

Принципы работы брокеров сообщений.

Погрузимся в детали работы брокеров, начав с рассмотрения моделей взаимодействия. Подчеркнем, что мы не привязываемся к определенному сервису и рассмотрим общие концепты работы брокеров. Начнем с взаимодействия.

Есть два основных шаблона взаимодействия: **издатель-подписчик** (publish-subscribe model) и **очередь сообщений** (message queue, называют еще "точка-точка" или point-to-point).

Шаблон издатель-подписчик (Pub-Sub)	Шаблон очередь сообщений (Message Queue)
Продюсеры отправляют сообщения в определенные темы, а консьюмеры получают сообщения только с теми темами, на которые они подписаны	Очередь сообщений представляет собой временное хранилище, где сообщения от продюсеров хранятся брокером и ожидают обработки
Сообщения могут храниться определенный срок, брокеру может быть не важно, успели его прочитать или нет	Сообщения хранятся на диске или в памяти до тех пор, пока их не прочтут или пока не истечет срок их действия
После получения сообщение обычно удаляется (возможны доп. настройки)	После получения сообщение обычно удаляется (возможны доп. настройки)
Получить одно и тоже сообщение могут разные консьюмеры	Получить одно и тоже сообщение может только один сервис (консьюмер)

Пример брокера, поддерживающего шаблон - Redis, Rabbit MQ	Пример брокера, поддерживающего шаблон – Rabbit MQ
--	---

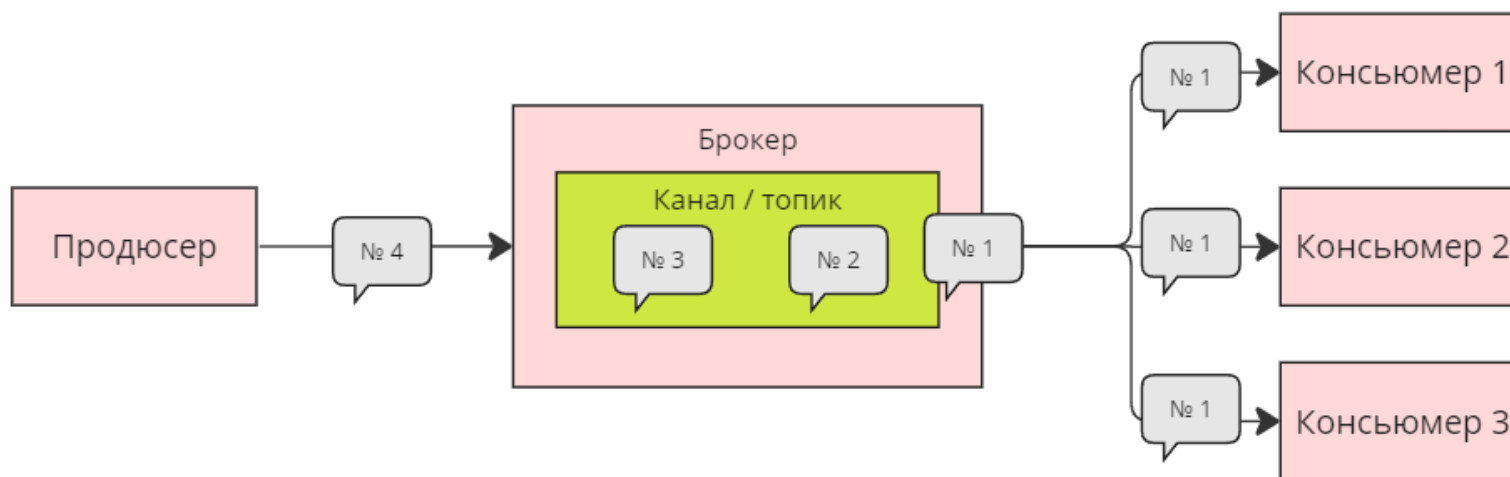
Сразу отметим, что в целом нет ограничений на количество продюсеров. Как один, так и множество продюсеров могут передавать сообщения брокеру сообщений. Концептуальные отличия моделей брокеров, как вы видите, в обработке и передаче сообщений.

Разные брокеры могут использовать разные шаблоны взаимодействия, а некоторые брокеры могут поддерживать и комбинацию шаблонов взаимодействия. Например, Rabbit MQ поддерживает и очередь сообщений, и издатель-подписчик, а Kafka поддерживает не только шаблон издатель-подписчик, но и позволяет читать сообщения без удаления множеству консьюмеров (о шаблоне работы Kafka под названием "журнал" мы поговорим позже). Брокеры могут иметь и расширяемую архитектуру, что позволяет добавлять новые шаблоны самостоятельно, т.е. вы с разработчиками можете придумать и реализовать свой паттерн взаимодействия. Запомним главное, выбор паттерна взаимодействия - важный аспект при проектировании данного стиля.

Подробнее про шаблон издатель-подписчик.

В этом шаблоне publisher (издатель, тоже самое что и продюсер) отправляет сообщение в channel (канал, еще называют темой или топиком), а зарегистрированные на канал subscribers (подписчики, тоже самое что и консьюмеры) получают это сообщение. Продюсеры и консьюмеры никак не связаны друг с другом, т.е. продюсер не беспокоится о том, сколько сервисов обрабатывают его сообщения. Сообщения из канала обрабатываются поочередно, и при обработке копия сообщения с помощью брокера окажется у всех консьюмеров, читающих данный топик.

Как это выглядит, если брокер сам отправляет сообщения (далее мы рассмотрим процесс, когда консьюмер сам запрашивает сообщения):



Когда используется? Модель используется, когда нужно передавать сообщение в несколько сервисов. Допустим, есть сервис регистрации и обработки заданий для сотрудников. Он отправляет в топик "task.event" события, когда у определенного типа задания поменялся статус. И есть сервисы разных бизнес-линий, которые выполняют дополнительную бизнес-логику. Некоторым из них требуется получать информацию, когда задание завершено. В таком случае они подписываются на канал, читают сообщения именно о завершении своих заданий и выполняют какие-либо действия на своей стороне. Например, консьюмер 1 отправляет клиенту СМС-сообщение, что работа по его обращению проведена. А консьюмер 2 создает еще одно задание, т.к. этого требует бизнес-процесс. Таким образом любые бизнес-линии могут легко реализовать свою логику на основе изменения статусов заданий.

Какие плюсы?

- Легко можем реализовать доставку сообщений нескольким потребителям (если выбрать очереди, то мы бы настраивали очередь для каждого консьюмера)
- Снимаем с клиента часть логики доставки запросов. Как вы заметили, сервису заданий не нужно дорабатывать свой продюсер для отправки сообщений консьюмеру 3, это все делает брокер сообщений. Достаточно поддерживать работу продюсера и брокера, чтобы новые потребители могли успешно подключаться
- Снижаем связанность между сервисами - продюсер и консьюмеры независимы друг от друга
- Обеспечиваем буфер для хранения сообщений, при нужных настройках сломанный консьюмер после восстановления работы может заново получить нужные сообщения (если не истек их срок)

- Возможны дополнительные настройки для долгосрочного хранения сообщений

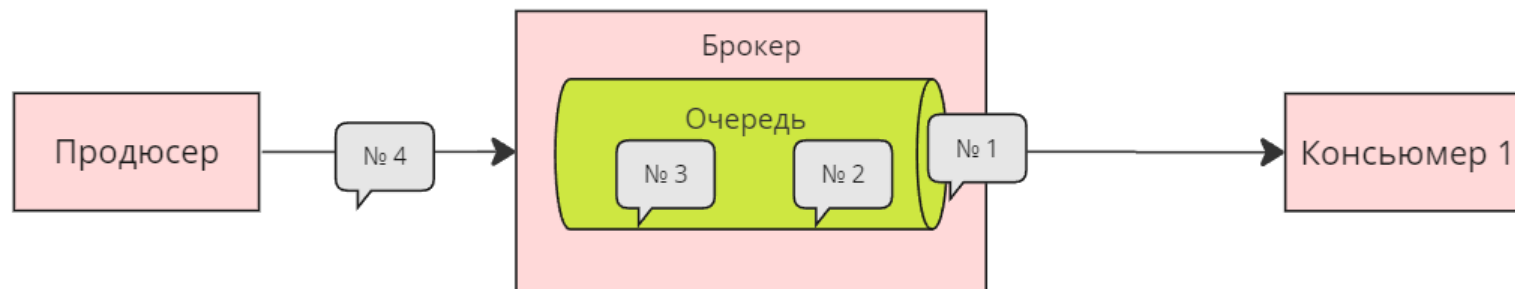
Ранее мы говорили, что сообщение обычно удаляется после прочтения действующими консьюмерами. Но возможны вариации в зависимости от настроек брокера:

1. Сообщение хранится в рамках установленного срока, пока не будет прочитано всеми консьюмерами (какие-то, например, могут быть недоступны). Т.е. отправили сообщение консьюмеру 1, 2, и брокер ожидает, пока консьюмер 3 тоже прочитает сообщение
2. Сообщение отправляется только доступным консьюмерам и удаляется из топика. Т.е. консьюмеры, что были недоступны, никогда не увидят сообщения
3. Иные настройки

Подробнее про шаблон очередь сообщений.

В данном шаблоне продюсер и консьюмер имеют прямую связь через очередь, сообщения доставляются только определенному консьюмеру. Например, консьюмер 2/3/4 никогда не сможет прочитать сообщения из очереди, если они прочитаны консьюмером 1. В данной модели сообщения также обрабатываются по порядку.

Как это выглядит, если сам брокер отправляет сообщения:



Когда используется? Модель используется, когда нужно передавать сообщение только одному сервису. Представьте, что у вас есть веб-приложение для заказа продуктов. Когда клиент размещает заказ, сервис как продюсер создает сообщение с деталями заказа и помещает его в очередь сообщений брокера. Затем, другой компонент приложения, например, сервис обработки заказов, подписан на эту очередь и

получает сообщения по мере их поступления. Таким образом, очередь сообщений позволяет отделить компоненты приложения, уменьшая связность. Также обеспечивается доставка сообщений, даже если один из компонентов временно недоступен, так как сообщения сохраняются в очереди до тех пор, пока они не будут успешно обработаны.

Какие плюсы?

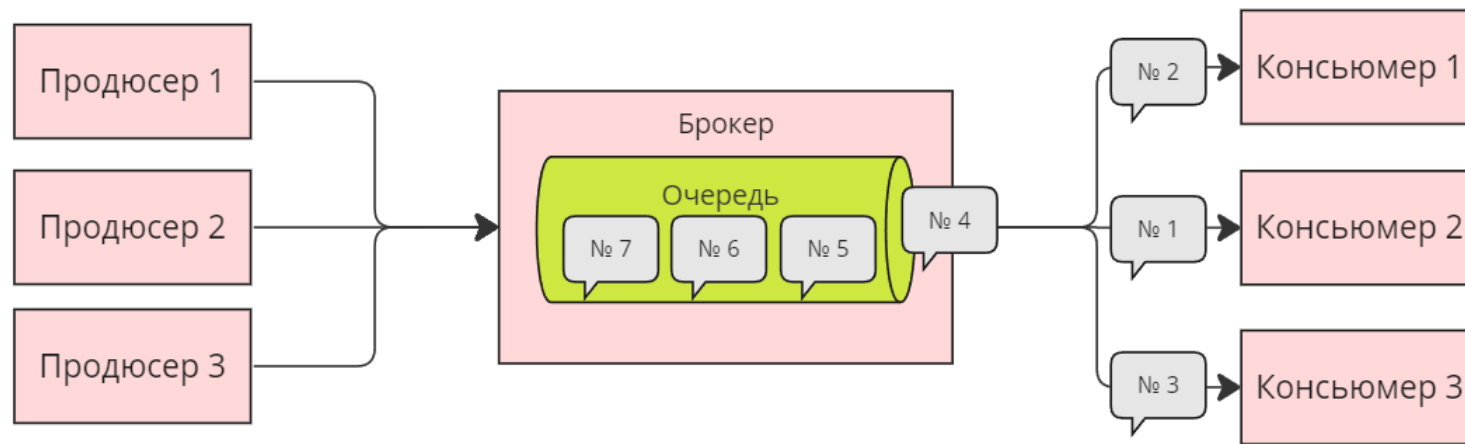
- Легко можем реализовать доставку сообщений одному потребителю
- Снимаем с клиента часть логики доставки запросов. Продюсер просто отправляет сообщения в брокер, достаточно отслеживать "трафик" на нужный консьюмер, проще говоря, опять же поддерживать работу продюсера и брокера
- Снижаем связанность между сервисами - продюсер и консьюмер независимы друг от друга
- Обеспечиваем буфер для хранения сообщений, сломанный консьюмер после восстановления работы может заново получить нужные сообщения

В данном случае есть еще один из главных плюсов данной модели и мы его рассмотрим, введя новое понятие.

Инстанс (иногда называют нода / node) - это запущенный экземпляр программы / сервиса на сервере.

Давайте представим, что у нас произошло большое масштабирование, количество заказов и соответственно сообщений, сильно выросло, наше веб-приложение для заказа продуктов должно справляться с этой нагрузкой. В нашем случае мы запускаем дополнительные инстансы одного и того же сервиса обработки заказов, чтобы они вместе разбирали очередь сообщений. Таким образом получаем несколько консьюмеров (продюсеров мы тоже увеличили).

Получится такая схема взаимодействия:



Мы назвали продюсеры и консьюмеры 1/2/3, на самом деле это просто повторные экземпляры сервисов для увеличения мощностей (называется горизонтальное масштабирование). Можно было назвать компоненты как Консьюмер 1, Консьюмер 1', Консьюмер 1" и т.д.

В данной схеме продюсеры пишут много сообщений в очередь и консьюмеры обрабатывают каждое сообщение отдельно, не пересекаясь друг с другом. Консьюмер 1 обрабатывает сообщение №2 (допустим, он был недоступен при отправке сообщения №1), консьюмер 2 обрабатывает сообщение №1, консьюмер 3 обрабатывает сообщение №3. Если возвращаться к логике обработки заказов, сервис не создаст лишние экземпляры заказов и в итоге справится с нужной нагрузкой. Но представьте, что будет, если мы используем модель pub-sub?

Верно. Каждый консьюмер прочитает каждое сообщение, получив его копию и заказы начнут дублироваться. Исправить это можно будет только доработкой консьюмера, например, разработать дедупликацию сообщений. Таким образом мы подчеркнули явное отличие модели очереди и выделили один из главных плюсов - сообщение может быть прочитано только одним консьюмером, один раз.

Чтобы выбрать паттерн взаимодействия, можете для начала ориентироваться на 3 вопроса:

- 1. Сообщения должны читать несколько потребителей или один?
 - 1. Несколько (pub-sub)
 - 2. Один (очереди)
- 2. Нужно хранить сообщения после отправки/прочтения консьюмером?

1. Да (pub-sub)

2. Нет (очереди)

3. Нужно ли экономить ресурсы (память)?

1. Да (очереди)

2. Нет (pub-sub)

Помните, нет парадигмы «что лучше» и «что надежнее», важно отталкиваться от контекста задачи и принимать решение исходя из количества плюсов выбранного решения. Очередь подойдет там, где нужно гарантировать обработку сообщения 1 раз. Подписка отлично подходит для масштабирования или для реализации источника данных, от которого многие сервисы должны получать сообщения. Сейчас все чаще встречаются модели pub-sub и журнал (Kafka), а очередь используется реже.

Давайте рассмотрим пример, когда мы можем реализовать оба паттерна в своем приложении.

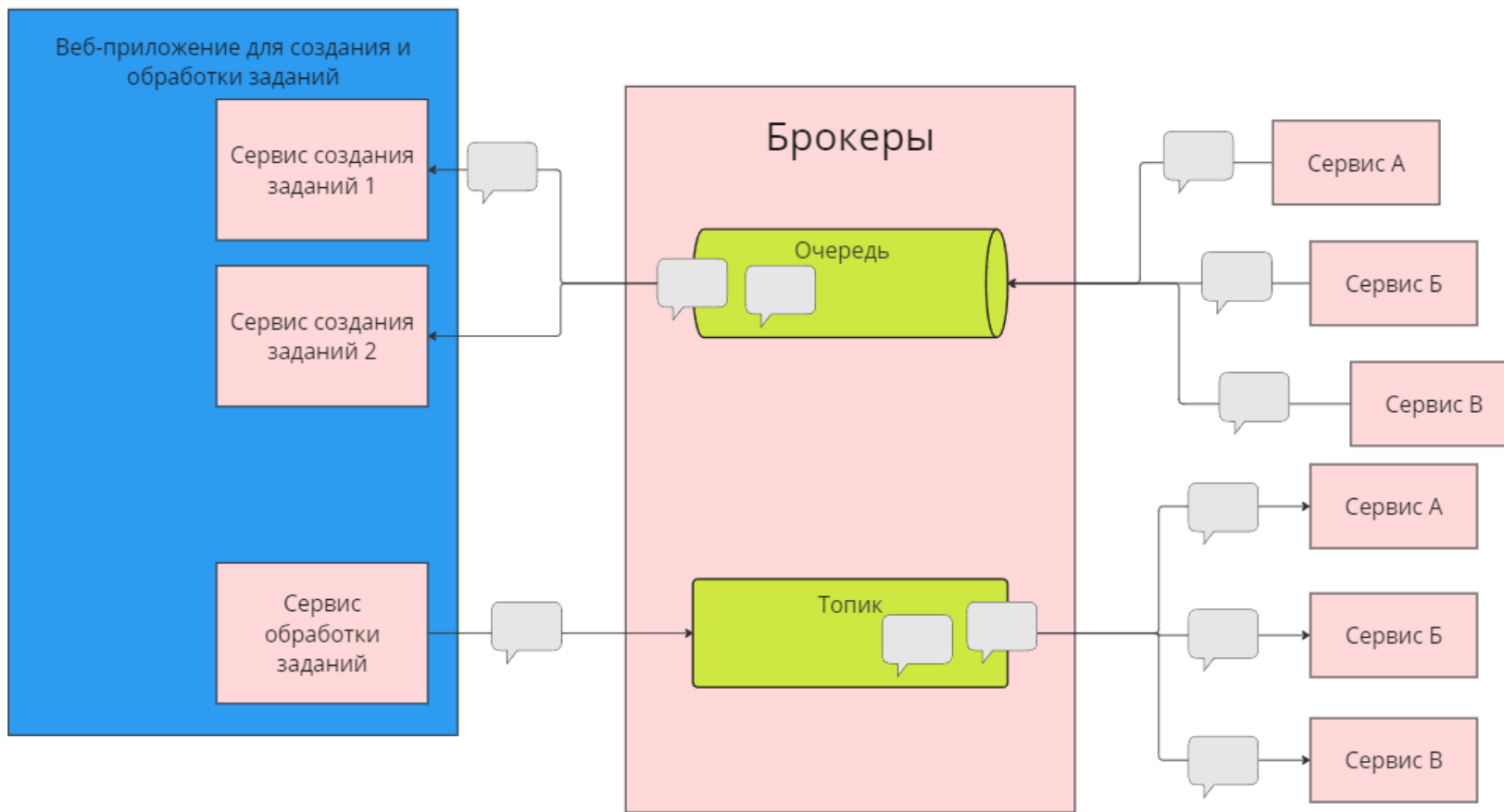
Дано:

- веб-приложение для создания и обработки заданий
 - есть сервис создания заданий, в двух инстансах
 - и есть сервис обработки заданий, в одном инстансе
- сервисы разных бизнес-линий, которые хотят создавать задания асинхронно
- сервисы разных бизнес-линий, которые хотят узнавать когда задание завершено

Нужно:

Спроектировать "кусоч" системы с помощью брокеров, показав как разные сервисы могут создавать задания и следить за их статусами.

Посмотрим, как можно реализовать схему взаимодействия:



В данном случае мы реализовали создание через очередь, когда много продюсеров могут отправлять сообщения в сервис создания заданий. И реализовали отправку сообщений по итогу обработки заданий (отложили, завершили или другое) в топик, который может начать "слушать" любой сервис, выполняя свою бизнес-логику после определенных сообщений. Это не единственное и не наилучшее решение данной задачи, а пример применения разных паттернов в одном проекте. Чтобы вы поняли - ограничения лишь в силах ваших разработчиков и в ресурсах, запустить можно разные схемы взаимодействия, сразу на одном проекте.