

Fundamentos de Sistemas Distribuídos e Arquitetura de Microsserviços

Pedro Ximenes

Universidade Católica de Pernambuco (UNICAP)

19 de outubro de 2026

Sumário I

- 1 Fundamentos de Sistemas Distribuídos
 - O que é (e o que não é) um sistema distribuído
 - Objetivos de projeto
 - Falácias da computação distribuída
- 2 O Papel do Middleware
 - Middleware como camada
- 3 Arquiteturas em Camadas
 - Camadas lógicas clássicas
- 4 Monolito, SOA e Microserviços
 - Monolito: o ponto de partida
 - SOA (Service-Oriented Architecture)
 - Microserviços
- 5 Decomposição de Domínio
 - Heurísticas para encontrar serviços
 - Anti-padrões recorrentes
- 6 Trade-offs de Comunicação

Sumário II

- Síncrona vs. assíncrona
- Consistência, latência e acoplamento

7 Apresentação do Projeto da 2º GQ

- Contexto e espírito do projeto
- Atividade em sala

8 Fechamento

Objetivos da aula (o que você deve dominar ao final)

Resultados de aprendizagem

Ao final desta aula, você deve ser capaz de:

- Explicar, com suas próprias palavras, o que é um **sistema distribuído** e quais são os seus objetivos de projeto.
- Caracterizar o papel de uma camada de **middleware** e justificar por que ela existe em quase todo sistema distribuído moderno.
- Comparar, em um nível arquitetural, as abordagens **monolito**, **SOA** e **microsserviços**, identificando vantagens, desvantagens e cenários de uso.
- Aplicar heurísticas de **decomposição de domínio** para encontrar limites razoáveis de serviços a partir de um problema real.
- Discutir **trade-offs de comunicação** (síncrona vs. assíncrona) e suas consequências para consistência, latência e acoplamento.
- Sair da aula com uma **visão preliminar do projeto da 2ª GQ**, ainda sem lançamento oficial, mas com possíveis domínios de aplicação já em mente.

Definindo sistema distribuído (visão do livro)

Definição operacional

Um **sistema distribuído** é uma coleção de componentes de software autônomos, executados em máquinas conectadas por uma rede, que aparecem para seus usuários como um **sistema coerente único**.

- Duas ideias-chave estão embutidas nessa definição:
 - **autonomia**: cada componente tem seu próprio ciclo de vida e recursos;
 - **coesão**: ao mesmo tempo, o usuário percebe o conjunto como um serviço único.
- O “cimento” que faz autonomia e coesão coexistirem é, quase sempre, uma camada de **middleware**.

Pergunta-guia

Sempre que estiver modelando um sistema, pergunte-se: *onde estão as fronteiras físicas (máquinas/redes) e onde estão as fronteiras lógicas (componentes/serviços)?*

Por que (quase) tudo hoje é distribuído?

- Escala de dados e de usuários cresceu muito mais rápido do que a capacidade de uma única máquina.
- Negócios exigem **disponibilidade contínua, integração entre organizações e evolução independente** de módulos.
- Dispositivos heterogêneos (web, mobile, IoT) disputam os mesmos dados.

Consequência prática

- A distribuição **não é uma escolha opcional** na maioria dos sistemas reais.
- A pergunta passa a ser **como distribuir bem, não se distribuir**.

Quatro objetivos clássicos de sistemas distribuídos

Objetivos segundo o livro-texto

- **Compartilhamento de recursos:** acesso controlado a dados, dispositivos e serviços remotos.
- **Transparência:** esconder, das aplicações e dos usuários, aspectos complexos da distribuição.
- **Abertura (openness):** interfaces padronizadas que permitem integração e evolução independente.
- **Escalabilidade:** crescer em tamanho, em área geográfica e em gerenciamento sem perder qualidade de serviço.

Atenção

Nenhum dos quatro é “de graça”. Cada um cobra um preço arquitetural: complexidade, consistência, latência, custo operacional.

Aprofundando “Abertura”: o caso do *sign-in* com Google

Objetivos segundo o livro-texto

- **Compartilhamento de recursos:** acesso controlado a dados, dispositivos e serviços remotos.
- **Transparência:** esconder, das aplicações e dos usuários, aspectos complexos da distribuição.
- **Abertura (openness):** interfaces padronizadas que permitem integração e evolução independente.
- **Escalabilidade:** crescer em tamanho, em área geográfica e em gerenciamento sem perder qualidade de serviço.

Aprofundando “Abertura”: o caso do *sign-in* com Google

Objetivos segundo o livro-texto

- **Compartilhamento de recursos:** acesso controlado a dados, dispositivos e serviços remotos.
- **Transparência:** esconder, das aplicações e dos usuários, aspectos complexos da distribuição.
- **Abertura (openness):** interfaces padronizadas que permitem integração e evolução independente.
- **Escalabilidade:** crescer em tamanho, em área geográfica e em gerenciamento sem perder qualidade de serviço.

Exemplo — “Sign-in with Google”. Seu app, o Google e o navegador não compartilham código, mas conseguem se integrar porque seguem um contrato *aberto* (OAuth 2.0 / OpenID Connect). Qualquer serviço que respeite esse padrão autentica usuários via Google *sem acordo bilateral prévio*, e é possível trocar de provedor de identidade (Microsoft, GitHub, Apple...) **sem reescrever o cliente** — exatamente a essência de *openness*.

Formas clássicas de transparência de distribuição

- Transparência não é binária: ela aparece em diferentes dimensões, e tentar garantir todas ao mesmo tempo costuma ser inviável.

Tipos mais comuns (visão de projeto)

- **Acesso:** esconder diferenças de representação e método de acesso aos dados.
- **Localização:** esconder *onde* o recurso está fisicamente.
- **Migração / realocação:** esconder que o recurso mudou de lugar.
- **Replicação:** esconder que existem múltiplas cópias.
- **Concorrência:** esconder compartilhamento por usuários independentes.
- **Falhas:** esconder a ocorrência e a recuperação de falhas.

Transparency	Description
Access	Hide differences in data representation and how an object is accessed
Location	Hide where an object is located
Relocation	Hide that an object may be moved to another location while in use
Migration	Hide that an object may move to another location
Replication	Hide that an object is replicated
Concurrency	Hide that an object may be shared by several independent users
Failure	Hide the failure and recovery of an object

Formas clássicas de transparência de distribuição

- Transparência não é binária: ela aparece em diferentes dimensões, e tentar garantir todas ao mesmo tempo costuma ser inviável.

Tipos mais comuns (visão de projeto)

- **Acesso:** esconder diferenças de representação e método de acesso aos dados.
- **Localização:** esconder *onde* o recurso está fisicamente.
- **Migração / realocação:** esconder que o recurso mudou de lugar.
- **Replicação:** esconder que existem múltiplas cópias.
- **Concorrência:** esconder compartilhamento por usuários independentes.
- **Falhas:** esconder a ocorrência e a recuperação de falhas.

Transparency	Description
Access	Hide differences in data representation and how an object is accessed
Location	Hide where an object is located
Relocation	Hide that an object has moved
Migration	Hide that an object has moved and that it is now a different object
Replication	Hide that there are multiple copies of an object
Concurrency	Hide that multiple users are using the same object
Failure	Hide that an object has failed and that it is now a different object

Exemplo — WhatsApp. Se você manda uma mensagem sem sinal, o app mostra um relógio e *reenvia sozinho* quando a conexão volta; se um servidor cai, outro assume; duplicatas são descartadas no cliente. Você vê apenas “✓” → “✓✓” → “✓✓ azul”. Toda a cadeia de falhas (rede, fila, servidor) fica **escondida** atrás dessa UX simples — essência de *transparência de falhas*.

As falácias clássicas (Peter Deutsch et al.)

Oito suposições perigosas em sistemas distribuídos

- A rede é **confiável**.
- A **latência** é zero.
- A **banda** é infinita.
- A rede é **segura**.
- A **topologia** não muda.
- Existe **um único administrador**.
- O custo de **transporte** é zero.
- A rede é **homogênea**.

Mensagem-chave

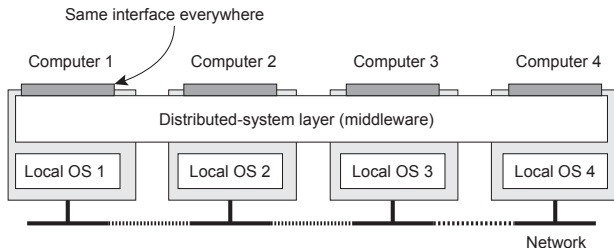
Toda arquitetura distribuída “boa” parte da hipótese de que *algumas dessas suposições vão falhar*. Projetar ignorando isso leva a sistemas frágeis e imprevisíveis.

Middleware: a camada “invisível” entre aplicação e rede

- **Middleware** é a camada de software que fica *entre* as aplicações distribuídas e o sistema operacional/rede local.
- Sua missão é oferecer **abstrações de alto nível** que escondem a complexidade da distribuição.

Responsabilidades típicas do middleware

- Expor **interfaces uniformes** (APIs) para serviços remotos.
- Esconder diferenças de **localização, replicação, concorrência e falhas**.
- Padronizar formato de dados, transporte e invocação remota.
- Oferecer serviços transversais: autenticação, transação, mensageria, nomeação.



Por que o middleware existe? (visão didática)

Cenário sem middleware

Cada aplicação teria que, sozinha:

- abrir sockets, tratar bytes, lidar com perda de pacote,
- serializar e deserializar dados,
- implementar autenticação e autorização,
- gerenciar transações, retries, timeouts,
- lidar com mudanças de localização dos serviços.

Cada **equipe** reescreveria, em cada **serviço**, a mesma lógica distribuída.

Com middleware

- Essas responsabilidades se concentram em **uma camada reutilizável**.
- A aplicação passa a focar em **lógica de negócio**, não em redes.
- Ganhos: **padronização, manutenção, velocidade** de desenvolvimento.

Exemplos reais de middleware

Classes usuais (só para contexto)

- **RPC / gRPC / Apache Thrift:** chamadas remotas tipadas.
- **Message brokers:** RabbitMQ, Apache Kafka, NATS.
- **API Gateways / Service Meshes:** Kong, Envoy, Istio, Linkerd.
- **Monitores / orquestradores transacionais:** coordenadores 2PC, sagas, etc.
- **Plataformas de dados:** bancos distribuídos, caches como Redis, serviços de nomeação.

Aviso importante do projeto

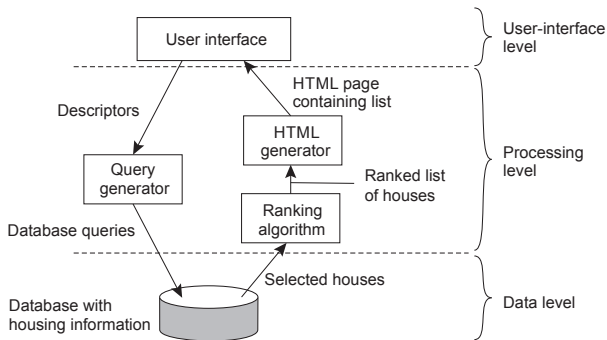
No projeto da 2ª GQ, você **não pode usar middleware pronto como solução principal**. A ideia é justamente *entender, implementar e medir* os componentes centrais do middleware.

Três camadas lógicas que aparecem em quase todo sistema

- Independentemente do estilo (monolito, SOA, microsserviços), é útil enxergar três responsabilidades:

Camadas lógicas

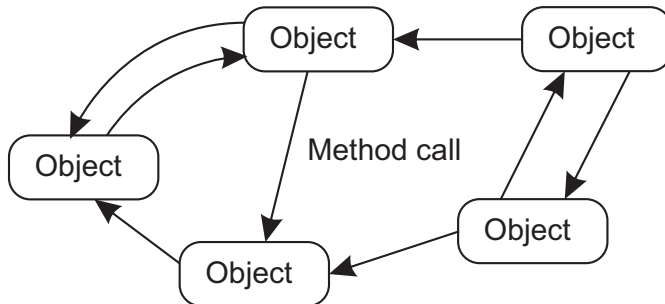
- **Interface da aplicação:** UI/endpoints, validação superficial, formatação.
- **Processamento:** regras de negócio, composição, orquestração.
- **Dados:** persistência, índices, caches, integrações com fontes externas.



Distribuição física: 2-tier, 3-tier e N-tier

Distribuição típica

- **2-tier:** cliente (UI + parte do processamento) ↔ servidor (dados e/ou processamento).
- **3-tier:** separa **servidor de aplicação** (processamento) e **servidor de dados** (persistência).
- **N-tier:** generaliza com múltiplos serviços especializados (autenticação, pagamento, recomendação etc.).



Por que 3-tier é tão comum?

- Isola lógica de negócio do armazenamento: a *mesma* lógica pode usar diferentes bancos ou consolidar vários.
- **Facilita escalonamento independente:** podemos replicar servidores de aplicação sem replicar o banco.
- É um ponto natural para inserir **coordenação** (transações, autorização, auditoria).

Conexão com o que vem à frente

Quando trocamos 1 grande servidor de aplicação por *vários* serviços menores, saímos de 3-tier tradicional e entramos no território dos **microsserviços**.

Por que 3-tier é tão comum?

- Isola lógica de negócio do armazenamento: a *mesma* lógica pode usar diferentes bancos ou consolidar vários.
- **Facilita escalonamento independente:** podemos replicar servidores de aplicação sem replicar o banco.
- É um ponto natural para inserir **coordenação** (transações, autorização, auditoria).

Conexão com o que vem à frente

Quando trocamos 1 grande servidor de aplicação por *vários* serviços menores, saímos de 3-tier tradicional e entramos no território dos **microserviços**.

Exemplo direto: Black Friday em e-commerce. O banco suporta a carga, mas o gargalo está no backend web. Em 3-tier, você sobe mais instâncias do servidor de aplicação (de 4 para 20, por exemplo) e mantém o banco como está. Escala só onde dói.

Arquitetura monolítica: o que é e quando faz sentido

Definição

Monolito é uma aplicação em que **todas** as funcionalidades são empacotadas e executadas como **uma única unidade de deploy**, mesmo que internamente esteja bem modularizada.

- Vantagens típicas:
 - simplicidade de desenvolvimento inicial,
 - *stack* única, *build* único, *deploy* único,
 - chamadas internas baratas (método, não rede).
- Desvantagens à medida que cresce:
 - qualquer mudança pequena exige redeploy do todo;
 - dificuldade de escalar partes específicas;
 - tempo de *build*/teste cresce;
 - times pisam nos pés uns dos outros no mesmo código.

Arquitetura monolítica: o que é e quando faz sentido

Definição

Monolito é uma aplicação em que **todas** as funcionalidades são empacotadas e executadas como **uma única unidade de deploy**, mesmo que internamente esteja bem modularizada.

- Vantagens típicas:
 - simplicidade de desenvolvimento inicial,
 - *stack* única, *build* único, *deploy* único,
 - chamadas internas baratas (método, não rede).
- Desvantagens à medida que cresce:
 - qualquer mudança pequena exige *deploy* do todo;
 - dificuldade de escalar partes específicas;
 - tempo de *build*/teste cresce;

Exemplo concreto: sistema interno de RH (férias, holerite e ponto) para uma empresa de 200 pessoas, com um time de 3 devs e poucas integrações externas. Um **monolito modular** é ideal: entrega rápida, operação simples e custo baixo, sem a sobrecarga operacional de microsserviços.

SOA em uma frase: serviços com contratos bem definidos

Ideia central

Estruturar o sistema como **serviços** independentes que se comunicam por **contratos explícitos**, geralmente sobre padrões abertos.

- Nasceu em contextos corporativos (anos 2000) com:
 - Web Services (SOAP/WSDL),
 - **ESB (Enterprise Service Bus)** como barramento de integração,
 - foco em **reuso** entre sistemas heterogêneos.
- Problema recorrente: o ESB tende a virar um “monstro central” — lento para mudar e difícil de operar.

Relação com middleware

Em SOA, o **middleware de integração** (ESB) é quase sempre a peça central da arquitetura e costuma acumular mais responsabilidades do que deveria.

Microserviços: motivação arquitetural

- Estilo arquitetural em que o back-end é decomposto em **muitos serviços pequenos**, autocontidos, com **deploy independente**.
- Cada serviço:
 - tem **sua própria base de código**,
 - frequentemente tem **sua própria base de dados**,
 - expõe uma API bem definida,
 - pode ser escalado individualmente.

Objetivos típicos

- Evolução independente (cada time entrega no seu ritmo).
- Escalabilidade seletiva (escala só o que está engasgando).
- Isolamento de falhas (um serviço degrada, o resto sobrevive).
- Alinhamento com **domínios de negócio**.

Microsserviços: o novo custo da complexidade

Troca fundamental

Você deixa de pagar em **acoplamento interno** e passa a pagar em **complexidade distribuída**.

- O que antes era chamada de método vira **chamada de rede**.
- Surgem problemas novos (ou ampliados):
 - **consistência** entre dados de serviços diferentes,
 - **observabilidade** (logs, métricas, *tracing* distribuído),
 - **segurança** inter-serviços,
 - **coordenação** e sagas para substituir transações ACID globais,
 - **versionamento** de contratos.

Regra de ouro

Microsserviço não é default. É uma resposta a problemas específicos (escala humana, escala técnica, domínios complexos). Se você não tem esses problemas, monolito modular costuma ser melhor.

Monolito vs. SOA vs. Microserviços (comparativo didático)

Comparativo rápido

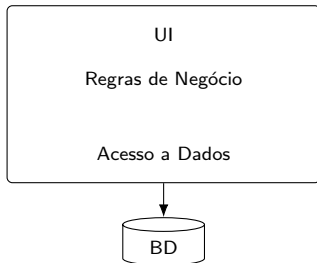
- **Unidade de deploy:** monolito (1) | SOA (vários serviços grandes) | microserviços (muitos pequenos).
- **Acoplamento:** monolito (alto interno) | SOA (forte via ESB) | microserviços (baixo, via contratos leves).
- **Complexidade de rede:** monolito (baixa) | SOA (média) | microserviços (alta).
- **Escalabilidade seletiva:** monolito (difícil) | SOA (parcial) | microserviços (forte).
- **Autonomia dos times:** monolito (baixa) | SOA (média, regulada pelo ESB) | microserviços (alta).
- **Complexidade operacional:** monolito (baixa) | SOA (alta, ESB) | microserviços (alta, plataforma).

Conclusão

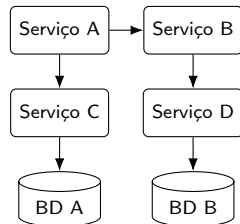
A escolha não é “qual é o melhor”, e sim **qual custo eu prefiro pagar** para os problemas que tenho agora.

Visualizando: monolito vs. microserviços

Monolito



Microserviços



- No monolito, tudo vive junto e compartilha banco.
- Em microserviços, cada serviço tem seu próprio ciclo de vida e, idealmente, seus próprios dados.

Como descobrir *quais* serviços devem existir?

Princípio geral

Siga o **domínio de negócio**, não a tecnologia. Serviços que espelham conceitos do mundo real tendem a mudar menos.

- Heurísticas úteis:

- **Bounded Context (DDD)**: cada contexto tem modelo e linguagem próprios.
- **Capacidades de negócio**: “agendamento”, “pagamento”, “catálogo” etc.
- **Eixos de mudança**: partes que mudam juntas provavelmente pertencem ao mesmo serviço.
- **Donos / times**: 1 serviço com 2 times dividindo = sinal de fronteira mal escolhida.

Pergunta prática

Se esse serviço precisasse ser explicado para uma pessoa de negócio, ele teria um nome claro e uma responsabilidade única? Se você precisa de 3 frases, o serviço está grande demais.

Exemplo de decomposição: e-commerce simples

Capacidades visíveis

- Catálogo de produtos.
 - Carrinho e pedidos.
 - Pagamentos.
 - Identidade e autenticação de usuários.
 - Entregas e rastreamento.
 - Notificações (e-mail, push).
-
- Cada um desses pode ser um **serviço**, com seus dados e regras.
 - Algumas integrações são naturalmente **síncronas** (consultar catálogo ao montar carrinho).
 - Outras são melhores **assíncronas** (enviar e-mail depois do pagamento, atualizar estoque fora do caminho crítico).

Anti-padrões: decomposições que parecem boas, mas não são

Sinais de alerta

- **Serviços CRUD por tabela:** um serviço por tabela do banco, sem ligação com negócio.
- **Serviço “deus”:** concentra regras que deveriam estar distribuídas.
- **Monolito distribuído:** “microserviços” que precisam ser deployados todos juntos para funcionar.
- **Compartilhamento de banco** entre serviços: mata a autonomia.
- **Chatty APIs:** 1 operação de negócio = N chamadas remotas encadeadas.

Regra de bolso

Se dois serviços sempre mudam juntos, eles provavelmente são um só serviço partido em dois.

Anti-padrão 1: serviço CRUD por tabela

Sinal de alerta

Um serviço por tabela do banco, sem fronteira de negócio real.

Exemplo didático

No sistema acadêmico, o time cria `AlunoService`, `CursoService`, `MatriculaService` e `ProfessorService`, cada um só expondo CRUD da própria tabela. Para fazer “matricular aluno”, a aplicação precisa orquestrar vários serviços sem que nenhum tenha a regra de negócio completa.

Por que é ruim

Você distribui a *estrutura do banco*, não o domínio. Resultado: alto acoplamento entre serviços e regras espalhadas no cliente/orquestrador.

Anti-padrão 2: serviço “deus”

Sinal de alerta

Um serviço central concentra regras de múltiplos domínios.

Exemplo didático

Em um e-commerce, o CoreService decide preço, estoque, frete, cupom, pagamento e notificação. Quase toda mudança de negócio entra nele, enquanto os demais serviços viram cascas finas.

Por que é ruim

Cria gargalo técnico e organizacional: um único time vira dono de tudo, aumenta risco de regressão e reduz autonomia dos outros domínios.

Anti-padrão 3: monolito distribuído

Sinal de alerta

Chamamos de microserviços, mas o deploy só funciona em bloco.

Exemplo didático

Para publicar uma nova versão de “Pedidos”, a equipe precisa subir Auth, Carrinho, Pagamento e Notificacao na mesma janela porque contratos quebram a cada release.

Por que é ruim

Você paga custo operacional de sistema distribuído sem ganhar deploy independente; na prática, continua com dependência de monolito, só que via rede.

Anti-padrão 4: compartilhamento de banco

Sinal de alerta

Dois ou mais serviços escrevem nas mesmas tabelas.

Exemplo didático

`ServicoPedidos` e `ServicoFaturamento` atualizam diretamente a tabela `pedido_status`. Uma alteração de coluna feita por um time quebra queries e deploy do outro.

Por que é ruim

Acoplamento por dados elimina autonomia: não existe contrato de API claro, qualquer mudança de esquema vira mudança global.

Anti-padrão 5: chatty APIs

Sinal de alerta

Uma ação simples do usuário exige muitas chamadas remotas em cadeia.

Exemplo didático

Ao abrir a tela de checkout, o front chama `/cliente`, depois `/endereco`, depois `/itens`, depois `/estoque`, depois `/frete`, depois `/cupom`. Se cada chamada leva 120 ms, a página passa de 700 ms só em rede.

Por que é ruim

Latência acumulada e maior chance de falha em cascata. Melhor compor respostas no backend (BFF/aggregator) e reduzir ida-e-volta.

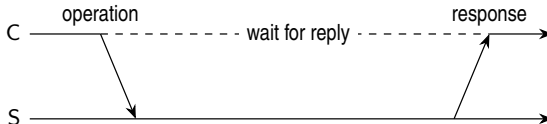
Comunicação síncrona (request-reply)

Características

- Um serviço chama outro e **aguarda** a resposta.
- Modelo mental simples (parecido com chamada de função).
- Tecnologias típicas: REST sobre HTTP, gRPC, chamadas RPC.

Custos

- **Acoplamento temporal**: os dois serviços precisam estar vivos ao mesmo tempo.
- **Latências acumulam**: cadeia longa $\Rightarrow$ p99 ruim.
- **Cascata de falhas**: um serviço lento pode derrubar toda a cadeia.



Comunicação assíncrona (mensagens e eventos)

Características

- Um serviço publica uma **mensagem** ou **evento** em um broker; outros consomem quando puderem.
- Tecnologias típicas: RabbitMQ, Kafka, NATS, Redis Streams.
- Padrões comuns: **pub-sub**, **work queues**, **event sourcing**, **request-reply assíncrono**.

Benefícios

- **Desacoplamento temporal**: o produtor não precisa esperar o consumidor.
- **Absorção de picos**: a fila segura carga que o consumidor ainda não dá conta.
- **Resiliência**: se um consumidor cai, mensagens ficam esperando.

Novos desafios

Idempotência, ordenação, deduplicação, *dead-letter queues* e *eventual consistency*.

Escolhendo síncrono vs. assíncrono (heurísticas)

Use **síncrono** quando

- A resposta é *necessária para continuar* a operação do usuário (ex.: validar pagamento antes de confirmar pedido).
- A latência ponto a ponto é baixa e controlada.
- Você realmente precisa saber, agora, o resultado.

Use **assíncrono** quando

- A ação pode ser *disparada e esquecida* (ex.: mandar e-mail, atualizar dashboards).
- O consumidor pode processar mais devagar que o produtor gera carga.
- Você quer **acoplamento fraco** entre times e serviços.

Padrão híbrido realista

Quase todo sistema robusto é **misto**: caminho crítico síncrono + efeitos colaterais assíncronos via eventos.

Os três eixos que toda decisão de comunicação toca

Três perguntas para cada integração

- **Consistência:** o quão “imediatos” precisa ser o alinhamento dos dados?
- **Latência:** quanto tempo o usuário está disposto a esperar?
- **Acoplamento:** quão livres os times/serviços precisam estar para evoluir separados?

Padrão prático

- Consistência **forte** + latência baixa + acoplamento forte $\Rightarrow$ síncrono com cuidado.
- Consistência **eventual** + picos + evolução independente $\Rightarrow$ assíncrono.

O projeto em uma frase

Ideia-central

Ao longo da 2ª GQ, cada grupo vai **projetar e implementar, do zero, um middleware distribuído** para uma aplicação de domínio livre, aplicando os conceitos da disciplina em um sistema real.

- Hoje temos uma **primeira visão do projeto**: o lançamento completo com requisitos e critérios de avaliação será na próxima aula (**22/10**).
- A intenção é que, ao sair daqui, vocês já comecem a pensar em:
 - possíveis **domínios** (e-commerce, saúde, logística, games, educação, etc.),
 - possíveis **colegas de grupo** (até 5 integrantes),
 - pontos de dúvida sobre **tecnologia** (Python, Java ou Go + Docker).

O que é *middleware* no contexto do projeto?

- Não estamos pedindo um *ESB* pronto.
- Estamos pedindo que você construa, com as próprias mãos, os mecanismos que:
 - permitam 2+ serviços **conversarem** (síncrono + assíncrono);
 - sejam **resilientes** (retries, timeouts, circuit breaker ou fallback);
 - tenham **observabilidade** básica (logs estruturados, métricas, correlation ID);
 - tenham **segurança** mínima (autenticação + autorização por papéis);
 - documentem uma **estratégia de falhas**.

Diretrizes técnicas preliminares

Backend em **Python**, **Java** ou **Go**. **Docker obrigatório**. **Proibido** usar middleware pronto como solução principal.

Atividade final: escolhendo domínios

Em duplas/trios informais (10–15 min)

Listem 2 a 3 **domínios candidatos** para o projeto, respondendo, para cada um:

- Qual o **produto/serviço** sendo modelado?
- Quais **3 a 5 capacidades** de negócio já dá para enxergar?
- Onde parece fazer sentido **comunicação síncrona**?
- Onde parece fazer sentido **comunicação assíncrona**?
- Qual é o **risco principal** desse domínio (complexidade, dados, escala)?

Para a aula prática

Anotem suas ideias; vamos usá-las como ponto de partida na **Aula Prática (26/10)**, onde definiremos grupos, domínio e arquitetura inicial.

Exemplo de recorte do projeto (modelo de referência)

Domínio exemplo: Delivery universitário

- **Produto/serviço:** app para pedido de refeições no campus.
- **Capacidades (5):** catálogo, carrinho/pedido, pagamento, entrega, notificações.

Fluxos que vocês poderiam propor

- **Síncrono (caminho crítico):** criar pedido → validar pagamento → confirmar pedido.
- **Assíncrono (efeitos colaterais):** pedido confirmado → evento para notificação e para fila da cozinha/entrega.

Risco principal + resposta técnica

Risco: provedor de pagamento indisponível no pico.

Resposta: timeout curto, retry com backoff e fallback para status “pagamento pendente” + reprocessamento assíncrono.

O que esperamos ver de cada grupo

Diagrama simples da arquitetura, lista de serviços com responsabilidade clara, contratos iniciais de API/evento e estratégia básica de falhas.

Síntese da aula

O que amarramos hoje

- Definimos **sistema distribuído** e discutimos objetivos: compartilhamento, transparência, abertura, escalabilidade.
- Vimos **por que o middleware existe** e o que ele esconde das aplicações.
- Revisamos **arquiteturas em camadas** (2-tier, 3-tier, N-tier).
- Comparamos **monolito**, **SOA** e **microserviços**, evitando a armadilha do “microserviços como default”.
- Introduzimos **decomposição de domínio** e **trade-offs de comunicação** (síncrona vs. assíncrona).
- Apresentamos a **visão inicial do projeto** da 2ª GQ.

Próxima aula (22/10)

Containers e Docker para sistemas distribuídos + **detalhamento do projeto** (enunciado, requisitos e critérios de avaliação).

Para leitura (recomendada)

- **Distributed Systems, 4ª ed.** (Tanenbaum & van Steen)
 - Cap. 1: Introdução, objetivos, transparência e tipos de sistemas distribuídos.
 - Cap. 2: Arquiteturas (camadas, cliente-servidor, pub-sub, multitier, microsserviços).
- Referências complementares (opcional):
 - *Building Microservices* (Sam Newman) — capítulos iniciais.
 - *Fallacies of Distributed Computing Explained* (Rotem-Gal-Oz) — leitura curta e direta.

Exercício de fixação

Escolha um sistema real que você usa todo dia e responda:

- Monolito, SOA ou microsserviços? Por quê?
- Onde deve haver comunicação síncrona e onde assíncrona?
- Quais seriam 3 serviços se você fosse desenhar do zero?