

# Mensageria

Prof. Rodrigo Macedo

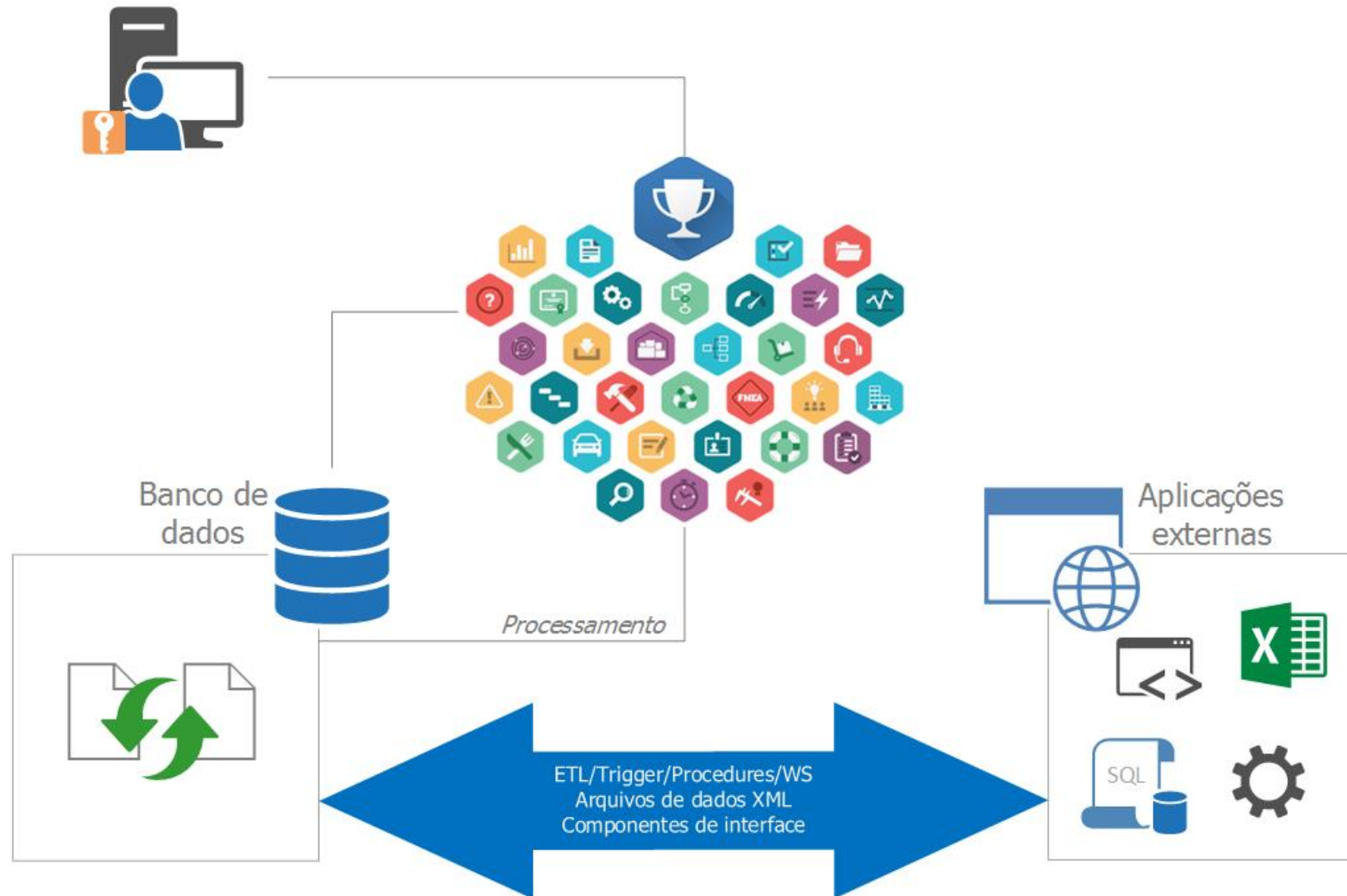
# Escopo do Curso

- Motivação
- Estratégias Mensageria
- Componentes
- Protocolos
- Microssserviços



# Motivação

- Integrar sistemas não é uma tarefa simples.
- Temos que lidar com múltiplas aplicações executando em diferentes ambientes de execução, escritos em várias linguagens e não raramente são sistemas restritos, caixas fechadas que não permitem a desenvolvedores ter acesso ao código fonte, que raramente é projetado para integrar com sistemas externos.



# Motivação

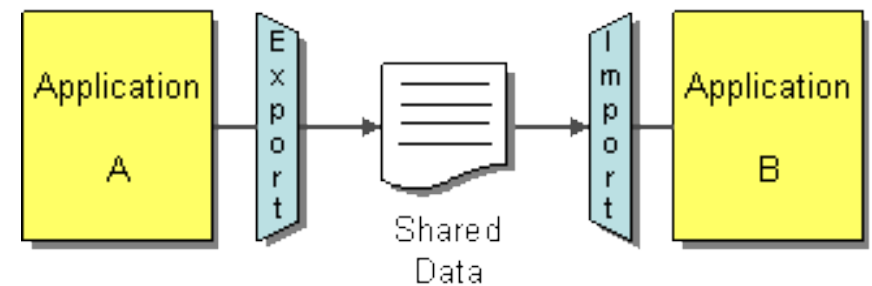
- **Interoperabilidade** entre aplicações é a palavra de ordem.
- Diante de um cenário em que os aplicativos estão em um ambiente heterogêneo e precisam se comunicar entre si, qual melhor estratégia?



# File Transfer

- Nessa estratégia, aplicações escrevem dados em arquivos para que outras aplicações possam ler.
- Um arquivo é um mecanismo universal disponível em qualquer sistema operacional que se preze. Sabendo disso, a abordagem mais simples possível de se compartilhar dados entre aplicações seria através arquivos. Cada aplicação produz um arquivo contendo informações que outras precisam consumir.

A grande vantagem aqui é que os sistemas não precisam conhecer a implementação um do outro, basta usarem um formato de arquivo comum.



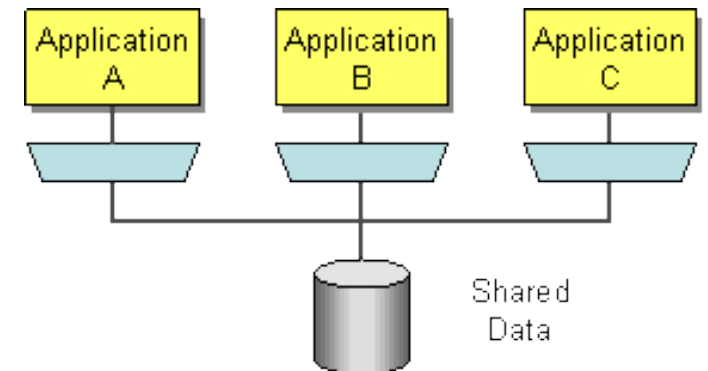
# File Transfer - Problemas

- Não há convenção de nomes ou formatos, a aplicação escrevendo dados em um arquivo deve conter alguma estratégia implementada para manter únicos os nomes de cada arquivo, bem como controlar seus diretórios.
- Problemas de concorrência: criar um mecanismo de lock para que aplicações não tentem ler um arquivo que ainda está sendo escrito por outra aplicação.
- Problemas de sincronização: um cliente atualiza seu endereço em dois sistemas diferentes da empresa. Em um dos sistemas houve um erro no momento da escrita dos dados no arquivo. Agora temos dois endereços diferentes para o mesmo cliente.

# Shared Database

- Nessa estratégia, aplicações armazenam dados em um repositório compartilhado entre aplicações interessadas.
- Os dados estão submetidos a um processo de escrita e leitura transacional, tratar erros aqui ficou mais bem simples e assim como os arquivos, bancos de dados relacionais são amplamente suportados por qualquer plataforma de desenvolvimento decente.

Com essa estratégia, temos dados sincronizados, estruturados em formato comum bem definido e sempre consistente.



# Shared Database - Problemas

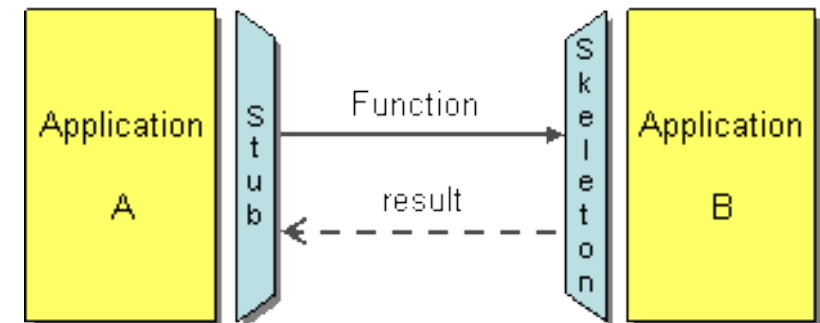
- Chegar a um design consistente para o esquema da base de dados única atendendo a múltiplos sistemas é muito difícil.
- Imagine contextos de negócio diferentes onde em um departamento da empresa o que se conhece por “produto” é um item qualquer em estoque, no outro, somente itens com um registro específico e um número SKU.
- Ter diferentes aplicações lendo e escrevendo em um mesmo banco de dados pode criar um gargalo de performance.
- Quando temos aplicações em locais diferentes do mundo, lendo e escrevendo no mesmo banco de dados, provavelmente também enfrentaremos problemas de lentidão nas operações.



# Remote Procedure Invocation

- Nessa estratégia, Aplicações expõem funções para que outras aplicações possam invocá-las remotamente.
- Remote Procedure Invocation (também conhecido como RPC, Remote Procedure Call) é o estilo de comunicação que utiliza o princípio do encapsulamento para compartilhar ações com outros sistemas.
- A ideia é que sempre que uma aplicação precise de informação disponível em outra ela faça essa solicitação diretamente.

Com essa estratégia, cada aplicação continua tendo sua lógica de implementação interna, seu próprio modelo de dados com seus formatos e particularidades.



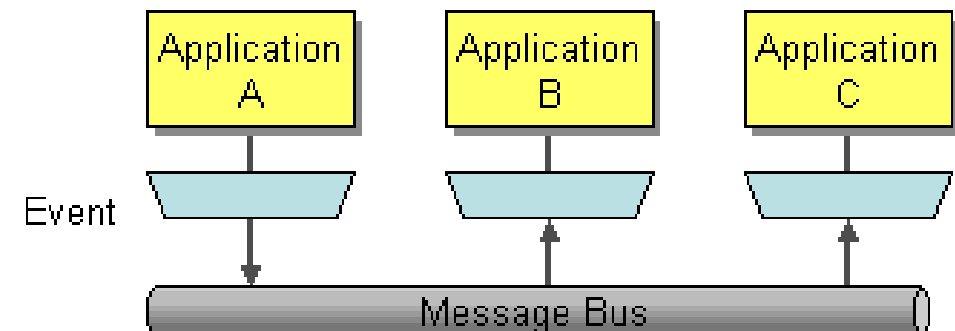
# Remote Procedure Invocation - Problemas

- A ideia do encapsulamento no RPC reduz o acoplamento entre sistemas removendo a base de dados única e compartilhada, mas cria outro acoplamento entre seus comportamentos definidos em interfaces.
- Desenvolvedores agora precisam se preocupar em escrever funções que atendem chamadas locais e chamadas remotas, que podem exigir particularidades de implementação que só fazem sentido para um sistema ou outro.
- A complexidade de se desenvolver funções sabendo se serão ou não utilizadas remotamente por um sistema ou outro, é mais uma armadilha que pode (e provavelmente vai) trazer problemas de manutenção.

# Mensageria

- Nessa estratégia, aplicações criam pequenos pacotes de dados que são enviados a canais, responsáveis pela entrega dos pacotes a sistemas interessados.
- Sistemas assíncronos de mensageria surgem como reação aos problemas que vimos nos modelos anteriores de integração.
- Postando mensagens em um canal (Message Channel), sistemas enviam mensagens que podem ser recebidas por um ou mais sistemas interessados (broadcasting).

Utilizando sistemas de mensageria temos a vantagem da assincronia. O sistema A não precisa esperar a resposta do sistema B para continuar seu processamento.



# Mensageria - Problemas

- Essa camada de integração que desacopla os sistemas conectados também carrega consigo uma série de novos conceitos bem particulares que precisam ser compreendidos para bem projetarmos sistemas de mensageria.
- Como um sistema envia um pacote de dados para outro?
- Envia-se uma mensagem a um canal (**Message Channel**) para que outro sistema se conecte ao canal e receba a mensagem.
- Como um sistema sabe para onde deve enviar a mensagem?
- Pode enviar os dados a um roteador (**Message Router**) que irá direcionar a mensagem ao devido recebedor.
- Como um sistema sabe qual deve ser o formado dos dados?
- O **Message Translator** converterá os dados de modo que o recebedor compreenda a mensagem.
- Como conectar minha aplicação a um sistema de mensageria?
- Pode ser implementado um **Message Endpoint** para enviar ou receber mensagens.

# Canal

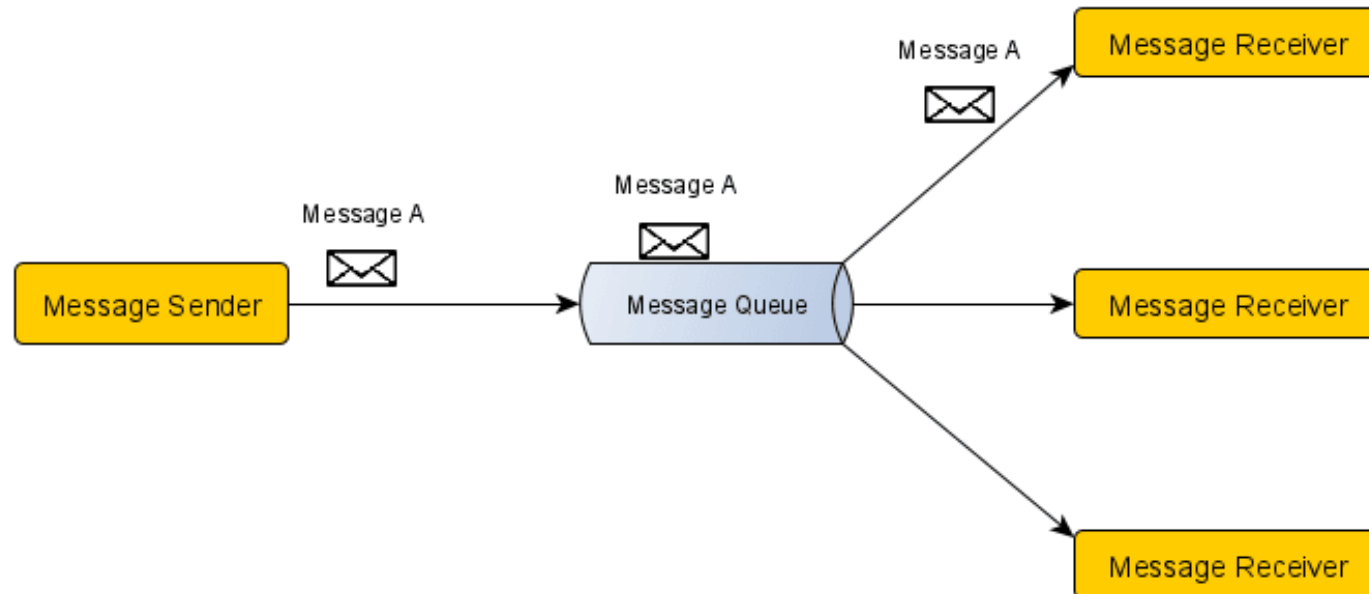
- Uma aplicação não envia uma nova mensagem de forma aleatória a um sistema de mensageria esperando que ela mágicamente seja entregue ao sistema de destino.
- A aplicação envia os dados a um canal de mensagens, este canal de mensagens tem um nome, e o sistema que deseja receber uma mensagem deve conhecer o nome que identifica este canal.
- Imagine um sistema A que receba os dados de compra de um cliente e deve enviar estes dados ao sistema B para que este, envie um email notificando o cliente que a compra foi recebida e está sendo processada.
- O sistema A deve criar uma mensagem contendo os dados da compra, e enviar a mensagem ao canal **“email.selling”**.
- O sistema B deve escutar mensagens disponíveis no canal **“email.selling”** e assim que encontrar, realizar o processamento (no caso, enviar o email contendo os dados presentes na mensagem).

# Canal

- Canais são endereços lógicos presentes em um sistema de mensageria.
- Cada sistema de mensageria implementa seus canais de uma forma, alguns podem garantir que mensagens postadas em um canal sejam entregues ao destino em ordem de chegada, outros podem por exemplo priorizar o tamanho das mensagens enviando primeiro a de menor tamanho.
- Canais de mensagens podem ser criados e configurados via interface gráfica administrativa ou utilizando APIs fornecidas pelo fornecedor do sistema de mensageria utilizado.
- Precisamos manualmente criar os canais que devem contar com seus nomes de identificação e suas configurações específicas, como política de envio, protocolo, políticas de retentativa de envio em caso de falha e etc.

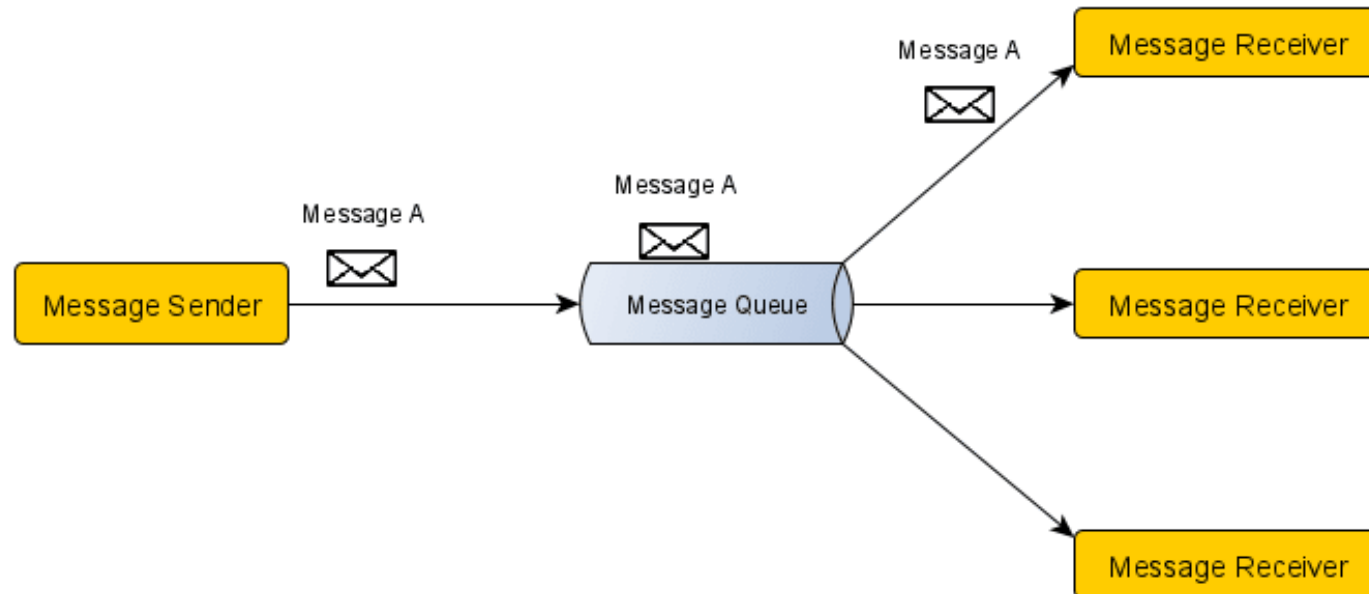
# Canal - Arquitetura

- **Ponto a Ponto:** No modelo ponto a ponto, a mensagem é enviada do remetente da mensagem para apenas um destinatário, mesmo que muitos destinatários da mensagem estejam atendendo na mesma fila de mensagens.
- No modelo ponto-a-ponto, os termos remetente da mensagem e receptor da mensagem são geralmente aplicados em vez de publicador da mensagem e consumidor da mensagem.



# Canal - Arquitetura

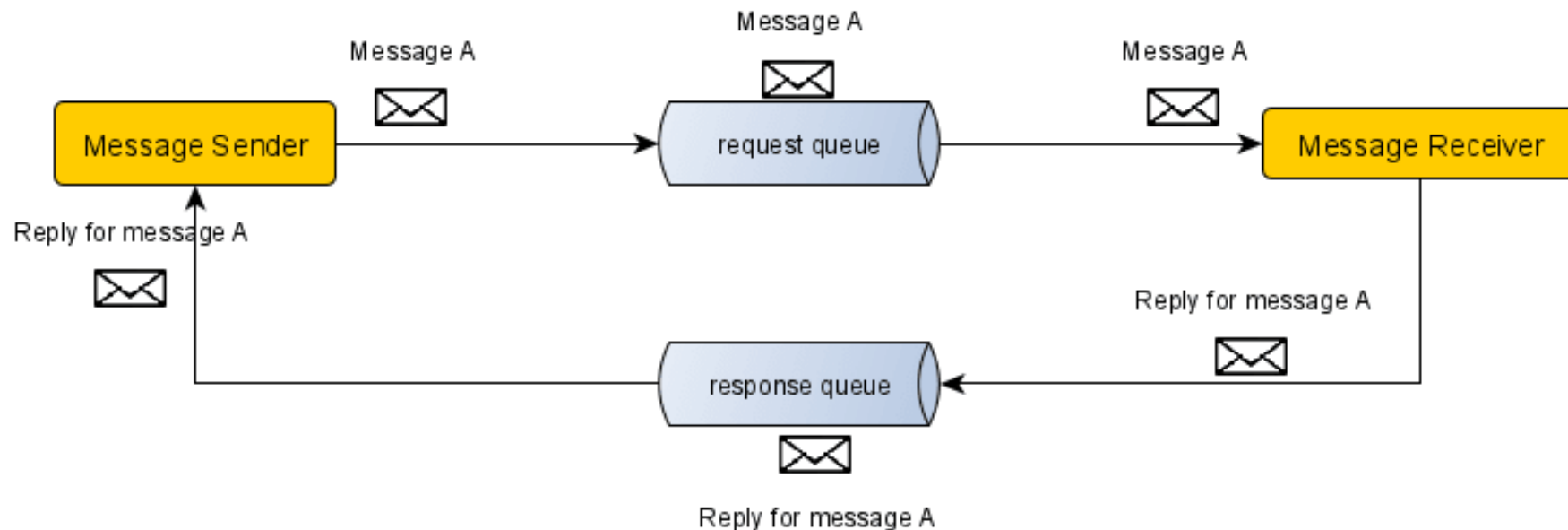
- **Ponto a Ponto:** No modelo ponto a ponto, a mensagem é enviada do remetente da mensagem para apenas um destinatário, mesmo que muitos destinatários da mensagem estejam atendendo na mesma fila de mensagens.
- No modelo ponto-a-ponto, os termos remetente da mensagem e receptor da mensagem são geralmente aplicados em vez de publicador da mensagem e consumidor da mensagem.





# Canal - Arquitetura

- **Requisição - Resposta:** no modelo de mensagens de solicitação/resposta, o remetente da mensagem envia uma mensagem em uma fila e, em seguida, aguarda a resposta do destinatário. com este modelo, o send se preocupa com o status da mensagem recebida ou ainda não.

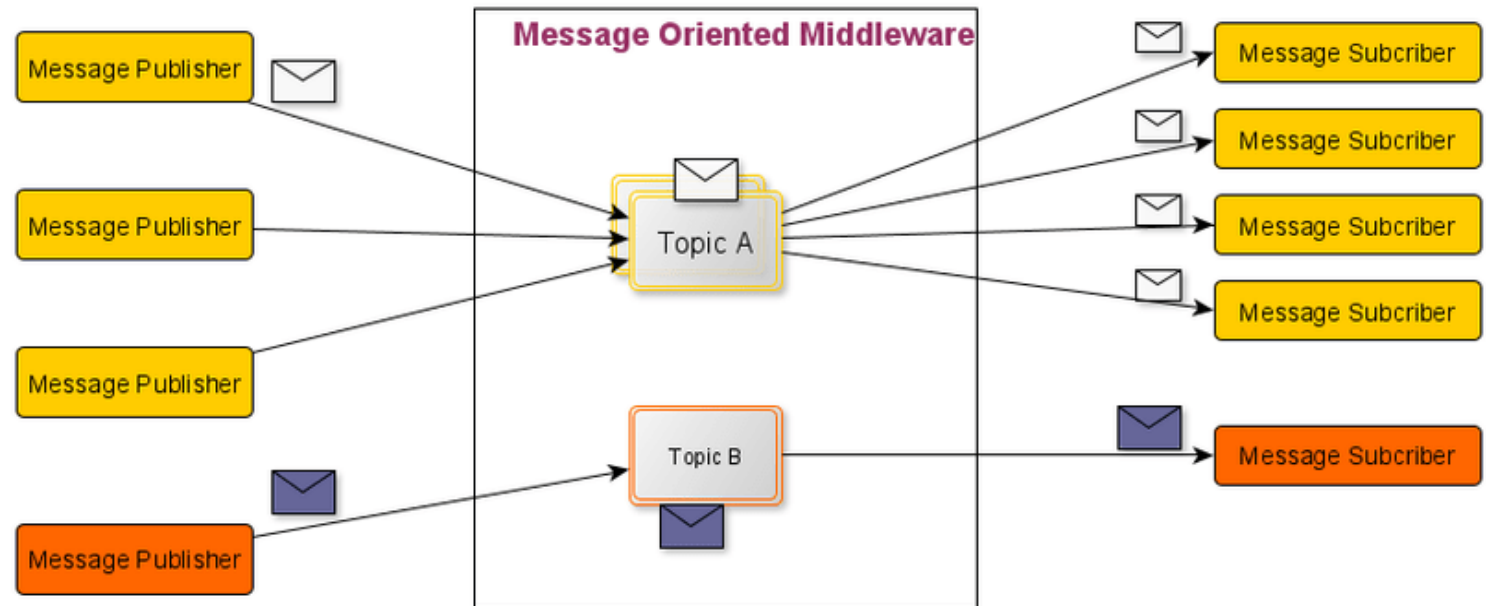


# Canal - Arquitetura

➤ **Publish - Subscribe:** funciona de forma assíncrona. Nesse domínio, os produtores de mensagens são chamados de publicadores e os consumidores de mensagens são chamados de assinantes. O editor produz mensagens para um tópico e todos os assinantes que se inscreveram nesse tópico receberão as mensagens de envio e as consumirão.

- Uma mensagem pode ser enviada para muitos consumidores.

- Os publicadores da mensagem não precisam saber onde a mensagem será consumida.
- Proporciona baixo acoplamento entre os agentes.



# Mensagem

- Mensagens são enviadas a um canal através de um processo de empacotamento na aplicação sender, e é recebida em uma aplicação receiver através de um processo de desempacotamento.
- Antes de enviar dados a um canal a aplicação sender deve transformar seus dados em uma mensagem.
- Após receber uma mensagem de um canal, uma aplicação deve transformar a mensagem em dados que possa compreender e processar.
- Esses processos são conhecidos como processos de empacotamento e desempacotamento.

# Mensagem

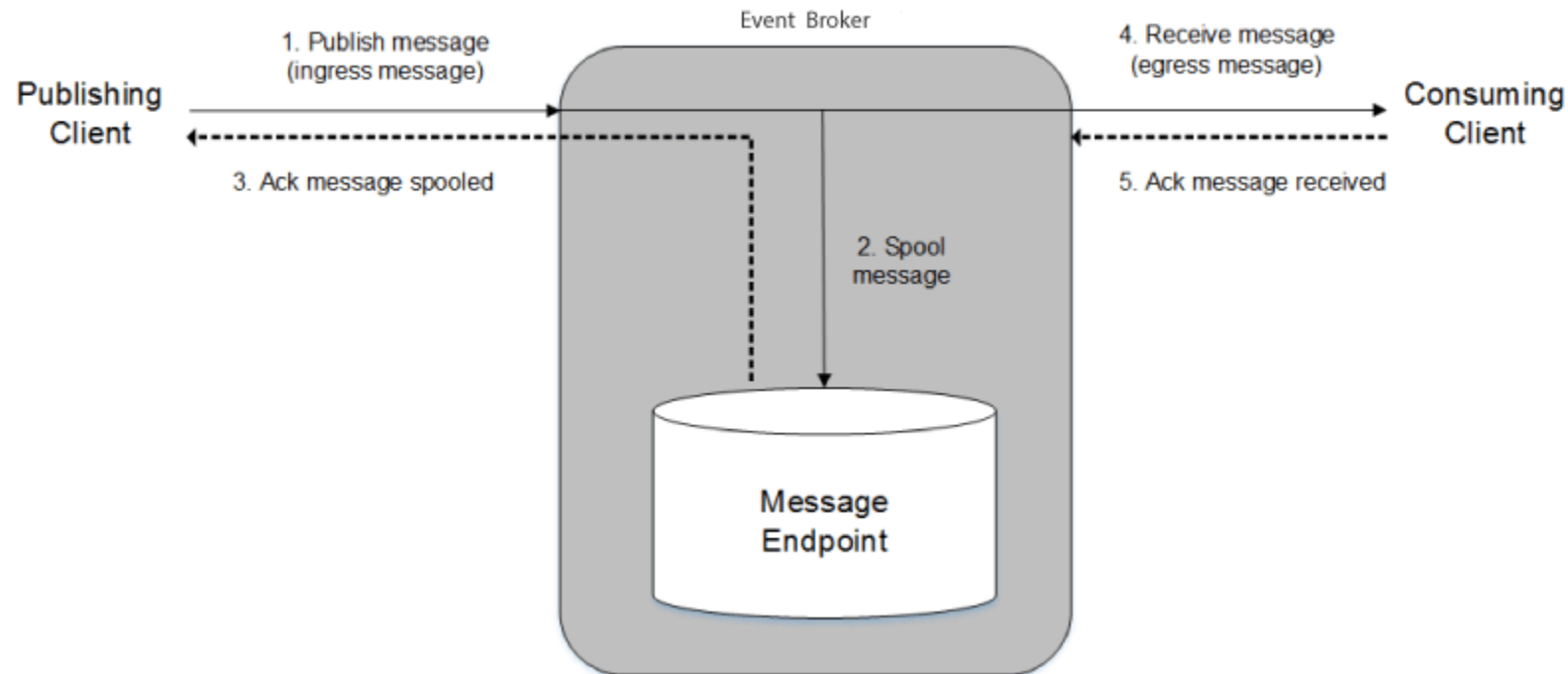
- Uma mensagem consiste em duas estruturas básicas:
  1. **Header:** Um cabeçalho contendo informações utilizadas pelo sistema de mensageria, que descrevem os dados sendo transmitidos, sua origem, destino, tamanho e etc.
  2. **Body:** Os dados propriamente ditos. Esta parte geralmente é ignorada pelo sistema de mensageria, que apenas transmite o conteúdo de uma mensagem sem qualquer verificação.
- Para sistemas de mensageria todas as mensagens são iguais: um cabeçalho e um corpo contendo dados que devem ser entregues a seu destino.

# Endpoint

- Chamamos de Message Endpoint a parte de uma aplicação que encapsula a implementação da comunicação com o sistema de mensageria.
- Este conceito básico utilizado por qualquer aplicação que troque mensagens com outros sistemas, nos permite ter a flexibilidade de alterar o sistema de mensageria a qualquer momento sem que toda implementação de uma aplicação seja afetada.
- Um Message Endpoint bem projetado possibilita a utilização de padrões de consumo seletivo de mensagens, onde podemos processar ou descartar determinadas mensagens de acordo com seus dados.

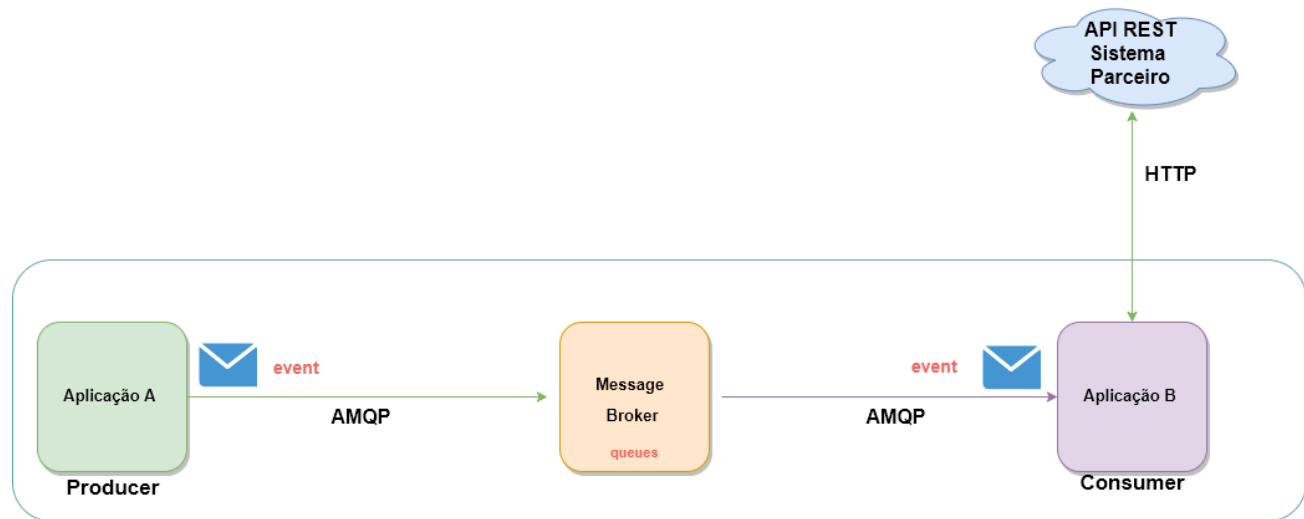
# Endpoint

- Temos um componente específico responsável por conhecer aspectos relacionados ao sistema de mensageria, como o formato das mensagens e os endereços dos canais, enquanto todo restante do sistema só precisa conhecer suas regras de negócio.



# Message Broker

- Um Message Broker nada mais é que um servidor de mensagens, responsável por garantir que a mensagem seja enfileirada e armazenada em disco (opcional), garantindo que ela fique lá enquanto necessário até que alguém (consumidor) a retire de lá.
- Em outras palavras, é como se fosse uma caixa de correio, e as mensagens são as cartas que serão depositadas(publicadas) ali e retiradas(consumidas) por alguém que tenha interesse em ler essas cartas.



# Message Broker

- **Event:** um evento é a mensagem em si. Pode ser um JSON, XML ou qualquer tipo de formato em bytes.
- **Producer:** é a aplicação que envia uma mensagem para uma queue do Message Broker.
- **Queue:** é uma fila que recebe as mensagens geradas por um producer. As mensagens ficarão dentro da fila até que alguma aplicação consumidora (consumer) retire a mensagem da fila.
- **Consumer:** é a aplicação que consumirá as mensagens que estão presentes na fila.



# Message Broker - Vantagens

- A Aplicação A não precisa se preocupar se a Aplicação B vai estar disponível no momento em que ela enviar o evento.
- Os eventos gerados podem ser persistidos em disco pelo Message Broker.
- Caso o consumer não consiga confirmar a leitura da mensagem (ack) a mesma continuará enfileirada, até que em outro momento ele consiga confirmar a leitura.
- Baixo acoplamento. Note que a Aplicação A, nem sabe como a Aplicação B foi desenvolvida, se foi com C#, JavaScript, Ruby ou seja lá qual for a linguagem.
- A única responsabilidade da Aplicação A é comunicar que houve um evento e que mesmo deve ser integrado quando possível.

# Protocolos

- **Java Message Services (JMS):** Como o próprio nome diz, voltado para integrações entre serviços ou componentes desenvolvidos em Java. Muito usado também em Application Servers onde aplicações corporativas desenvolvidas em Java são implantadas. Foi uma dos primeiros protocolos com esse propósito.
- **Advanced Message Queuing Protocol (AMQP):** Um dos protocolos mais atuais e indiferente da linguagem de programação usada no desenvolvimento de aplicações. Se a sua linguagem favorita para desenvolvimento tem suporte a esse protocolo, suas aplicações conseguem se comunicar através de um Broker que usa o mesmo protocolo.

# Protocolos

- **Message Queue Telemetry Transport (MQTT):** Muito utilizado para comunicação de dispositivos inteligentes (IoT). Com esse protocolo, fica mais fácil a implementação de objetos se comunicarem em rede e enviarem mensagens, seja entre objetos ou mesmo para outros sistemas que precisam de notificações desses dispositivos.
- **Simple/Streaming Text Oriented Messaging Protocol (STOMP):** voltado para Streaming de dados, onde os mesmos são serializados e quando a informação precisa ser entregue praticamente em tempo real para os consumidores.
- **Microsoft Message Queuing (MSMQ):** é uma implementação Microsoft, bastante usado em aplicações desenvolvidas em linguagem proprietária da empresa. Uso muito específico da plataforma Microsoft.

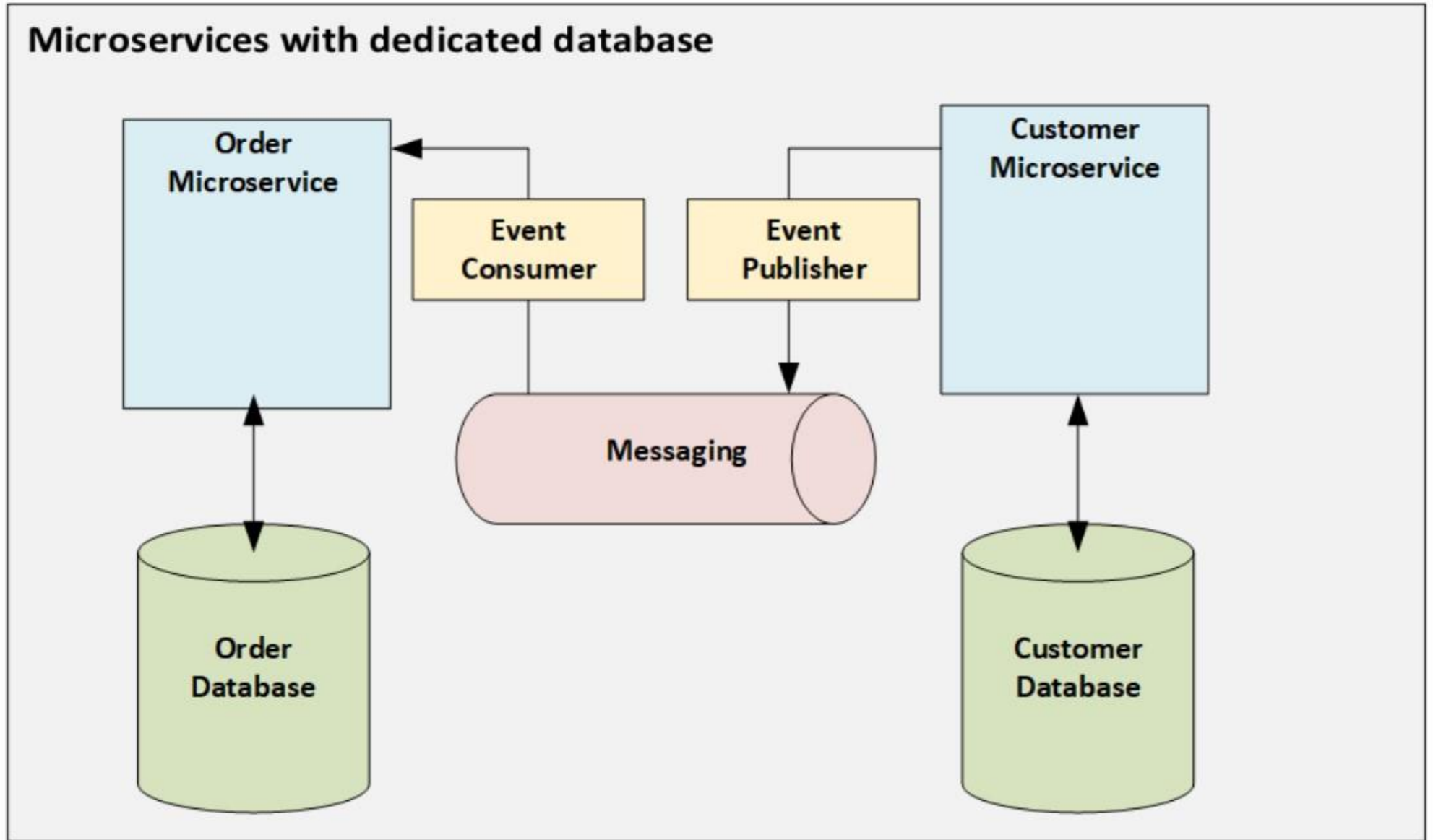
# Protocolos

- **Java Message Services (JMS):** Como o próprio nome diz, voltado para integrações entre serviços ou componentes desenvolvidos em Java. Muito usado também em Application Servers onde aplicações corporativas desenvolvidas em Java são implantadas. Foi uma dos primeiros protocolos com esse propósito.
- **Advanced Message Queuing Protocol (AMQP):** Um dos protocolos mais atuais e indiferente da linguagem de programação usada no desenvolvimento de aplicações. Se a sua linguagem favorita para desenvolvimento tem suporte a esse protocolo, suas aplicações conseguem se comunicar através de um Broker que usa o mesmo protocolo.

# Mensageria x Microserviços

- O tema mensageria também está muito presente atualmente, quando estamos trabalhando em projetos relacionados a Arquitetura de Microserviços.
- Uma vez que cada microserviço deve ser independente um do outro, mas mesmo assim, ainda temos que compartilhar dados ou mesmo enviar dados de um serviço para o outro ou mesmo alertar que uma ação aconteceu.
- Usando mensageria, pode resolver vários desses cenários de comunicação entre microserviços sem gerarmos acoplamento entre eles, e assim, continuaremos com eles independentes uns dos outros.

# Mensageria x Microserviços



# Questão Concursos

**Q1) [CESPE TJ-SE 2014]** Acerca dos conceitos de Datawarehouse, de Datamining e de mensageria, julgue os itens a seguir.

O modelo de mensageria publish/subscribe é adequado para comunicação síncrona entre o remetente e o receptor das mensagens.

**Q2) [CESPE SLU-DF 2019]** Com relação a desenvolvimento de software, julgue o item a seguir.

Na arquitetura de API JMS (Java Message Service) e no modelo Publish/Subscribe (Pub/Sub) de troca de mensagens, uma mensagem publicada em um tópico será entregue a uma única aplicação consumidora.

# Questão Concursos

**Q1) [CESPE TJ-SE 2014]** Acerca dos conceitos de Datawarehouse, de Datamining e de mensageria, julgue os itens a seguir.

O modelo de mensageria publish/subscribe é adequado para comunicação síncrona entre o remetente e o receptor das mensagens. **ERRADO.**

**Q2) [CESPE SLU-DF 2019]** Com relação a desenvolvimento de software, julgue o item a seguir.

Na arquitetura de API JMS (Java Message Service) e no modelo Publish/Subscribe (Pub/Sub) de troca de mensagens, uma mensagem publicada em um tópico será entregue a uma única aplicação consumidora. **ERRADO.**