**Эволюция архитектуры**

**1990’s**

**Spaghetti-Oriented Architecture**

Когда было только процедурное программирование, и не было классов. Был только набор методов и все они пересылали данные друг другу, и по сути каждый мектод был завязан на другом.

Spaghetti code is a phrase used for unstructured and difficult-to-maintain source code. Like a spaghetti, the flow of code is tangled and it’s hard to figure out the relation between different pieces of code. For example, code that includes a lot of GOTO statements instead of structured programming constructs is spaghetti code – the resulting code is convoluted and difficult to understand.

**2000’s**

**Lasagna-Oriented Architecture (aka layered monolith)**

В этой архитектуре мы пишем слои, что бы все было разделено по каким то правилам.

Lasagna code is used to describe code that is structured, understandable, and layered. Lasagna code, although structured, is monolithic and difficult to modify. The layers are so complicated and intertwined that making a change in one layer may necessitate changes in all other layers. An example of this type of code is an object-oriented code with a lot of small classes and several unpredictable interdependencies between them.

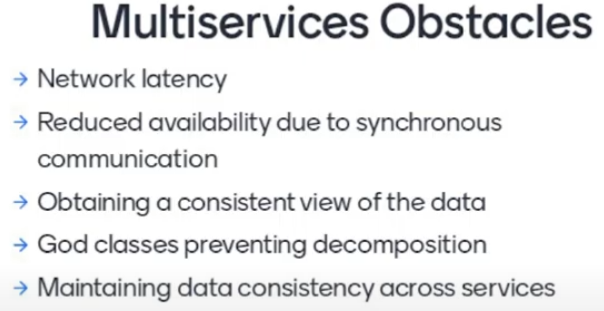
**2010’s**

**Ravioli-Oriented Architecture (aka microservices)**

Когда мы разбиваем всю логику на много маленьких самодостаточных кусков, которые можно переиспользовать

Ravioli code is the ideal structure for your software. It is specific to object-oriented programming and refers to code that are comprises of well-structured classes that are easy to understand in isolation. In ravioli code, each of the components, or objects, are self-contained; ​so, any component can be modified or replaced without significantly affecting other components.

**Недостатки SOA**



Если например у нас в одном сервисе лежит адрес клиента, в другом сервисе лежат деньги клиента, и тд, то выполняя транзацию на покупку какого то товара, очень легко запутаться и выполняя распределенную транзакцию, очень просто нарваться на неконсистентность данных – race condition.

Еще пример – создаваемый заказ может “перебиться” другим заказом

**Коммуникация между микросервисами**

**Synchronous**

Банально вызываем API

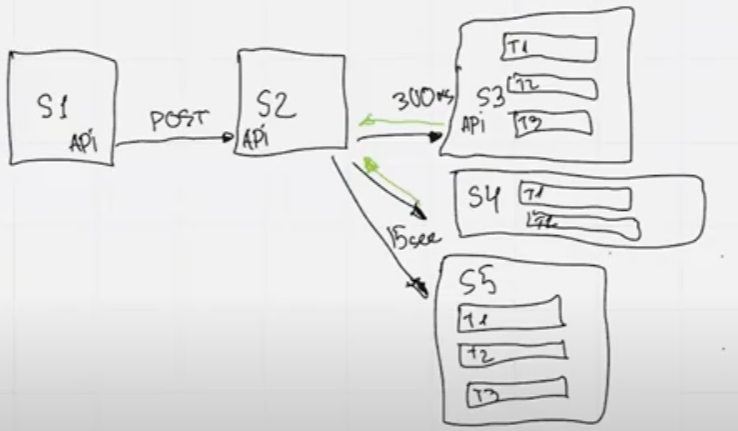
**Asynchronous**

**One-to-one**



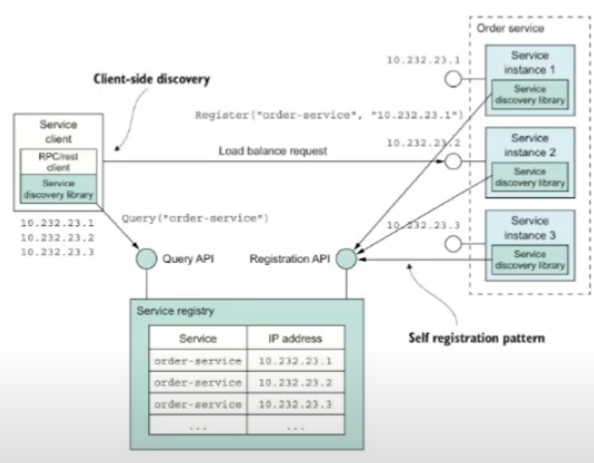
**One-to-many**

Например, если у нас есть какие то сервисы, которые выполняют запросы долго, и нам не обязательно знать его сразу



**Service Discovery**

Если у нас есть большое кол-во сервисов, нужна система **Service Discovery**

**Self registration**

Сервису нужно понимать, что сервис который зарегестрировался у нас еще живой

**Client side**

Например DNS record. Например если сервисы лежат в разных регионах и разных дата центрах

**Server side**

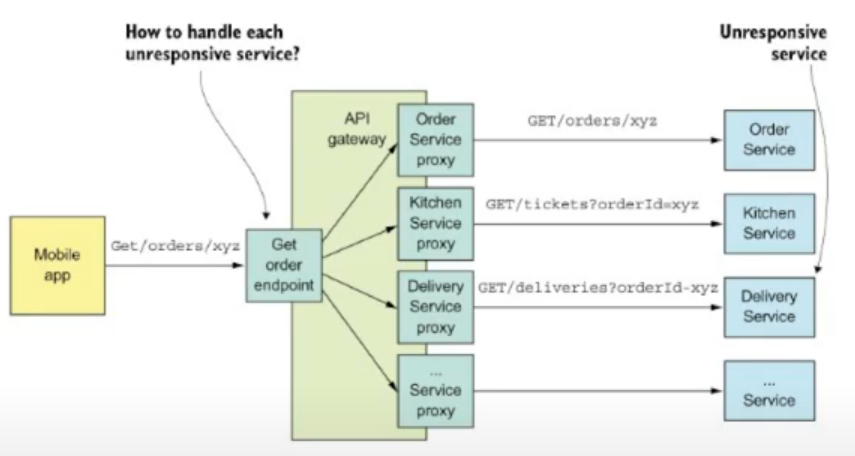
Сервер сам менеджит таски внутри себя. Внутри есть load balancer

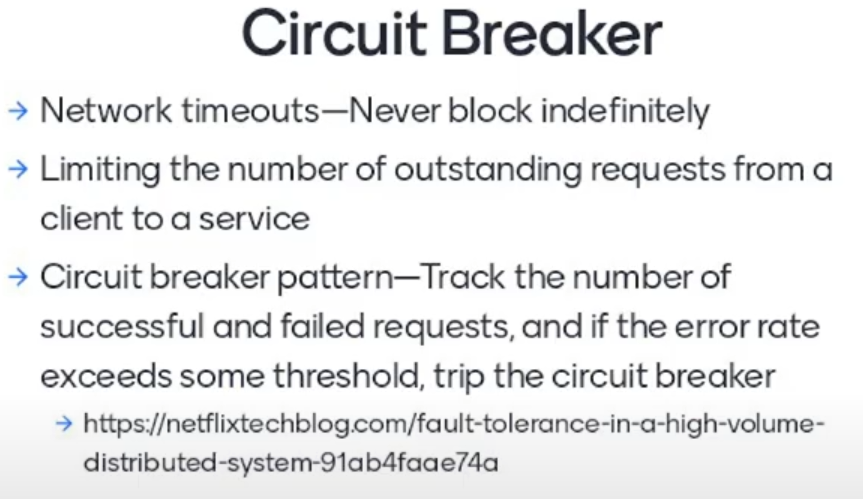
**3rd party registration**

Независымый сервис, который сам поднимает нужные такси и регистрирует их в load balancer. Независымый от инфрастуктуры.

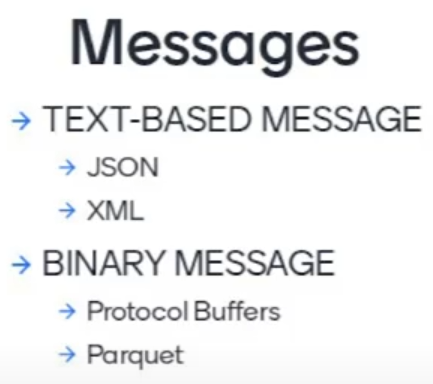
**Как хэндлить неработающие сервисы**

Паттерн **Circuit Breaker**





**Обмен сообщениями между сервисами**

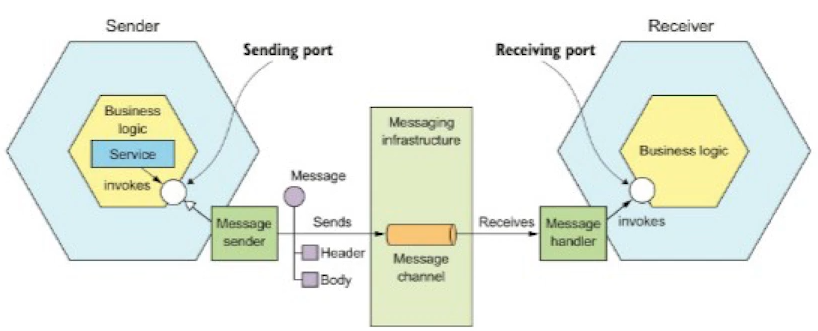


Бинарные форматы работаю быстрее. Стоит использовать если мы передаем много информации и тратим много времени на сериализацию. Но тогда нужны специальные инстурменты, так как в логах сообщения бинарных форматов будет сложно прочитать (они могут быть представлены в виде двоичного кода)

**Типы сообщений**

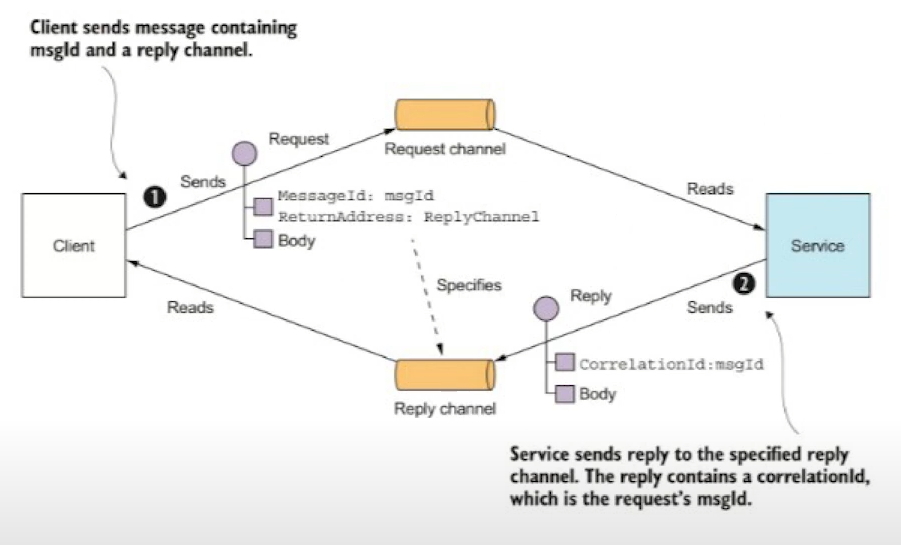
* Документ – данные которые мы хотим переслать от одного сервиса к другому
* Комманда – когда мы не просто передаем данные, а выполняем операцию (например POST/PATCH/… запросы)
* Событие (event) – сообщение которое говорит что что то произошло (например поменялся статус)

**Каналы передачи сообщений**



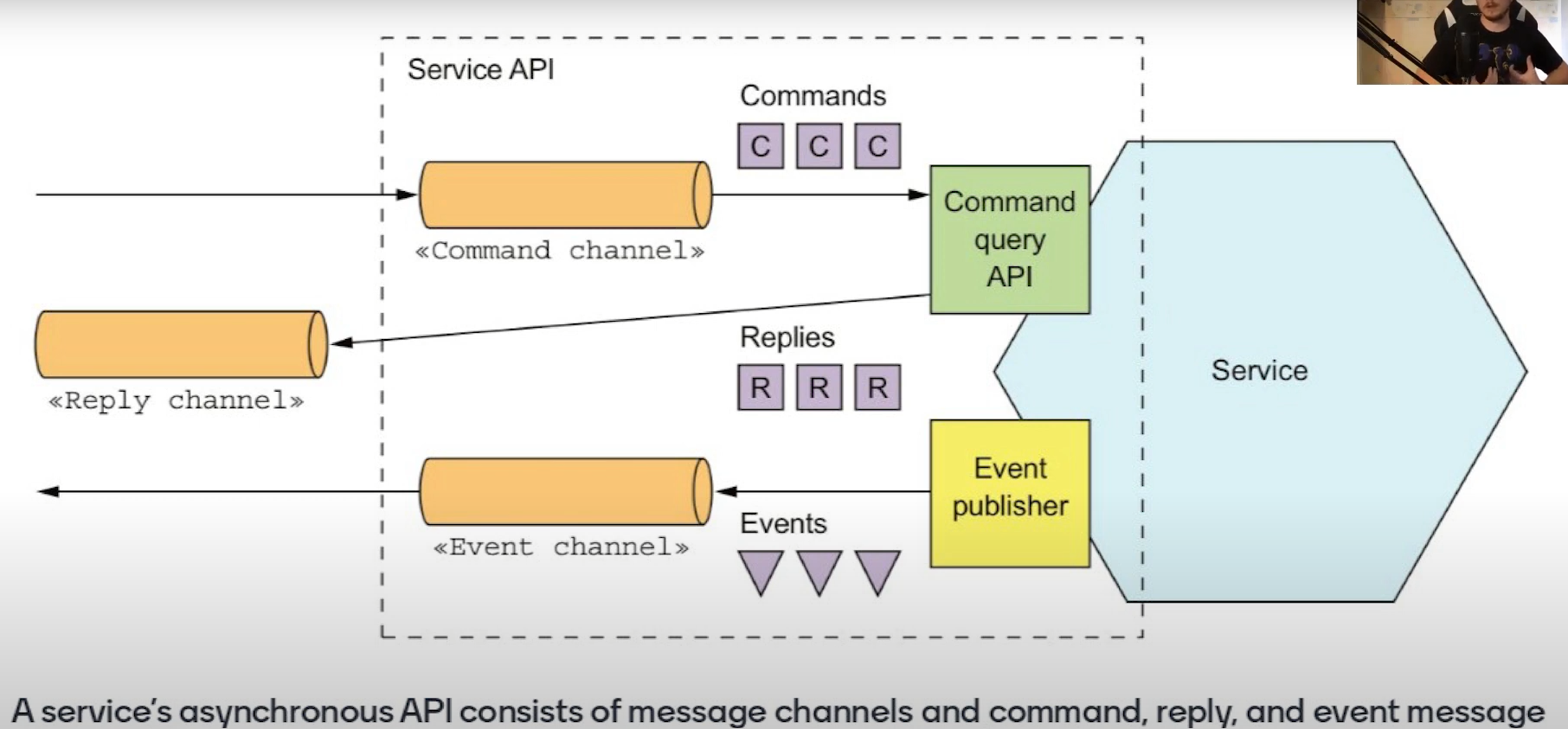
* Point-to-point
* Publish-subscribe (обычно используется для events)

**Асинхронная передача сообщений**



Нужно создать 2 канала. Один для передачи сообщения, а другой для получения сообщений

**Каналы**

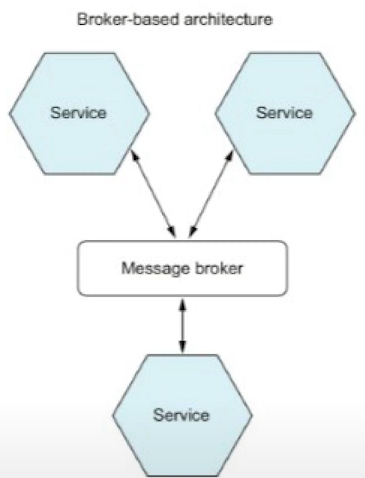
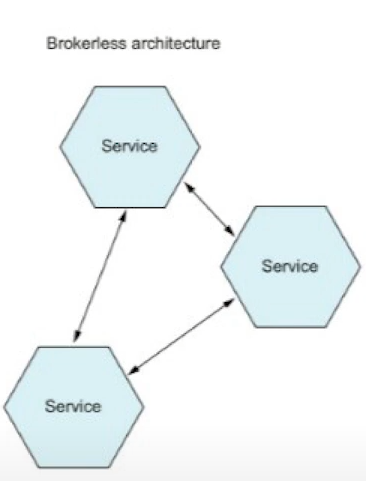


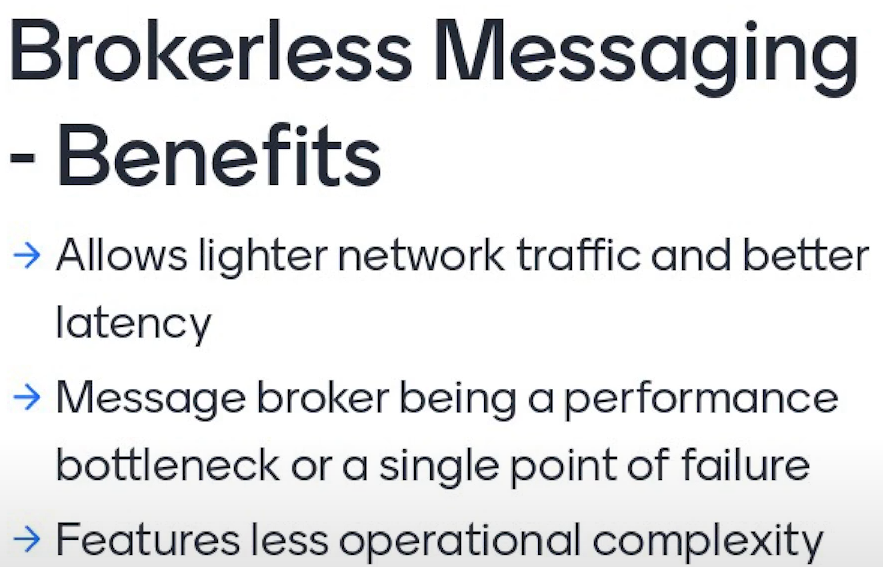
В идеале любой асинхронный сервис должен иметь

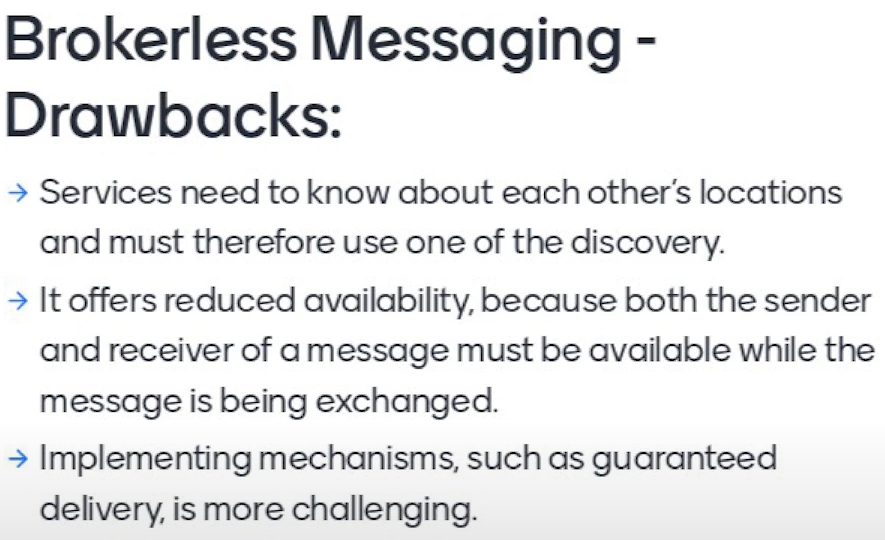
* Канал с его командами (например POST запросы)
* Канал с ответами (документы,)
* Канал с events, которые он транслирует на всех подписчиков, что происходит внутри сервиса

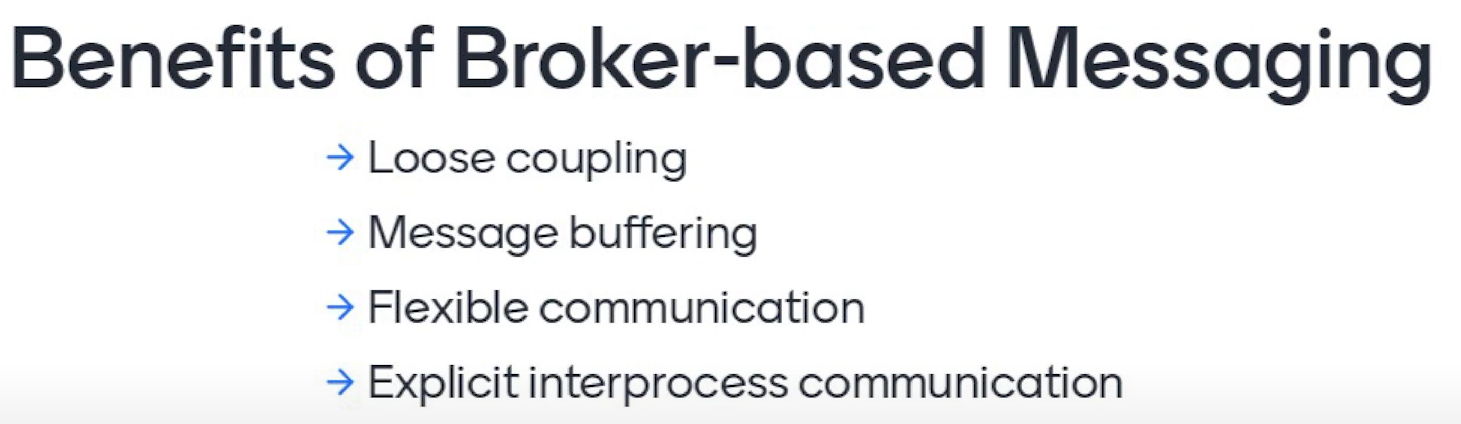
Канал это очередь (RabbitMQ,…) и по необходимости stream(kafka,…)

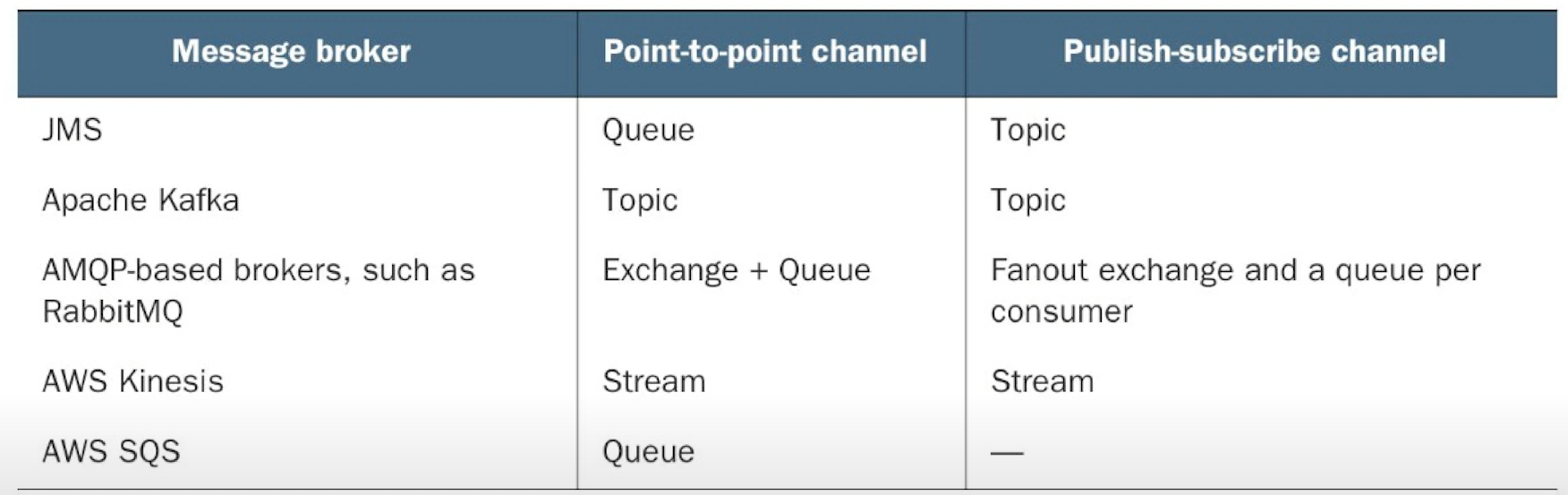
Если в сервисе много тасок, то для каждой из них должен быт свой канал. Так же если у нас несколько асинхронных сервисов, то для каждого из них должны быть свои каналы. Разумно создать отдельный сервис, который бы менеджил все каналы. На помощь приходит **брокер сообщений** (RabbitMQ, Kafka, …)

****









Point-to-point – это как правило очередь

Publish-subscribe – это как правило stream

