Resumo Arquitetura de Sistemas

O conteúdo a seguir foi estruturado para atender aos requisitos acadêmicos e práticos, destacando explicitamente com um asterisco (*) as sessões que correspondem diretamente aos tópicos avaliados na lista de exercícios fornecida. Esta marcação visual facilita a correlação entre a teoria exposta e a aplicação prática exigida nas avaliações.

---

1. Fundamentos Computacionais: Da Centralização à Distribuição (*)

A compreensão da arquitetura moderna exige uma análise comparativa rigorosa entre os paradigmas de computação centralizada e distribuída. Esta dicotomia moldou as decisões de infraestrutura nas últimas cinco décadas.

1.1. O Paradigma do Mainframe e a Transição para Sistemas Distribuídos (*)

Historicamente, a computação empresarial era sinônimo de mainframes. Estes sistemas representam o ápice da centralização, onde o processamento, o armazenamento e a lógica de negócios residem em um único hardware robusto e monolítico.

Vantagens do Mainframe em Relação aos Sistemas Distribuídos (*)

Apesar da percepção de obsolescência, os mainframes mantêm vantagens competitivas em nichos específicos, que devem ser compreendidas pelos arquitetos de sistemas:

1. **Confiabilidade e Disponibilidade Extrema:** Mainframes são projetados para operar por décadas sem interrupção. A centralização facilita a implementação de redundância de hardware em nível de componente (CPUs, fontes, memória) dentro do mesmo chassi, oferecendo um MTBF (Mean Time Between Failures) inigualável para sistemas distribuídos de baixo custo.
2. **Segurança e Gestão Centralizada:** A superfície de ataque em um mainframe é fisicamente contida. O controle de acesso e a auditoria são simplificados, pois não há necessidade de trafegar dados sensíveis por redes públicas ou inseguras entre múltiplos nós. A gestão de *backups* e recuperação de desastres (DR) é centralizada, eliminando a complexidade de sincronizar estados distribuídos.
3. **Consistência de Dados:** Em um ambiente centralizado, transações ACID (Atomicidade, Consistência, Isolamento, Durabilidade) são garantidas pelo banco de dados local. Não há necessidade de lidar com os problemas de consistência eventual ou teoremas de CAP (Consistência, Disponibilidade, Tolerância a Partição) que assolam os sistemas distribuídos.

Desvantagens do Mainframe e a Ascensão dos Sistemas Distribuídos (*)

No entanto, as limitações do modelo centralizado impulsionaram a adoção massiva de sistemas distribuídos:

1. **Ponto Único de Falha Crítico:** Se o mainframe falhar catastroficamente (ex: incêndio no data center ou falha na placa-mãe principal), toda a operação da organização cessa. Diferentemente, sistemas distribuídos possuem tolerância a falhas inerente; se um nó falha, o sistema pode redistribuir a carga, degradando-se graciosamente.
2. **Escalabilidade Vertical Limitada e Custo (CapEx):** A escalabilidade de um mainframe é vertical (*scale-up*), limitada pelo hardware máximo que pode ser instalado em uma única máquina. O custo de aquisição (CapEx) é proibitivo, e os custos de manutenção (energia, refrigeração especializada) são elevados. Sistemas distribuídos permitem escalabilidade horizontal (*scale-out*) utilizando hardware *commodity* mais barato, crescendo linearmente com a demanda.
3. **Complexidade de Desenvolvimento em Sistemas Distribuídos:** É crucial notar que, embora resolvam problemas de escala, os sistemas distribuídos introduzem complexidade de software. Desenvolvedores devem lidar com latência de rede, falhas parciais, serialização de dados e concorrência, desafios inexistentes no ambiente de memória compartilhada do mainframe.

1.2. Arquiteturas Modernas: Centralizada, Microsserviços e Computação de Borda (*)

A evolução arquitetural não parou na distribuição cliente-servidor. Atualmente, o espectro de design de sistemas abrange desde a nuvem centralizada até a borda da rede.

Análise Comparativa das Arquiteturas (*)

A tabela abaixo sintetiza as diferenças fundamentais exigidas para a compreensão do cenário atual:

Característica	Arquitetura Centralizada (Monolítica/Cloud Central)	Arquitetura de Microsserviços	Computação de Borda (Edge Computing)
Localização do Processamento	Único data center ou cluster centralizado.	Distribuído logicamente dentro do data center/nuvem, decomposto por domínio.	Distribuído geograficamente, próximo à fonte do dado (sensores, torres 5G).

Característica	Arquitetura Centralizada (Monolítica/Cloud Central)	Arquitetura de Microsserviços	Computação de Borda (Edge Computing)
Acoplamento	Alto. Componentes compartilham memória e banco de dados.	Baixo (idealmente). Comunicação via API, bancos de dados isolados.	Baixo. Processamento autônomo local com sincronização eventual.
Latência	Alta para usuários distantes geograficamente do centro.	Dependente da rede interna do cluster e da localização da região da nuvem.	Mínima (tempo real). Processamento ocorre a milissegundos do usuário.
Foco Principal	Simplicidade de gestão e consistência forte.	Agilidade de desenvolvimento, escala seletiva e resiliência.	Redução de latência e economia de largura de banda.

1. Arquitetura de Microsserviços (*):
   - A principal distinção dos microsserviços em relação à arquitetura centralizada reside na decomposição funcional. Enquanto um sistema centralizado escala como um bloco único (replicando todo o monólito), os microsserviços permitem escalar apenas os componentes gargalo (ex: escalar o serviço de "Pagamentos" sem duplicar o serviço de "Cadastro"). A comunicação ocorre via rede (HTTP/gRPC), introduzindo desacoplamento que permite o uso de tecnologias heterogêneas (poliglotismo tecnológico).
2. Computação de Borda (Edge Computing) (*):
   - A computação de borda surge como resposta ao crescimento exponencial da IoT (Internet das Coisas). A diferença fundamental para os microsserviços (que geralmente rodam em nuvens centralizadas) é a localização física.
     - Motivação: Em cenários como carros autônomos ou cidades inteligentes, enviar dados para processamento em uma nuvem centralizada e aguardar a resposta é inviável devido à latência e ao consumo de banda.
     - Mecanismo: O processamento é movido para a "borda" (gateways locais, servidores em torres de celular). Isso permite decisões críticas em tempo real e filtra dados antes de enviá-los à nuvem, aumentando a segurança e reduzindo custos de tráfego de dados.

---

2. A Fortaleza Digital: Segurança em Profundidade (*)

A segurança em arquiteturas distribuídas não pode ser perimétrica (baseada apenas em firewalls). Ela deve ser intrínseca à aplicação, composta por autenticação robusta, controle de acesso granular e criptografia onipresente.

2.1. Criptografia: Simétrica vs. Assimétrica (*)

A criptografia é a base matemática da confidencialidade e integridade. A distinção entre os modelos simétricos e assimétricos é um conceito central na segurança da informação.

Diferença Funcional e Implicações Práticas (*)

1. Criptografia Simétrica:
   - Mecanismo: Utiliza uma única chave secreta compartilhada para cifrar (codificar) e decifrar (decodificar) a informação. Algoritmos clássicos incluem o DES e o moderno AES (Advanced Encryption Standard).
   - Vantagem (Performance): É computacionalmente extremamente eficiente. Processadores modernos possuem instruções de hardware dedicadas (AES-NI) que permitem cifrar gigabytes de dados por segundo. É ideal para proteger dados em repouso (discos rígidos, bancos de dados) e grandes fluxos de comunicação (streaming de vídeo).
   - Implicação Negativa (O Problema da Distribuição): A maior fraqueza é logística. Como o emissor e o receptor precisam ter a *mesma* chave, existe o risco inerente de interceptação durante a troca inicial dessa chave. Se a chave for comprometida em trânsito, toda a comunicação futura e passada é comprometida.
2. Criptografia Assimétrica:
   - Mecanismo: Utiliza um par de chaves matematicamente vinculadas: uma Chave Pública (disseminada livremente) e uma Chave Privada (mantida em segredo absoluto). Dados cifrados com a chave pública só podem ser decifrados pela chave privada correspondente (confidencialidade). Inversamente, dados cifrados pela chave privada podem ser verificados pela pública (autenticidade/assinatura digital). O algoritmo RSA é o exemplo mais notório, baseando-se na dificuldade de fatorar números primos grandes.
   - Vantagem (Segurança na Troca): Resolve o problema da distribuição de chaves. Duas partes podem iniciar uma comunicação segura em um canal inseguro sem nunca terem trocado segredos previamente.
   - Implicação Negativa (Custo Computacional): É algoritmicamente complexa e lenta (centenas ou milhares de vezes mais lenta que a simétrica). Não é viável para cifrar o corpo de mensagens longas ou sessões inteiras.

A Síntese: Criptografia Híbrida

Sistemas modernos (como HTTPS/TLS) utilizam uma abordagem híbrida para obter o "melhor dos dois mundos". A criptografia assimétrica é usada apenas no "handshake" inicial para autenticar o servidor e negociar de forma segura uma chave simétrica temporária

(chave de sessão). Uma vez estabelecida, a comunicação prossegue usando a criptografia simétrica de alta velocidade.

2.2. Controle de Acesso e Identidade

Além de proteger os dados, é necessário controlar quem pode acessá-los.

- Autenticação: O processo de verificar a identidade ("Quem é você?"). Práticas modernas exigem MFA (Multi-Factor Authentication), combinando algo que o usuário sabe (senha), algo que tem (token) e algo que é (biometria). Em sistemas distribuídos, o SSO (Single Sign-On) e o uso de tokens JWT (JSON Web Tokens) permitem que a identidade trafegue entre serviços sem reautenticação constante.
- Autorização (RBAC, ABAC, PBAC): Define o que o usuário autenticado pode fazer. O RBAC (Baseado em Papel) é estático e baseado no cargo; o ABAC (Baseado em Atributos) é dinâmico, considerando contexto como hora e local; o PBAC (Baseado em Políticas) usa regras lógicas centralizadas para governança.

---

3. Comunicação entre Componentes Distribuídos (*)

A eficácia de um sistema distribuído é determinada pela eficiência de sua comunicação. A escolha entre modelos síncronos e assíncronos, e o protocolo de transporte, define o acoplamento e a latência do sistema.

3.1. Comunicação Síncrona vs. Assíncrona (*)

Diferenças Fundamentais e Casos de Uso (*)

1. Comunicação Síncrona:
   - Definição: O cliente envia uma requisição e bloqueia sua execução aguardando a resposta do servidor. Existe uma dependência temporal direta; ambos os sistemas devem estar ativos simultaneamente.
   - Caso de Aplicação 1 (Interação em Tempo Real): Autenticação de usuário em um portal. O usuário precisa saber imediatamente se a senha está correta para prosseguir.
   - Caso de Aplicação 2 (Leitura de Dados Críticos): Um serviço de cálculo de preço que precisa consultar a taxa de câmbio atual antes de exibir o valor final ao cliente.
   - Risco: Acoplamento temporal. Se o serviço downstream for lento, o chamador também se torna lento (propagação de falha).
2. Comunicação Assíncrona:
   - Definição: O cliente envia uma mensagem (evento ou comando) para uma fila ou tópico e continua seu processamento imediatamente, sem esperar resposta. O processamento ocorre em segundo plano.
   - Caso de Aplicação 1 (Processamento Pesado): Geração de relatórios mensais ou renderização de vídeo. O usuário solicita a tarefa e recebe uma notificação quando estiver pronta.
   - Caso de Aplicação 2 (Desacoplamento de Microsserviços): Um serviço de "Vendas" publica um evento `PedidoCriado`. O serviço de "Logística" e "Estoque" escutam esse evento e agem independentemente. Isso aumenta a resiliência; se o estoque estiver fora do ar, ele processa a mensagem quando voltar.1

3.2. Protocolos de Webservice: SOAP, REST e gRPC (*)

Webservices são as interfaces técnicas que viabilizam essa comunicação. A lista de exercícios exige uma comparação detalhada entre as três tecnologias predominantes.

Característica	SOAP (Simple Object Access Protocol)	REST (Representational State Transfer)	gRPC (Google Remote Procedure Call)
Filosofia	Protocolo baseado em ações e contratos rígidos. Foca na exposição de métodos.	Estilo arquitetural baseado em recursos (entidades). Usa a semântica da Web.	Framework RPC moderno focado em performance e ações remotas.
Formato de Dados	XML: Verboso, pesado para parsear, mas autodescritivo.	JSON: Leve, legível por humanos, padrão de mercado para Web/Mobile.	Protobuf: Binário, comprimido, fortemente tipado. Não legível sem ferramentas.
Transporte	Agnóstico (HTTP, SMTP, JMS), mas usualmente HTTP.	HTTP/1.1 (majoritário) ou HTTP/2.	HTTP/2: Exige suporte a multiplexação e streaming bidirecional.
Contrato	WSDL: Define estritamente tipos e operações. Forte governança.	OpenAPI/Swagger: Opcional, mas boa prática. Flexível.	.proto: Contrato estrito obrigatório para gerar código cliente/servidor.
Exemplo de Uso (*)	Integrações bancárias, governamentais e sistemas legados (ERPs) onde transações ACID distribuídas e	APIs públicas (Twitter, Google Maps), backends para Front-end (SPA/Mobile) devido à facilidade de consumo universal.	Comunicação interna entre microsserviços ("East-West traffic") onde baixa latência e alto throughput são críticos.

Característica	SOAP (Simple Object Access Protocol)	REST (Representational State Transfer)	gRPC (Google Remote Procedure Call)
	segurança WS-Security são mandatórias.		

---

4. Arquiteturas de Serviço: SOA e Microsserviços (*)

A organização lógica dos serviços evoluiu para resolver problemas de escalabilidade humana e técnica.

4.1. SOA (Service-Oriented Architecture) e suas Defasagens (*)

Definição (*)

SOA é um estilo arquitetural corporativo que surgiu para integrar sistemas heterogêneos grandes e complexos. O conceito central é o reuso de serviços de negócio através de um contrato padronizado. A implementação clássica da SOA depende fortemente de um Enterprise Service Bus (ESB), um middleware centralizado responsável por rotear, transformar (ex: XML para CSV) e orquestrar mensagens entre serviços.

Defasagens Atuais (*)

A lista de exercícios questiona por que a SOA perdeu espaço para microsserviços. As razões principais incluem:

1. O ESB como Gargalo: O ESB tornou-se um ponto único de falha e de gargalo de desenvolvimento. Como continha muita regra de negócio ("Smart Pipes"), qualquer alteração exigia a intervenção de especialistas em middleware, retardando o *time-to-market*.
2. Governança Excessiva: A busca por um "Modelo Canônico de Dados" (um formato único para toda a empresa) gerou acoplamento burocrático. Mudanças simples exigiam aprovação de comitês para garantir que não quebrariam outros sistemas.
3. Tecnologia Pesada: A dependência de XML e SOAP tornou as implementações lentas e difíceis de manter em comparação com a agilidade de JSON/REST.

4.2. Microsserviços: Autonomia e DDD (*)

Conceito e Diferença para Centralização (*)

Microsserviços são uma abordagem onde a aplicação é construída como um conjunto de pequenos serviços, cada um executando em seu próprio processo. Diferente da SOA, eles defendem "Smart endpoints and dumb pipes" (lógica nos serviços, rede simples).

A diferença para a arquitetura centralizada é o isolamento de falhas e deploy. Em um monólito, um vazamento de memória em uma função derruba todo o servidor. Em microsserviços, derruba apenas aquele container específico.

Design Orientado ao Domínio (DDD)

O sucesso dos microsserviços depende do DDD para definir fronteiras corretas (Bounded Contexts). Cada serviço deve possuir seu próprio banco de dados (Database-per-service) para evitar acoplamento em nível de dados. Transações distribuídas são gerenciadas não por 2PC (Two-Phase Commit), que trava recursos, mas por Sagas, sequências de transações locais com ações compensatórias em caso de falha.

---

5. O Mundo dos Containers e Docker (*)

A revolução dos microsserviços foi viabilizada pela tecnologia de containers, que resolveu o problema de "matriz de compatibilidade" entre aplicações e infraestrutura.

5.1. Funcionamento e Vantagens dos Containers (*)

Como Funciona um Container (*)

Ao contrário do senso comum, um container não é uma mini máquina virtual. É um processo isolado no sistema operacional do host.

1. Namespaces (Isolamento de Visão): O kernel do Linux usa *Namespaces* para enganar o processo, fazendo-o acreditar que é o único na máquina. Ele tem sua própria árvore de processos (PID), sua própria rede (NET), e seus próprios pontos de montagem (MNT). O processo dentro do container não enxerga os processos do host.
2. Cgroups (Isolamento de Recursos): Os *Control Groups* limitam o quanto de CPU e memória aquele grupo de processos pode usar. Isso impede que um container "vampiro" drene todos os recursos do servidor, garantindo QoS (Qualidade de Serviço) para os vizinhos.

Vantagens (*)

- Portabilidade: O container empacota o código e todas as suas dependências (bibliotecas, runtime). Isso garante que o software rode exatamente da mesma forma no laptop do desenvolvedor e no servidor de produção.
- Eficiência: Como compartilham o kernel do host e não precisam bootar um SO completo (como VMs), iniciam em milissegundos e ocupam muito menos memória.

5.2. Instruções de Composição do Dockerfile (*)

O `Dockerfile` é a receita para construir uma imagem de container. A compreensão de suas instruções é vital para a operação eficiente.

Instrução	Explicação Resumida e Detalhes Técnicos (*)
FROM	Define a imagem base (pai) para o build (ex: `FROM alpine:latest`). É a fundação sobre a qual as camadas subsequentes são adicionadas.
RUN	Executa comandos no shell durante a fase de construção (build) da imagem. Usado para instalar pacotes (`apt-get install`), criar pastas ou configurar permissões. Cada `RUN` cria uma nova camada imutável na imagem.
COPY	Copia arquivos do diretório local (host/contexto de build) para dentro do sistema de arquivos da imagem. Essencial para injetar o código-fonte da aplicação na imagem.
WORKDIR	Define o diretório de trabalho padrão para as instruções seguintes. Funciona como um comando `cd` persistente dentro do processo de build e runtime.
CMD	Fornece o comando padrão e/ou parâmetros para executar quando o container iniciar. Pode ser facilmente sobrescrito pelo usuário na linha de comando (`docker run <imagem> <novo_comando>`). Define o comportamento padrão da execução.
ENTRYPOINT	Configura o container para rodar como um executável dedicado. O comando definido aqui não é sobrescrito facilmente; argumentos passados no `docker run` são anexados a ele. Garante que o container sempre cumpra sua função primária.

---

6. Computação em Nuvem e Escalabilidade (*)

A nuvem transformou a infraestrutura de um ativo de capital (CapEx) para um custo operacional (OpEx), oferecendo elasticidade sob demanda.

6.1. Modelos de Serviço em Nuvem: IaaS, PaaS e SaaS (*)

A lista de exercícios pede a distinção clara entre os três níveis de abstração de serviço, focando em suas diferenças e restrições.

Modelo	Definição e Controle	Diferenças e Restrições (*)
IaaS (Infraestrutura como Serviço)	O provedor entrega recursos fundamentais de computação (CPU, Disco, Rede) virtualizados. O cliente gerencia desde o SO até a aplicação.	Diferença: Controle total sobre o ambiente. Ideal para migração de sistemas legados ("Lift and Shift"). Restrições: Maior complexidade operacional. O cliente é responsável por patches de segurança do SO, backups e configuração de rede.
PaaS (Plataforma como Serviço)	O provedor entrega um ambiente de execução (runtime) e ferramentas de deploy. O cliente gerencia apenas a aplicação e os dados.	Diferença: Foco total no código. O provedor cuida do SO, atualizações e escalabilidade automática. Restrições: Menor flexibilidade. O cliente fica restrito às linguagens, bibliotecas e versões suportadas pela plataforma ("Lock-in" de plataforma).
SaaS (Software como Serviço)	O provedor entrega a aplicação completa via internet. O cliente é apenas um usuário final ou administrador de configurações.	Diferença: Zero manutenção de infraestrutura ou código. Modelo de assinatura. Restrições: Customização limitada às opções oferecidas pelo fornecedor. Dependência total da disponibilidade e segurança do provedor para acessar os dados.

6.2. Estratégias de Deploy: Recreate Deploy e suas Limitações (*)

A gestão de riscos em produção envolve escolher como atualizar o software. Embora existam estratégias sofisticadas como *Blue-Green* (zero downtime, custo dobrado) e *Canary* (teste gradual em produção), a lista de exercícios foca no Recreate Deploy.

Funcionamento do Recreate Deploy (*)

É a estratégia "tudo ou nada". O processo envolve:

1. Parar e remover todas as instâncias da versão atual (V1).
2. Somente após a limpeza, iniciar as instâncias da nova versão (V2). Isso garante que nunca haja duas versões diferentes rodando ao mesmo tempo, simplificando problemas de compatibilidade de banco de dados.1

Limitações Críticas (*)

1. **Downtime (Indisponibilidade):** A principal limitação. Existe um intervalo de tempo inevitável entre o desligamento da V1 e a disponibilidade da V2 (tempo de boot). Durante esse período, o serviço está totalmente inacessível para o usuário final, resultando em erros e perda de receita.
2. **Impossibilidade de Rollback Instantâneo:** Se a V2 falhar ao iniciar, não há uma versão V1 em espera para assumir o tráfego. É necessário reexecutar todo o processo de deploy da versão antiga para recuperar o sistema.
3. **Uso Recomendado:** Estritamente para ambientes de desenvolvimento/teste ou aplicações que não exigem alta disponibilidade (SLA baixo).

---

7. Conclusão

A análise do conteúdo da disciplina de Arquitetura de Sistemas revela uma progressão tecnológica orientada pela necessidade de lidar com a complexidade e a escala. A migração dos mainframes para sistemas distribuídos não foi apenas uma mudança de hardware, mas uma mudança de filosofia: da consistência centralizada para a disponibilidade distribuída e eventual.

Os tópicos abordados na lista de exercícios — desde a criptografia simétrica até as estratégias de deploy — são os pilares práticos dessa nova realidade. O arquiteto moderno deve dominar a segurança híbrida para proteger dados em trânsito, escolher protocolos de comunicação (REST/gRPC) adequados aos requisitos de latência, e orquestrar microsserviços em containers para garantir agilidade. A nuvem fornece a infraestrutura elástica necessária, mas exige disciplina operacional (Observabilidade, FinOps) para não se tornar um dreno financeiro. Em suma, a arquitetura de sistemas contemporânea é a arte de gerenciar *trade-offs* em um ambiente dinâmico e distribuído.

Este relatório foi compilado com base na análise profunda dos materiais didáticos fornecidos e pesquisa complementar para assegurar a cobertura técnica dos temas avaliados.