Alta	Alta

# SADS E CONSISTÊNCIA

- **Consistência:**

- GFS/HDFS: replicação automática com consistência eventual.
- AFS: forte para escrita, eventual para leitura.
- NFS: depende da configuração, geralmente fraco.

- **Aplicações práticas:**

- NFS: empresas pequenas, redes locais.
- AFS: ambientes acadêmicos.
- GFS: serviços Google.
- HDFS: processamento de Big Data (ex: Amazon EMR, Apache Spark).

# SISTEMAS DE NOMEAÇÃO

- Mapeia nomes simbólicos para identificadores de recursos.
- Permite localizar arquivos, serviços e dispositivos.

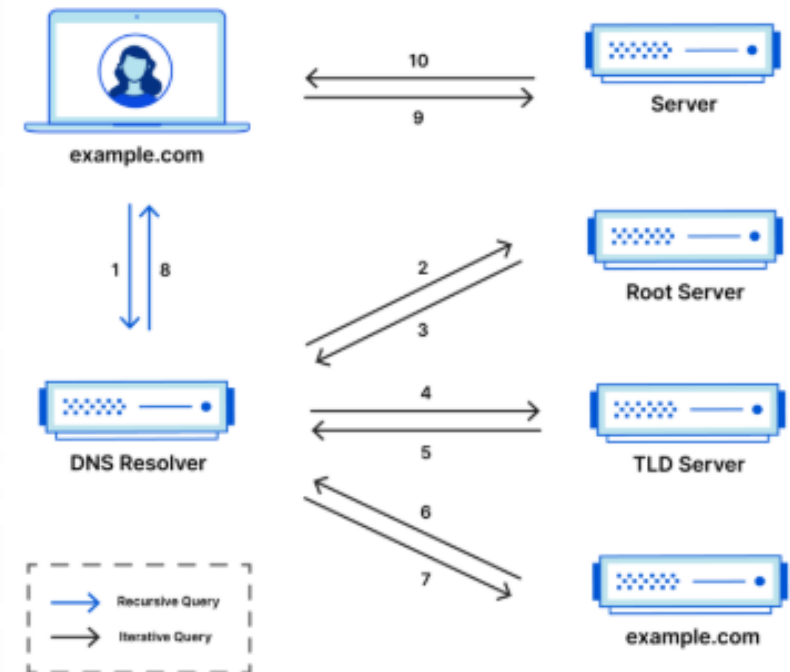
## Tipos de Nomeação:

- **Hierárquico:** como no DNS.
- **Plana:** sem estrutura, baseado em IDs únicos.
- **Descritiva:** baseada em atributos (ex: "impressora cor 3º andar").



# DNS – DOMAIN NAME SYSTEM

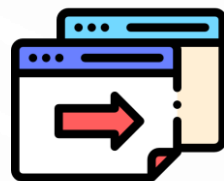
- Resolve nomes como `www.exemplo.com` para IPs.
- Distribuído, replicado e altamente disponível.
- Utiliza cache para reduzir tráfego.



# NOMEAÇÃO EM SADS

- SADs também usam nomeação: Ex: HDFS permite acessar `/user/alunos/arq.txt`.
- O NameNode resolve esse nome para localizar os blocos reais em DataNodes.

# ENCERRAMENTO



## Próxima Aula

Sistemas Distribuídos para Internet



**`alvaro.bastos@docente.unievangelica.edu.br`**



Bacharelados em  
Computação

**UniEVANGÉLICA**  
UNIVERSIDADE EVANGÉLICA DE GOIÁS

# ENGENHARIA DE SOFTWARE

## SISTEMAS DISTRIBUÍDOS

### AULA 11 – SISTEMAS DISTRIBUÍDOS PARA WEB

PROF. ÁLVARO LOPES BASTOS

ANÁPOLIS – 2025.1

# A WEB COMO SISTEMA DISTRIBUÍDO

- Utiliza arquitetura cliente-servidor.,
- Baseada em protocolos padronizados, como HTTP.
- Permite a comunicação entre navegadores (clientes) e servidores remotos.



# ARQUITETURA DE 3 CAMADAS NA WEB


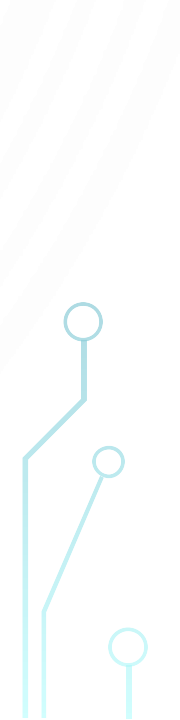
**Camada de Apresentação:** interface com o usuário (HTML, CSS, JS).

**Camada de Negócios:** lógica da aplicação (backend).

**Camada de Dados:** persistência (banco de dados).



# FUNCIONAMENTO DO NAVEGADOR

- Cliente Web que solicita recursos ao servidor.
  - Recebe arquivos (HTML, CSS, JS) e os interpreta.
  - Interage com APIs REST e serviços remotos via HTTP
- 
- 



# COMUNICAÇÃO HTTP

- Protocolo stateless (sem memória entre requisições).
- Métodos principais:
  - GET: busca dados
  - POST: envia dados
  - PUT: atualiza
  - DELETE: remove
- Utiliza URLs para identificar recursos.

# SOAP (SIMPLE OBJECT ACCESS PROTOCOL)

## O que é?

SOAP é um **protocolo de comunicação** que permite a troca de mensagens estruturadas entre aplicações através da **Web**. É amplamente usado em **Web Services**, especialmente em ambientes corporativos.

## Principais características:

- Baseado em **XML** para formatação das mensagens.
- Utiliza protocolos como **HTTP** ou **SMTP** para transporte.
- **Extensível** e padronizado pelo W3C.
- Oferece suporte robusto a **segurança, transações e controle de mensagens**.

# XML (EXTENSIBLE MARKUP LANGUAGE)

## O que é?

XML é uma **linguagem de marcação** usada para armazenar, transportar e descrever dados. Foi projetada para ser **autoexplicativa** e **legível por humanos e máquinas**.

## Principais características:

- Possui **estrutura hierárquica**, com **tags** que descrevem o conteúdo.
- Independente de plataforma e linguagem de programação.
- Utilizado como **formato padrão** de mensagens em diversos protocolos, como o SOAP.

```
1  <cliente>
2    <nome>Joao da Silva</nome>
3    <idade>35</idade>
4    <email>joa[email].com</email>
5  </cliente>
```

# APIS RESTFUL

- **O que é?**

Um estilo arquitetural para construção de APIs web baseadas no protocolo HTTP.

- **Características:**

- Usa métodos HTTP: **GET, POST, PUT, DELETE**
- Requisições baseadas em URLs que representam recursos
- Stateless (não mantém estado entre requisições)
- Fácil de integrar com frontends e sistemas distribuídos

# JSON (JAVASCRIPT OBJECT NOTATION)

- **O que é?**

- Um formato leve de troca de dados, baseado em texto e legível por humanos.

- **Características:**

- Estrutura em pares chave-valor
- Altamente usado em APIs RESTful
- Alternativa moderna ao XML
- Suportado por quase todas as linguagens de programação

```
1  {  
2    "nome": "Ana",  
3    "idade": 28,  
4    "email": "ana@email.com"  
5  }
```

# ESCALABILIDADE

- É a **capacidade de um sistema crescer e manter seu desempenho** à medida que a demanda aumenta. Em sistemas distribuídos para Web, é essencial garantir que o sistema continue rápido, estável e disponível mesmo com:
- Mais usuários simultâneos
- Maior volume de dados
- Mais requisições por segundo

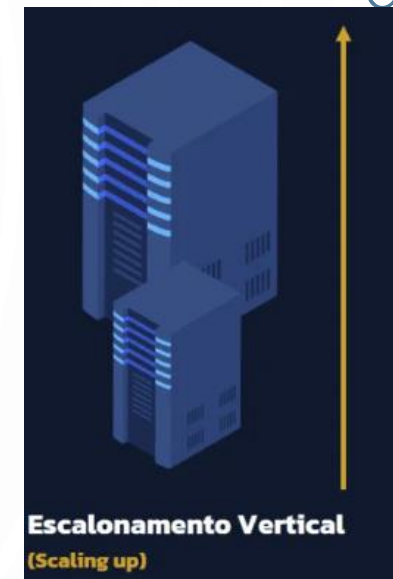
# TIPOS DE ESCALABILIDADE

## 📌 Escalabilidade Vertical (Scale Up)

- Aumenta o **poder de uma máquina** existente (CPU, RAM, armazenamento).
- Exemplo: mudar de uma VM t3.micro para uma t3.large na AWS.
- Mais simples, mas tem limite físico e custo crescente.

## 📌 Escalabilidade Horizontal (Scale Out)

- Adiciona **mais servidores/máquinas** ao sistema.
- Exemplo: adicionar novos nós à aplicação em um cluster Kubernetes.
- Mais flexível e eficiente em sistemas distribuídos e microsserviços.





# ESCALABILIDADE NA WEB

- **Load Balancer (Balanceador de carga)**

Distribui automaticamente as requisições entre múltiplas instâncias.

Exemplos: Nginx, HAProxy, AWS Elastic Load Balancer.

- **Replicação de Dados**

Duplica dados em múltiplos servidores para melhorar leitura e resiliência.

Exemplos: MongoDB Replica Set, PostgreSQL Streaming Replication.

- **Cache Distribuído**

Armazena em memória dados acessados com frequência para acelerar a resposta.

Exemplos: Redis, Memcached.

- **Auto Scaling**

Ajusta automaticamente a quantidade de instâncias com base no uso.

Exemplos: AWS Auto Scaling, Google Cloud Instance Groups.

# MICROSSERVIÇOS

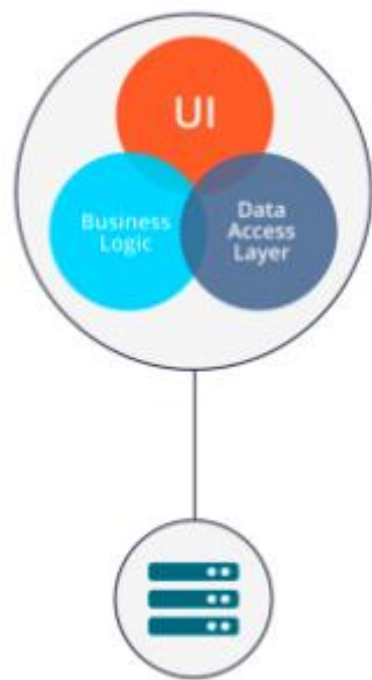
- **O que é?**

São uma abordagem arquitetural onde a aplicação é dividida em **vários serviços pequenos, independentes e especializados**, que se comunicam entre si por meio de interfaces bem definidas.

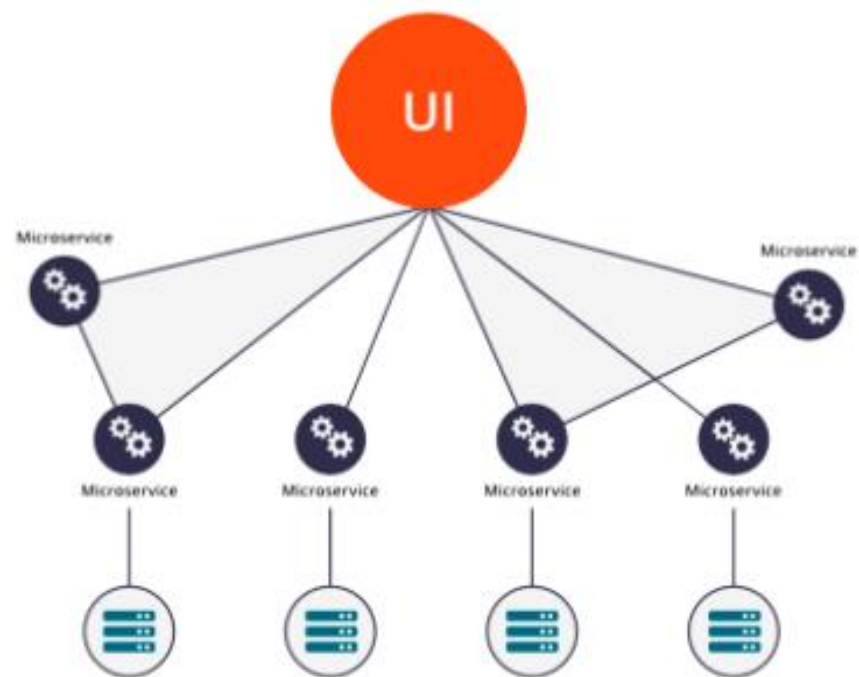
- **Características:**

- Responsabilidade única (single responsibility)
- Sua própria lógica de negócio
- Seu banco de dados independente
- Desenvolvidos, testados e implantados de forma autônoma
- Facilitam o deploy contínuo, escalabilidade seletiva e manutenção modular

# MICROSSERVIÇOS



Arquitetura Monolítica



Arquitetura de Microserviços

# DOCKER EM SISTEMAS DISTRIBUÍDOS

- Docker é uma **plataforma de containerização** que permite empacotar aplicações com todas as suas dependências em **containers portáteis e isolados**.

## O que são Containers?

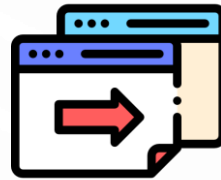
- Unidades leves e portáteis de software.
- Executam **isoladamente** do sistema operacional anfitrião.
- Incluem: aplicação, bibliotecas, configurações e binários necessários.



## Principais Vantagens do Docker

- **Portabilidade:** roda o mesmo container em qualquer ambiente (dev, teste, produção)
- **Velocidade:** mais leve e rápido que máquinas virtuais.
- **Isolamento:** múltiplos containers no mesmo host sem conflito.
- **Escalabilidade:** ideal para microsserviços e ambientes distribuídos.

# ENCERRAMENTO



## Próxima Aula

Tolerância a Falhas: Modelos de Falhas e Recuperação, Algoritmos de Detecção de Falhas e Checkpointing



**`alvaro.bastos@docente.unievangelica.edu.br`**



Bacharelados em  
Computação

**UniEVANGÉLICA**  
UNIVERSIDADE EVANGÉLICA DE GOIÁS

# ENGENHARIA DE SOFTWARE

## SISTEMAS DISTRIBUÍDOS


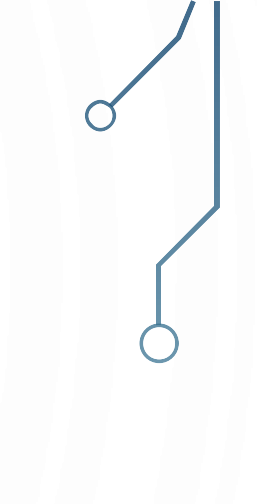
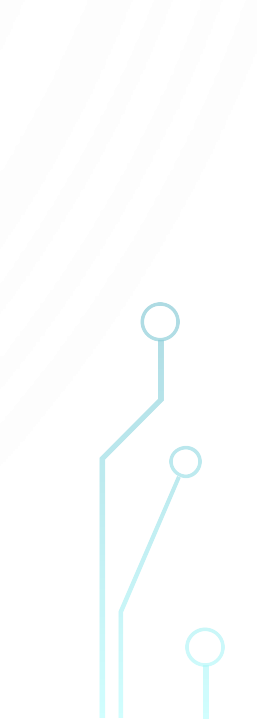
### AULA 12 – TOLERÂNCIA A FALHAS

PROF. ÁLVARO LOPES BASTOS

ANÁPOLIS – 2025.1



# OBJETIVOS DA AULA

- Compreender os modelos de falhas
  - Analisar mecanismos de detecção e recuperação de falhas
  - Estudar algoritmos de checkpointing
  - Entender a importância da tolerância a falhas
- 
- 
- 

# TOLERÂNCIA A FALHAS

- Capacidade de continuar operando mesmo diante de falhas
- Fundamental para garantir disponibilidade, confiabilidade e segurança

- **Tipos de Falhas:**

- Falha por Omissão: perda de mensagens
- Falha de Envio/Recepção
- Falha de Canal: perda no meio da comunicação
- Falhas de Colapso (Crash)
- Falhas Bizantinas (Arbitrárias)

## TOLERÂNCIA A FALHAS EM SISTEMAS DISTRIBUÍDOS

### MODELOS DE FALHAS



- Omissão
- Colapso
- Bizantina

### ALGORITMOS DE DETECÇÃO



- Timeouts
- Heartbeats
- Watchdogs

### CHECKPOINTING



- Pontos de recuperação
- Rollback e recuperação



# FALHA POR OMISSÃO

Ocorre quando uma mensagem esperada não chega ao destino.

- **Perspectiva do receptor:** ele esperava algo e nada chegou.
- **Pode ser causada por:**
  - Interrupção no meio de transmissão (como queda de rede).
  - Tempo limite esgotado (timeout).
  - Perda da mensagem no caminho.
- **Exemplo:** Cliente envia requisição ao servidor, mas o servidor nunca a recebe (ou vice-versa).

# FALHA DE ENVIO/RECEPÇÃO

A falha ocorre no momento do envio ou da recepção da mensagem.

- **Perspectiva do emissor/receptor:**

- **Envio:** o processo não consegue enviar (ex: buffer cheio).
- **Recepção:** o processo não consegue receber (ex: erro no socket, processo não está ouvindo).

- Mais relacionada ao próprio processo do que ao canal.

- **Exemplo:** O processo A tenta enviar uma mensagem, mas falha devido a um erro interno (como falta de recurso); ou o processo B está inativo e não consegue receber a mensagem.

# FALHA DE CANAL

Ocorre quando a mensagem é corrompida, alterada ou perdida durante a transmissão.

- **Perspectiva do meio de comunicação:** O canal não entrega como deveria.
- Mais relacionada à camada de rede ou física.
- **Exemplo:** Um pacote é perdido devido a interferência em conexão Wi-Fi, ou chega com erro de verificação (checksum inválido).

# FALHA DE COLAPSO (CRASH)

Um processo ou componente deixa de funcionar repentinamente.

- **Perspectiva do sistema:** o processo simplesmente para.
- Mais comum em servidores, bancos de dados e sistemas de controle.
- **Exemplo:** Um servidor de autenticação trava durante uma requisição, e o cliente não obtém resposta

# FALHA BIZANTINA (ARBITRÁRIA)

O processo apresenta comportamento imprevisível, podendo inclusive enviar informações incorretas ou contraditórias.

- **Perspectiva dos demais processos:** comportamento anômalo e não confiável.
- Mais difícil de detectar e tratar.
- **Exemplo:** Um servidor comprometido (por falha ou ataque) responde com dados errados para diferentes clientes, sem padrão.

# DETECÇÃO DE FALHAS

A detecção de falhas é o processo de **identificar se um componente de um sistema distribuído deixou de funcionar corretamente**, seja um processo, servidor ou canal de comunicação.

## Técnicas comuns:

- **Timeout:** se o tempo para uma resposta excede o limite esperado, presume-se falha.
- **Heartbeat (batimento):** envio periódico de sinais entre componentes; ausência do sinal indica falha.
- **Protocolo de suspeição:** usa heurísticas para tratar incertezas (ex: atraso  $\neq$  falha).

## Desafios:

- Rede lenta pode simular falha (ex: timeout falso).
- Difícil distinguir entre processo lento e processo caído.
- **Ferramentas:**
  - Detecta indisponibilidade de serviços e aciona respostas

# CHECKPOINTING

Checkpointing é a técnica de **salvar periodicamente o estado de um processo ou sistema**, permitindo que ele seja restaurado em caso de falha.

- **Tipos de Checkpoint:**

- **Coordenado:** todos os processos sincronizam seus checkpoints para manter consistência.
- **Independente:** cada processo salva seu estado autonomamente (pode gerar inconsistência).
- **Incremental:** salva apenas as diferenças desde o último checkpoint.

**Finalidade:**

- Minimizar perda de progresso após falhas.
- Facilitar reinício a partir de um estado estável e conhecido.

**Exemplo prático:**

- Sistemas de HPC (High Performance Computing) como MPI usam checkpoint coordenado.
- Banco de dados fazem checkpoint em disco a cada transação confirmada.
- Jogos online salvam automaticamente o progresso a cada etapa.

# RECUPERAÇÃO DE FALHAS

Recuperação de falhas é o conjunto de **ações tomadas para restaurar um sistema ao seu estado correto após uma falha.**

- **Modos de recuperação:**

- **Reinicialização** a partir do último checkpoint salvo.
- **Reexecução de mensagens** (logs determinísticos permitem refazer operações).
- **Rollback** para estado anterior (reversão de ações parciais).

- **Importância do log:**

- Logs de eventos permitem replay seguro.
- Técnicas como *write-ahead logging* (WAL) são amplamente usadas.

## **Exemplo prático:**

- Um sistema de pagamentos refaz a última transação após uma falha usando logs.
- Softwares como PostgreSQL e MySQL usam redo logs para recuperação após crash.



# REDUNDÂNCIA E REPLICAÇÃO

**Redundância** é o uso de componentes extras para garantir funcionamento contínuo.

**Replicação** é a **cópia e manutenção de dados ou serviços em múltiplos nós.**

## Formas de Redundância:

- **De hardware:** discos RAID, fontes de energia duplicadas.
- **De software:** processos e serviços duplicados.
- **De dados:** cópias sincronizadas em múltiplos servidores.

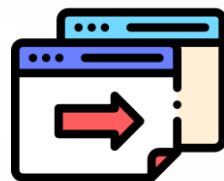
## Vantagens:

- Alta disponibilidade e tolerância a falhas.
- Balanceamento de carga.
- Continuidade do serviço mesmo com falha parcial.

## Exemplos práticos:

- DNS: replicado globalmente para garantir resolução de nomes.
- Google File System: divide e replica blocos de arquivos em diferentes nós.
- Netflix: usa clusters replicados em várias regiões com failover automático.

# ENCERRAMENTO



## Próxima Aula

Prova Prática



**[alvaro.bastos@docente.unievangelica.edu.br](mailto:alvaro.bastos@docente.unievangelica.edu.br)**