Revisão – Arquitetura de Sistemas Distribuídos

1. Fundamentos e Limites

- **Sistema**: conjunto de componentes que interagem e colaboram para atingir objetivos comuns, podendo ser tanto de software quanto de hardware.
- **Arquitetura de sistemas**: descreve como esses componentes se organizam, comunicam e se integram. A forma como essa arquitetura é definida impacta diretamente escalabilidade, confiabilidade, desempenho e segurança.
- **Modelos**:
    - **Centralizado**: processamento concentrado em mainframes; usuários acessam por terminais burros. Fácil de administrar, mas vulnerável a falhas (ponto único), caro e pouco escalável.
    - **Distribuído**: recursos e processamento espalhados em múltiplos nós interligados por redes. Favorece escalabilidade horizontal, tolerância a falhas e flexibilidade, mas aumenta a complexidade de coordenação e segurança.

- **Conceitos essenciais**:

    - **Nó**: unidade de processamento (máquina física ou virtual).
    - **Rede**: meio que conecta nós e permite troca de mensagens.
    - **Escalabilidade**:
        - Vertical → aumentar capacidade de um único servidor.
        - Horizontal → adicionar mais nós à rede.

    - **Transparência**: ocultar detalhes da distribuição (ex.: usuário não percebe em qual nó está rodando).
    - **Tolerância a falhas**: capacidade de continuar operando mesmo diante de falhas.
    - **Latência**: tempo entre envio e recebimento de uma mensagem.
    - **Concorrência**: execução intercalada de tarefas.
    - **Sincronização**: coordenação temporal de operações.
    - **Consistência**:
        - Forte: todos os nós veem a mesma versão imediatamente.
        - Eventual: os nós convergem com o tempo.
        - Causal: mantém ordem lógica de eventos relacionados.
        - Sequencial: garante ordem de execução para operações concorrentes.

- **Limites**:
    - **Físicos**: miniaturização dos transistores, barreira térmica, velocidade da luz como limite de comunicação e consumo energético crescente.
    - **Teóricos**: problemas não computáveis (Halting Problem), complexidade de algoritmos (P vs NP), indecidibilidade (Gödel) e limites da capacidade de transmissão (Shannon).
    - **Práticos**: sistemas monolíticos difíceis de escalar, altos custos de manutenção, gargalos de integração e dificuldades de adaptação a novas tecnologias.

---

2. Evolução Histórica

- **1950-70**: Mainframes (ex.: IBM System/360). Centralização extrema, alto custo, baixa tolerância a falhas e escalabilidade restrita.
- **1970-80**: Minicomputadores e LANs. Popularização em empresas, mas servidores ainda eram gargalos.
- **1980-90**: Arquitetura cliente-servidor. Parte do processamento no cliente, parte no servidor. Mlhor desempenho, mas maior risco de falhas na comunicação.
- **1990s**: Internet e sistemas corporativos distribuídos. Expansão global da conectividade.
- **2000s**: Grids e clusters. Distribuição de processamento em paralelo (HPC).
- **2010s**: Microsserviços e Cloud Computing. Elasticidade, escalabilidade sob demanda, abstração de infraestrutura.
- **2020+**: Edge computing e IoT. Processamento próximo da origem dos dados para reduzir latência e melhorar eficiência.

---

3. Processos, Threads e Comunicação

- **Processos**: instâncias independentes de programas. Possuem memória isolada, o que aumenta segurança e robustez, mas exige mecanismos de comunicação complexos (IPC).

- **Threads**: unidades de execução dentro de processos. Compartilham memória, são mais leves e rápidas, mas aumentam riscos de falhas como condições de corrida.

- **Comparação**:
    - Processos → isolamento maior, mais estáveis, mas consomem mais recursos.
    - Threads → maior desempenho e eficiência, mas mais vulneráveis a bugs internos.
- **Concorrência vs Paralelismo**:
    - Concorrência → múltiplas tarefas em progresso (turnos de CPU).
    - Paralelismo → execução simultânea real em múltiplos núcleos.
- **IPC em distribuídos**: comunicação via pipes, filas de mensagens, memória compartilhada distribuída e RPC. Essencial para coordenação entre processos e serviços espalhados.

---

4. Comunicação em Sistemas Distribuídos

- **Por que é essencial**: garante sincronização, consistência de dados, escalabilidade do sistema e tolerância a falhas.
- **Tipos**:
    - **Síncrona**: emissor bloqueia até resposta (RPC). Simples, mas pode gerar latência.
    - **Assíncrona**: emissor não espera resposta (mensagerias como RabbitMQ/Kafka). Favorece desempenho, mas aumenta complexidade de consistência.
- **Protocolos**:
    - **TCP**: orientado a conexão, confiável, ordena pacotes, usado em cenários críticos.
    - **UDP**: sem conexão, não confiável, mas rápido. Usado em jogos, VoIP, streaming.
- **Problemas clássicos**: latência de rede, perda de pacotes, duplicação de mensagens, desordenação e falhas de nós.

- **RPC (Remote Procedure Call)**: abstrai a rede permitindo chamadas de funções remotas como se fossem locais.

    - **Componentes**: stubs cliente/servidor, marshalling/unmarshalling de dados, transporte.

- **Middleware**: RabbitMQ e Kafka permitem mensageria distribuída, alta disponibilidade e filas assíncronas.
- **Streams**: fluxo contínuo de dados em tempo real, com mecanismos de offset e resiliência contra falhas.

---

5. Tolerância a Falhas

- **Princípio**: falhas são regra, não exceção (Jim Gray).
- **Tipos de falhas**:
    - **Crash**: interrupção abrupta de nó ou processo.
    - **Omissão**: mensagens não chegam.
    - **Temporais**: resposta fora do tempo esperado.
    - **Bizantinas**: comportamento arbitrário ou malicioso, mais difícil de lidar.

- **Estratégias**:

    - Replicação de serviços e dados.
    - Health checks e monitoramento contínuo.
    - Failover automático.
    - Checkpoints e logs para recuperação.
    - Retry com backoff exponencial.
    - Quórum e consenso (Raft, Paxos).
    - Multi-zonas e redundância geográfica.

6. Recuperação e Resiliência

- **Recuperação de erros**: retornar a um estado consistente após falha. Ex.: rollback em transações de banco de dados, logs de auditoria, checkpoints.

- **Resiliência**: manter funcionamento contínuo mesmo durante falha, ainda que degradado.

    - **Continuidade de serviço**: não interromper operação principal.

    - **Degradação graciosa**: redução de funcionalidades sem parar (ex.: Netflix baixa resolução em congestionamento).

    - **Elasticidade**: auto escalonamento dinâmico em cloud.

    - **Autonomia**: plataformas como Kubernetes reiniciam serviços automaticamente.

- **Estratégias práticas**: retries com backoff + jitter, replicação de dados, balanceamento de carga, monitoramento proativo e circuit breakers para evitar cascatas de falhas.

7. Segurança em Sistemas Distribuídos

- **Objetivos**: confidencialidade (dados só para quem tem permissão), integridade (dados não alterados), disponibilidade (serviço sempre acessível) e autenticidade (identidade confiável).

- **Desafios**: superfície de ataque ampliada → múltiplos nós, APIs expostas, containers, filas e bancos.

- **Autenticação e Autorização**:

    - **Protocolos**: OAuth 2.0 (autorização), OpenID Connect (identidade), JWT (tokens assinados).

    - **Ferramentas**: Keycloak, Okta, AWS Cognito.

    - **Zero Trust**: nenhum serviço é confiável por padrão, cada requisição precisa ser validada.

- **Comunicação segura**: TLS/HTTPS protege contra interceptações e ataques MITM.

- **Gestão de segredos**: cofres (HashiCorp Vault, AWS KMS, Azure Key Vault). Rotação periódica, princípio do menor privilégio e tokens temporários.

- **Ameaças**:

    - **DDoS**: ataques massivos de negação de serviço. Mitigação com rate limiting, balanceadores inteligentes, CDNs e firewalls de aplicação (WAF).

    - **Falhas internas**: configurações incorretas, credenciais expostas em repositórios, permissões excessivas em IAM. Frequentemente mais perigosas que ataques externos.

Quadro Comparativo – Arquitetura de Sistemas Distribuídos

Tema	Opção 1	Opção 2
Arquitetura	**Centralizado**: mainframe, único ponto de controle, fácil de administrar, escalabilidade vertical, ponto único de falha.	**Distribuído**: múltiplos nós, escalabilidade horizontal, tolerância a falhas, maior complexidade de segurança e coordenação.
Escalabilidade	Vertical: aumentar capacidade de um único servidor (CPU, RAM). Limitada por hardware.	Horizontal: adicionar mais nós. Mais flexível, mas exige coordenação e balanceamento.
Processos x Threads	Processos: instâncias independentes, isolamento de memória, mais seguros, comunicação via IPC, custo mais alto.	Threads: leves, compartilham memória, eficientes, comunicação fácil, mas vulneráveis a falhas internas (condições de corrida).
Concorrência x Paralelismo	Concorrência: múltiplas tarefas em progresso, alternando execução (um núcleo).	Paralelismo: execução simultânea real em múltiplos núcleos.
Comunicação	Síncrona: bloqueante (ex.: RPC). Simples, mas pode gerar latência.	Assíncrona: não bloqueante (ex.: RabbitMQ, Kafka). Escalável, mas mais complexa.
Protocolos de Rede	TCP: confiável, orientado a conexão, ordena pacotes, mais lento.	UDP: não confiável, sem conexão, não ordena pacotes, muito rápido.
Chamada Remota	RPC: chamada remota como se fosse local, mais simples, mas fortemente acoplada.	Mensageria: comunicação assíncrona via filas/streams, desacoplada, escalável.
Falhas	Crash/Omissão/Temporais: mais simples de tratar com replicação, retry, failover.	Bizantinas: comportamento arbitrário/malicioso, exigem protocolos complexos (Paxos, Raft, BFT).

Recuperação x Resiliência	Recuperação: voltar a estado consistente após falha (checkpoints, rollback).	Resiliência: manter sistema funcionando mesmo durante falha, ainda que degradado (ex.: Netflix baixa resolução).
Segurança	Modelo Tradicional: confiar na rede interna (firewalls, perímetro).	Zero Trust: nenhum nó é confiável por padrão, cada requisição precisa ser autenticada/autorizada.
Identidade	Autorização (OAuth 2.0): controla o que o usuário pode fazer.	Autenticação (OIDC + JWT): define quem é o usuário, com tokens assinados digitalmente.
Gestão de Segredos	Insegura: hardcode, arquivos .env, repositórios Git, planilhas.	Segura: cofres de segredos (Vault, AWS KMS, Azure Key Vault), rotação periódica, menor privilégio.
Ataques	DDoS: sobrecarga com milhares de requisições → defesa com rate limiting, load balancing, CDNs.	Falhas internas: permissões excessivas, credenciais expostas, configs erradas → mitigação com boas práticas DevSecOps.