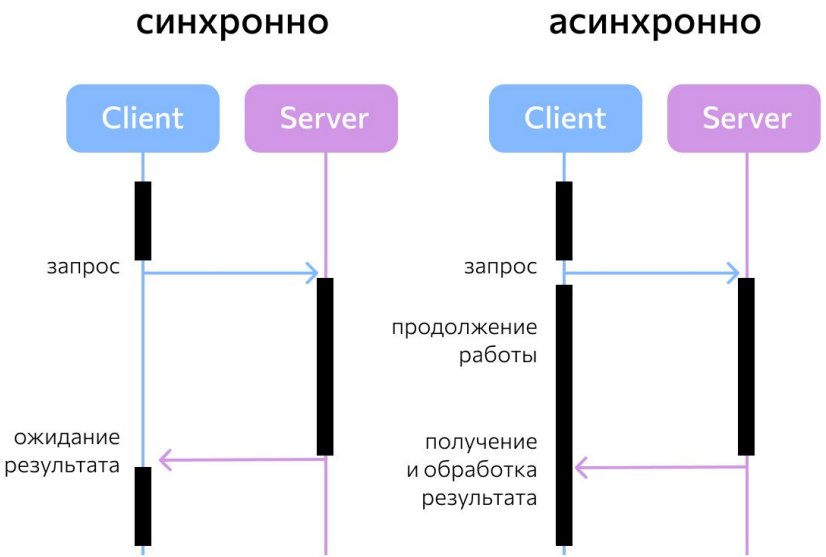
Messaging for Inter-service communication

Контекст

Вы применили шаблон архитектуры микросервисов. Сервисы должны обрабатывать запросы клиентов приложения. Более того, сервисы часто взаимодействуют для обработки этих запросов.

Следовательно, они должны использовать **протокол межсервисного взаимодействия**.



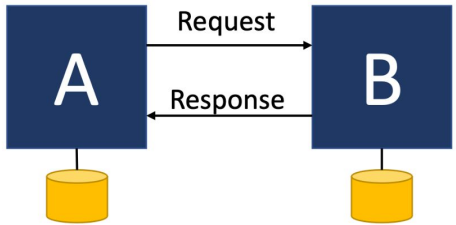
Синхронное взаимодействие (RPC)

Когда выбирать

- Если требуется мгновенный отклик и задержки недопустимы, синхронное взаимодействие может быть предпочтительным.
- Если сервису A нужны данные из сервиса B, но не целесообразно их асинхронно реплицировать из B в A

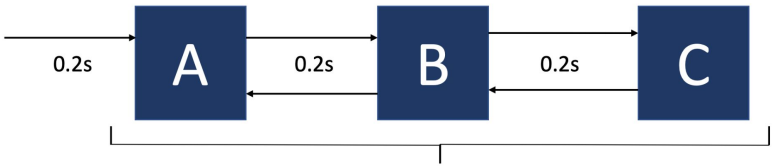
Достоинства

- Простота. Синхронное взаимодействие проще в реализации и отладке.
- Прозрачность. Синхронное взаимодействие позволяет легко отслеживать и управлять последовательностью выполнения операций.



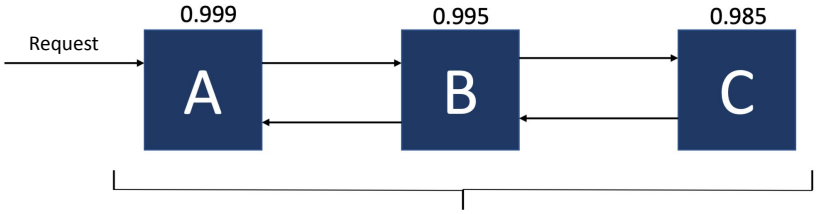
Синхронное взаимодействие (RPC)

Недостатки. Синхронная взаимодействие обеспечивает тесную связь во время выполнения: клиент и сервис должны быть доступны в течение всего времени выполнения запроса.



0.6s

Performance bottleneck



Request SLA = 0.999 * 0.995 * 0.985

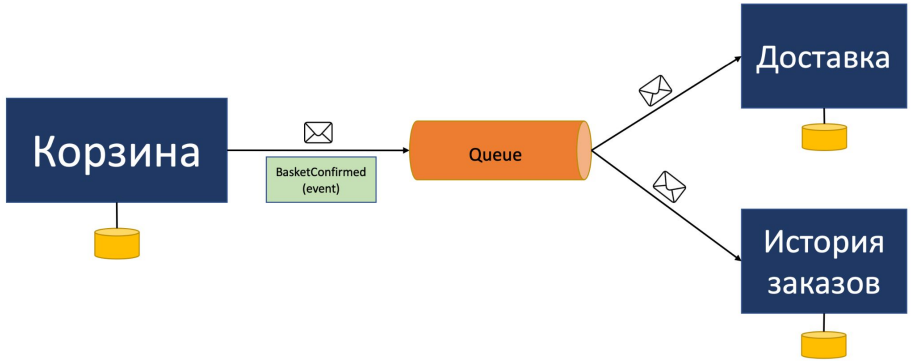
0.979
(97,9%)

Reduced availability

Асинхронное взаимодействие (Messaging)

- Сервисам часто требуется совместная работа
- Длинные запросы (создание отчетов, аналитические запросы)
- НЕ требуется мгновенный отклик и задержки допустимы

=> асинхронное взаимодействие. Сервисы взаимодействуют посредством обмена сообщениями по заданным каналам.



Асинхронное взаимодействие (Messaging)

Достоинства

- *Отказоустойчивость*. Асинхронное взаимодействие позволяет избежать блокировки клиентского микросервиса при недоступности вызываемого микросервиса.
- *Масштабируемость*. Асинхронное взаимодействие может быть параллельным, что способствует лучшей масштабируемости системы.

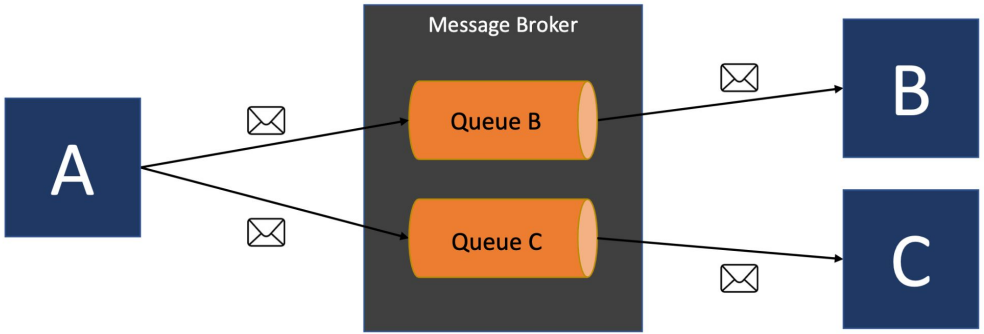
Недостатки

- *Сложность*. Асинхронное взаимодействие требует более сложной реализации, так как необходимо обрабатывать асинхронные ответы и управлять состоянием запросов.
- *Усложнение отладки*. Отслеживаниеи отладкаасинхронноговзаимодействия может быть сложнее из-за распределения запросов и ответов во времени.

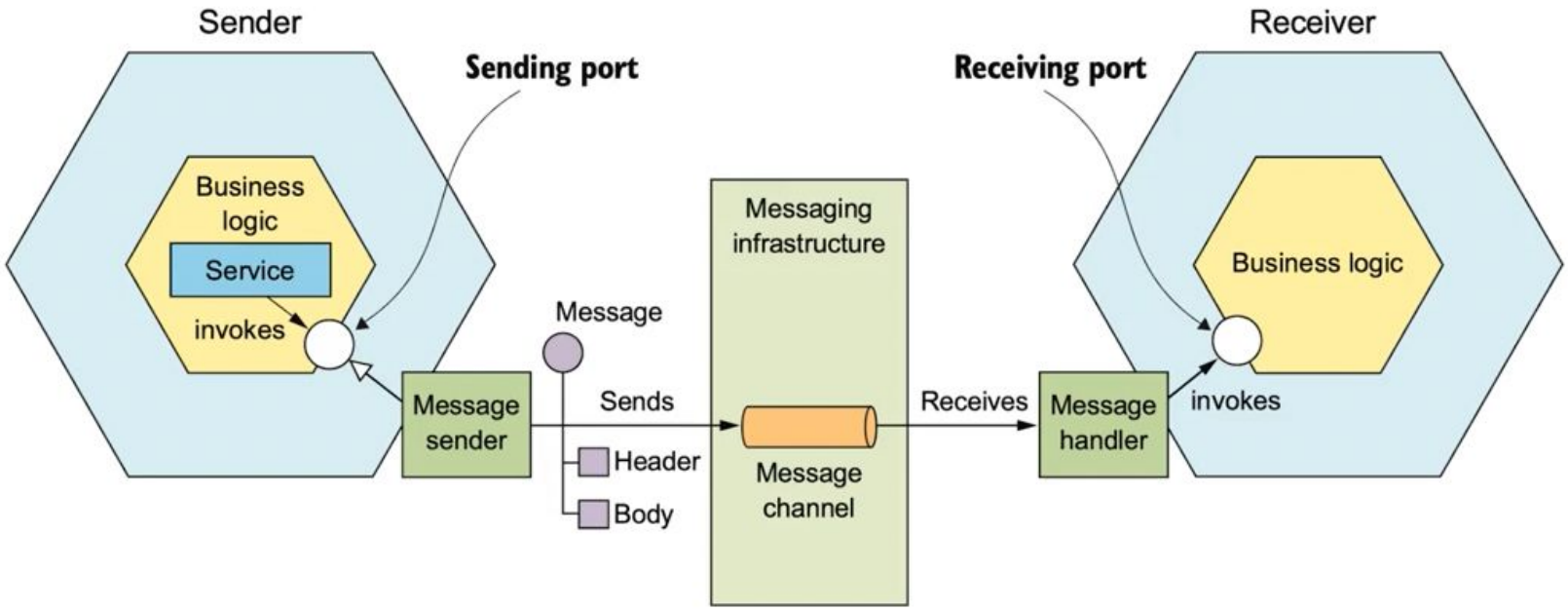
Асинхронное взаимодействие (Messaging)

Как можно организовать такое взаимодействие?

1. Через БД — медленно, неэффективно
2. Через IPC shared memory — сложно в реализации + блокировки
3. **Message Broker**



Message Broker



Message Broker

Плюсы

- Слабая связь во время выполнения, поскольку она отделяет отправителя сообщения от потребителя.
- Повышенная доступность, поскольку брокер сообщений буферизует сообщения до тех пор, пока потребитель не сможет их обработать.
- Поддержка различных шаблонов взаимодействия, включая запрос/ответ, уведомления, запрос/асинхронный ответ, публикация/подписка, публикация/асинхронный ответ и т. д.

Минусы

- Потенциальное узкое место производительности
- Потенциальная единая точка отказа
- Дополнительная сложность в администрировании

AMQP

Advanced Message Queuing Protocol (AMQP) — это открытый стандартный протокол прикладного уровня для промежуточного программного обеспечения, ориентированного на сообщения.

- **Broker** (Server): Приложение, реализующее модель AMQP, которое принимает соединения от клиентов для маршрутизации сообщений, организации очередей и т. д.
- **Message**: Содержимое маршрутизированных данных, включая такую информацию, как payload (тело сообщения) и атрибуты сообщения.
- **Producer**: Приложение, которое помещает сообщения в очередь.
- **Consumer**: Приложение, которое получает сообщения из очередей.

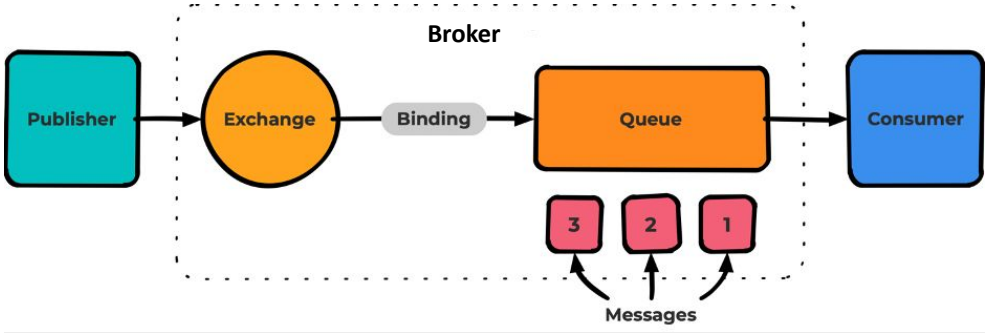
Модель AMQP

Определяет как сообщения принимаются, маршрутизируются, хранятся, ставятся в очередь, а также как работают части приложения, обрабатывающие эти задачи.

Exchange: часть брокера, которая принимает сообщения и направляет их в очереди.

Queue (message queue): Именованная сущность, с которой связаны сообщения и откуда их получают потребители.

Bindings: Правила распределения сообщений из обменов в очереди.



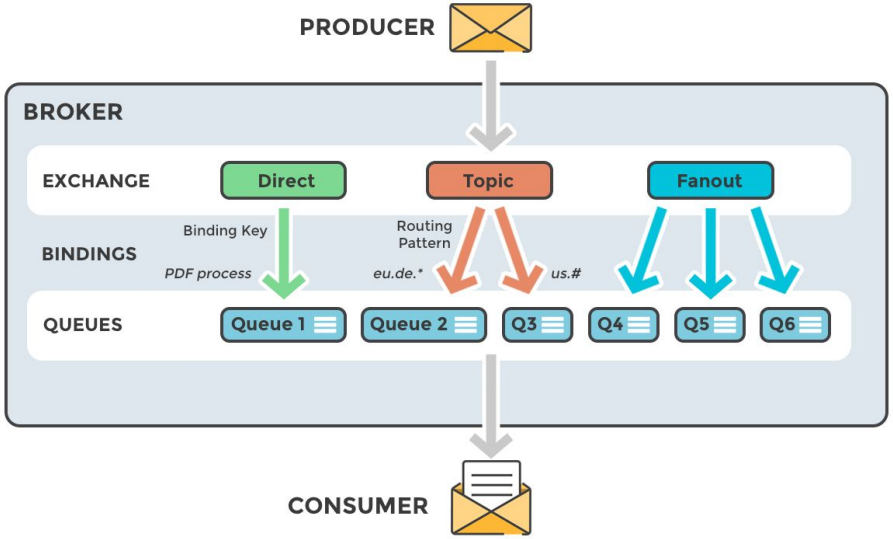
AMQP Exchanges

Получив сообщения от издателей (т.е. клиентов), exchange обрабатывают их и направляют в одну или несколько очередей.

Direct exchange: предполагает доставку сообщений в очереди на основе routing keys.

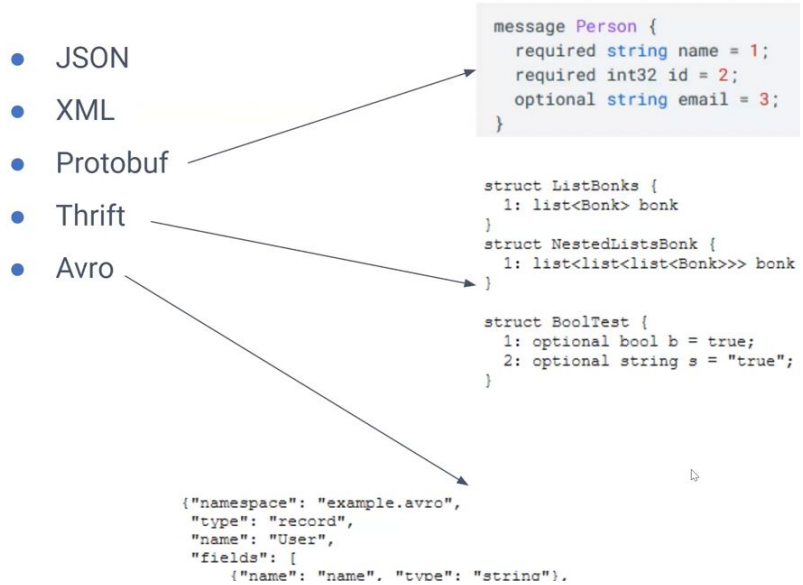
Fanout exchange: отправляет любое сообщение во все очереди, привязанные к нему.

Topic exchange: для сопоставления и отправки сообщений используется шаблон routing key.



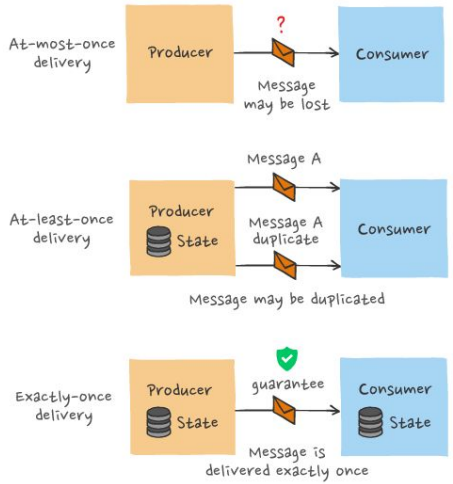
Формат сообщений

Тело сообщения представляет собой набор байтов => формат может быть **любым**.



Требования к Message Broker

1. Гарантии доставки *
   a. at most once
   b. at least once
   c. exactly once
2. Порядок сообщений
3. Ограничения на очереди
4. Сохранность сообщений
5. Кластеризация



* Лучше всего, конечно, иметь «exactly once». Для достижения пунктов 1 и 2 может помочь сам брокер, а для достижения пункта 3 — либо идемпотентные сообщения, либо сохранение полученных сообщений.

RabbitMQ

- Гарантии доставки
  - at most once (по умолчанию)
  - at least once (Persistent + Durable, DLQ, …)
- Порядок сообщений - FIFO
- Возможность сохранения данных на диск
- Подтверждение отправки и получения
- Ограничения на кол-во сообщений в очереди
- Управление для недоставленных сообщений
- Кластеризация (не нативная)

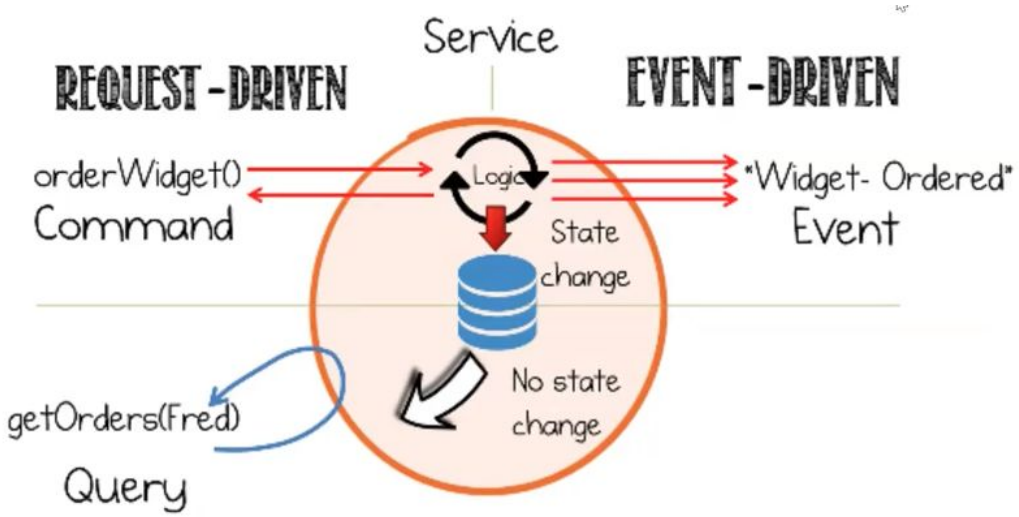
Основные концепции RabbitMQ

- **Producer**: Приложение, отправляющее сообщения.
- **Consumer**: Приложение, получающее сообщения.
- **Queue**: Буфер, хранящий сообщения.
- **Message**: Информация, отправляемая производителем потребителю через RabbitMQ.
- **Exchange**: Получает сообщения от производителей и помещает их в очереди в зависимости от правил, определяемых типом обмена. Для получения сообщений очередь должна быть привязана как минимум к одному обмену.
- **Binding**: Связь между очередью и обменом.
- **Routing key**: Ключ, который Exchange использует для определения способа маршрутизации сообщения в очереди.
- **Connection**: TCP-соединение между вашим приложением и брокером RabbitMQ.
- **Channel**: Виртуальное соединение внутри Connection. Публикация или получение сообщений из очереди происходит по каналу.
- **Vhost, virtual host**: Обеспечивает разделение приложений, использующих один и тот же экземпляр RabbitMQ.
-

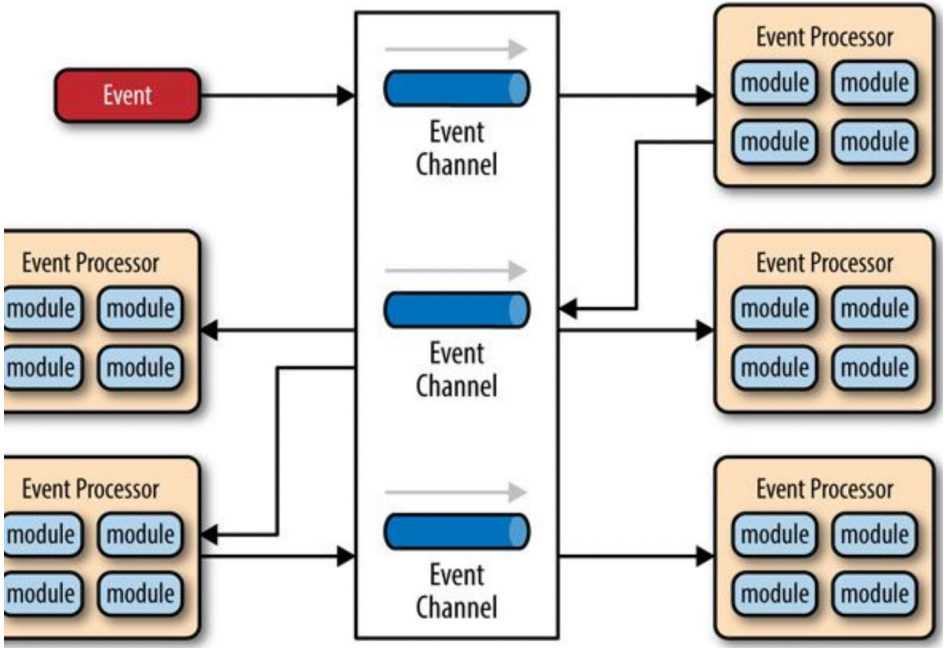
Демонстрация…

Event vs Command

- **Command** это «глагол» — требование что-то сделать (Add Product)
- **Event** - «факт» — сообщение о том, что что-то произошло (ProductAdded)



Event Driven Architecture



Event можно определить как «значительное изменение состояния».

Состоит из: event emitters (or **agents**), event consumers (or **processor**), и event **channels**.

Event Emitter не знает потребителей события, он даже не знает, существует ли потребитель!

Задание: Приложение с RabbitMQ

Оценка: 2

Описание: Создайте простое приложение, которое генерирует и потребляет данные, используя RabbitMQ в качестве брокера сообщений. Разверните приложение с помощью Kubernetes/Docker Compose.

- Producer: Приложение, которое периодически отправляет данные для расчета в RabbitMQ, используя маршрутизацию.
- Consumer: Приложение, которое обрабатывает данные, применяя различные операции, и записывает результат в журнал. Если операция завершилась неудачей, отклоняет сообщение в очередь DLQ.

DoD: GitLab/GitHub репозиторий, README со списком команд для запуска и тестирования + любые дополнительные файлы для работы системы.