Сообщения, наряду с блоками вычисления и хранения, составляют три основных блока почти в каждой блок-схеме системы. Очереди сообщений, по существу, являются связующим звеном между различными процессами в ваших приложениях и обеспечивают надежный и масштабируемый интерфейс взаимодействия с другими подключенными системами и устройствами.

**Очередь** — структура данных с дисциплиной доступа к элементам «первый пришёл — первый вышел». Добавление элемента возможно лишь в конец очереди, выборка — только из начала очереди, при этом выбранный элемент из очереди удаляется.

**Зачем нужны очереди сообщений в микросервисной архитектуре:**

При проектировании микросервисов часто возникает вопрос: какой способ связи между ними выбрать.

Многие отдают предпочтение RESTful API. Однако этот подход не всегда эффективен, так как в отдельных случаях чреват долгим ожиданием на клиентской стороне и потерей информации в случае сбоев.

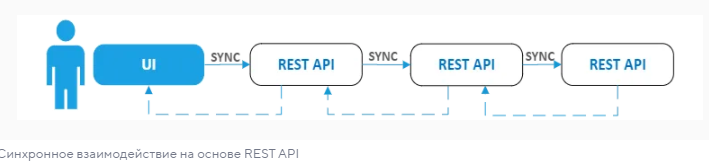
**Sync vs Async: синхронное и асинхронное взаимодействие**

**Очереди сообщений (Message Queue)** — это форма асинхронной коммуникации между сервисами.

Предположим, вы разрабатываете сайт книжного магазина и у вас есть сервис, к которому обращается пользователь, например отправка рецензии на прочитанную книгу. При нажатии кнопки «Отправить» вызывается некоторый API, который, в свою очередь, может обратиться к другим API.

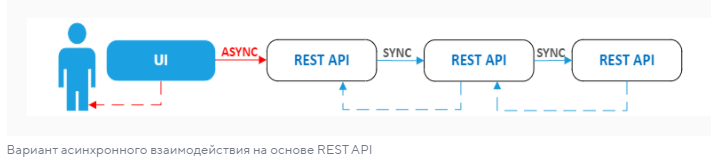
При **синхронном взаимодействии** все запросы в этой цепочке вызовов выполняются строго друг за другом, а при выполнении последнего запроса ответы последовательно передаются в обратном направлении. В итоге пользователь вынужден пару секунд ждать сообщения о публикации своего отзыва, хотя его не интересуют особенности серверной обработки и он вполне обоснованно хочет увидеть сообщение сразу после нажатия кнопки. Конечно, время ожидания будет во многом определяться мощностью оборудования, но при пиковых нагрузках оно может стать серьезной проблемой.

Еще один недостаток такой схемы — обработка сбоев. Если на одном из шагов возникнет исключение, оно каскадно возвратится назад, и пользователь получит уведомление об ошибке с просьбой повторно отправить рецензию. Вряд ли кого-то обрадует получение подобного сообщения после длительного ожидания.



Описанную схему можно изменить, добавив асинхронные вызовы. Достаточно вызвать в асинхронном режиме первый REST API и параллельно вернуть пользователю сообщение о том, что его рецензия принята и будет размещена, например, в течение суток. В итоге сайт не блокируется, а вызовы всех последующих API происходят независимо от пользователя.

Но у такой схемы также есть существенный недостаток: в случае сбоя в одном из API информация, введенная пользователем, будет потеряна. Если в первом примере в случае ошибок достаточно повторно отправить рецензию, то здесь ее необходимо заполнить заново.

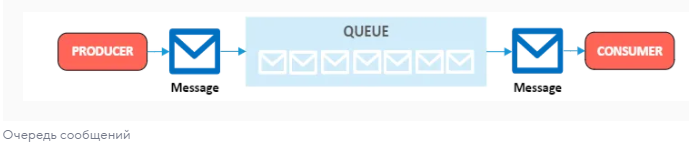


Для устранения недостатков обеих схем как раз и предназначены очереди сообщений.

**Принципы работы очередей сообщений**

Очереди предоставляют буфер для временного хранения сообщений и конечные точки, которые позволяют подключаться к очереди для отправки и получения сообщений в **асинхронном режиме**.

В сообщениях могут содержаться запросы, ответы, ошибки и иные данные, передаваемые между программными компонентами. Компонент, называемый производителем (Producer), добавляет сообщение в очередь, где оно будет храниться, пока другой компонент, называемый потребителем (Consumer), не извлечет сообщение и не выполнит с ним необходимую операцию.



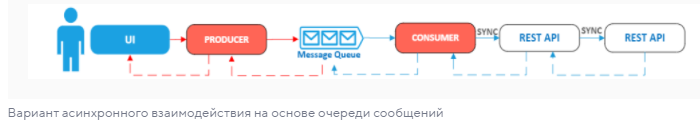
Очереди поддерживают получение сообщений как методом Push, так и методом Pull:

* **метод Pull** подразумевает периодический опрос очереди получателем по поводу наличия новых сообщений;
* **метод Push** — отправку уведомления получателю в момент прихода сообщения. Второй метод реализует модель «Издатель/Подписчик» (Publisher/Subscriber).

Так как очереди могут использоваться несколькими производителями и потребителями одновременно, обычно их реализуют с помощью дополнительной системы, называемой брокером. **Брокер сообщений (Message Broker)** занимается сбором и маршрутизацией сообщений на основе предопределенной логики. Сообщения могут передаваться с некоторым ключом — по этому ключу брокер понимает, в какую из очередей (одну или несколько) должно попасть сообщение.

Вернемся к примеру с отправкой рецензии. Пусть та часть сервиса, к которому обращается пользователь, выступит в качестве производителя и будет направлять запросы на создание рецензий в очередь. Сразу после добавления сообщения в очередь пользователю можно направлять уведомление об успехе операции. Вся последующая логика обработки будет выполняться независимо от него на стороне потребителя, подписанного на очередь.

Завершив обработку, потребитель отправит подтверждение в очередь, после чего исходное сообщение будет удалено. Но если во время обработки произойдет сбой и подтверждение не будет получено вовремя, сообщение может быть повторно извлечено потребителем из очереди.



Таким образом, использование очередей сообщений решает сразу две задачи: сокращает время ожидания пользователя за счет асинхронной обработки и предотвращает потерю информации при сбоях. Но не следует рассматривать очереди как универсальное средство для любого вида приложений: как и у любого инструмента, у них есть свои преимущества и недостатки

**Польза и преимущества очередей сообщений в микросервисной архитектуре**

Используя очереди сообщений в качестве основного средства взаимодействия микросервисов (Microservices Communication), можно добиться следующих преимуществ:

* **Отделение логически независимых компонентов друг от друга** **(Decoupling)**

Отличительная черта микросервисов — их автономность. И очереди во многом помогают уменьшить зависимости между ними. Каждое сообщение, передаваемое в очереди, — это всего лишь массив байтов с некоторыми метаданными. Метаданные нужны для направления в конкретную очередь, а информация, содержащаяся в основной части (теле) сообщения, может быть практически любой. Брокер не анализирует данные, он выступает лишь в качестве маршрутизатора. Это позволяет настроить взаимодействие между компонентами, работающими даже на разных языках и платформах.

* **Улучшение масштабируемости**

Очереди сообщений упрощают независимое масштабирование микросервисов. Наблюдая за состоянием очередей, можно масштабировать те сервисы, на которые приходится большая часть нагрузки. Кроме этого, очереди легко позволяют не только увеличивать число экземпляров существующих сервисов, но и добавлять новые с минимальным временем простоя. Все, что для этого требуется, — добавить нового потребителя, прослушивающего события в очереди.

Однако сами очереди также необходимо масштабировать, и это может создать дополнительные сложности.

* **Балансировка нагрузки**

Если один из сервисов не справляется с нагрузкой, требуется возможность запускать больше его экземпляров быстро и без дополнительных настроек. Обычно для этих целей используют балансировщик нагрузки, интегрированный с сервером обнаружения служб и предназначенный для распределения трафика. При использовании очередей сообщений сам брокер по умолчанию является балансировщиком нагрузки. Если несколько потребителей слушают очередь одновременно, сообщения будут распределяться между ними в соответствии с настроенной стратегией.

* **Повышение надежности**

Выход из строя одного из компонентов не сказывается на работе всей системы: при восстановлении он обработает сообщение, находящееся в очереди. Ваш веб-сайт по-прежнему может работать, даже если задерживается часть обработки заказа, например, из-за проблем с сервером БД или системой электронной почты.

Правда, при этом очередь сама приобретает статус SPoF (Single Point Of Failure), поэтому необходимо заранее предусмотреть действия на случай ее аварийного отключения.

* **Безопасность**

Большинство брокеров выполняют аутентификацию приложений, которые пытаются получить доступ к очереди, и позволяют использовать шифрование сообщений как при их передаче по сети, так и при хранении в самой очереди. Таким образом, очередь снимает с ваших сервисов бремя организации авторизации запросов.

**Варианты использования очередей сообщений**

Очереди сообщений полезны в тех случаях, где возможна асинхронная обработка. Рассмотрим наиболее частые сценарии использования очередей сообщений (Message Queue use Cases):

* **Фоновая обработка долгосрочных задач на веб-сайтах**

Сюда можно отнести задачи, которые не связаны напрямую с основным действием пользователя сайта и могут быть выполнены в фоновом режиме без необходимости ожидания с его стороны. Это обработка изображений, преобразование видео в различные форматы, создание отзывов, индексирование в поисковых системах после изменения данных, отправка электронной почты, формирование файлов и так далее.

* **Буферизация при пакетной обработке данных**

Очереди можно использовать в качестве буфера для некоторой массовой обработки, например пакетной вставки данных в БД или HDFS. Очевидно, что гораздо эффективнее добавлять сто записей за раз, чем по одной сто раз, так как сокращаются накладные расходы на инициализацию и завершение каждой операции. Но для стандартной архитектуры может стать проблемой генерация данных клиентской службой быстрее, чем их может обработать получатель. Очередь же предоставляет временное хранилище для пакетов с данными, где они будут храниться до завершения обработки принимающей стороной.

* **Отложенные задачи**

Многие системы очередей позволяют производителю указать, что доставка сообщений должна быть отложена. Это может быть полезно при реализации льготных периодов. Например, вы разрешаете покупателю отказаться от размещения заказа в течение определенного времени и ставите отложенное задание в очередь. Если покупатель отменит операцию в указанный срок, сообщение можно удалить из очереди.

* **Сглаживание пиковых нагрузок**

Помещая данные в очередь, вы можете быть уверены, что данные будут сохранены и в конечном итоге обработаны, даже если это займет немного больше времени, чем обычно, из-за большого скачка трафика. Увеличить скорость обработки в таких случаях также возможно — за счет масштабирования нужных обработчиков.

* **Гарантированная доставка при нестабильной инфраструктуре**

Нестабильная сеть в сочетании с очередью сообщений создает надежный системный ландшафт: каждое сообщение будет отправлено, как только это будет технически возможно

* **Упорядочение транзакций**

Многие брокеры поддерживают очереди FIFO, полезные в системах, где важно сохранить порядок транзакций. Если 1000 человек размещают заказ на вашем веб-сайте одновременно, это может создать некоторые проблемы с параллелизмом и не будет гарантировать, что первый заказ будет выполнен первым. С помощью очереди можно определить порядок их обработки.

* **Сбор аналитической информации**

Очереди часто применяют для сбора некоторой статистики, например использования определенной системы и ее функций. Как правило, моментальная обработка такой информации не требуется. Когда сообщения поступают в веб-службу, они помещаются в очередь, а затем при помощи дополнительных серверов приложений обрабатываются и отправляются в базу данных.

* **Разбиение трудоемких задач на множество маленьких частей**

Если у вас есть некоторая задача для группы серверов, то вам необходимо выполнить ее на каждом сервере. Например, при редактировании шаблона мониторинга потребуется обновить мониторы на каждом сервере, использующем этот шаблон. Вы можете поставить сообщение в очередь для каждого сервера и выполнять их одновременно в виде небольших операций.

* **Прочие сценарии, требующие гарантированной доставки информации и высокого уровня отказоустойчивости**

Это обработка финансовых транзакций, бронирование авиабилетов, обновление записей о пациентах в сфере здравоохранения и так далее.

**Сложности использования и недостатки очередей сообщений: как с ними справляться**

Несмотря на многочисленные преимущества очередей сообщений, самостоятельное их внедрение может оказаться довольно сложной задачей по нескольким причинам:

* По сути, это еще одна система, которую необходимо купить/установить, правильно сконфигурировать и поддерживать. Также потребуются дополнительные мощности.
* Если брокер когда-либо выйдет из строя, это может остановить работу многих систем, взаимодействующих с ним. Как минимум необходимо позаботиться о резервном копировании данных.
* С ростом числа очередей усложняется и отладка. При синхронной обработке сразу очевидно, какой запрос вызвал сбой, например, благодаря иерархии вызовов в IDE. В очередях потребуется позаботиться о системе трассировки, чтобы быстро связать несколько этапов обработки одного запроса для обнаружения причины ошибки.
* При использовании очередей вы неизбежно столкнетесь с выбором стратегии доставки сообщений. В идеале сообщения должны обрабатываться каждым потребителем однократно. Но на практике это сложно реализовать из-за несовершенства сетей и прочей инфраструктуры. Большинство брокеров поддерживают две стратегии: доставка хотя бы раз (At-least-once) или максимум раз (At-most-once). Первая может привести к дубликатам, вторая — к потере сообщений. Обе требуют тщательного мониторинга. Некоторые брокеры также гарантируют строго однократную доставку (Exactly-once) с использованием порядковых номеров пакетов данных, но даже в этом случае требуется дополнительная проверка на стороне получателя.

Хорошая новость в том, что многие облачные провайдеры сейчас предлагают очереди как сервис (MQ as a Service). Поэтому если у вас недостаточно ресурсов для самостоятельной настройки и поддержки очередей сообщений, то можно воспользоваться одним из готовых решений. Большинство из них включает автоматизацию настройки, масштабирование, диагностику ошибок и техническую поддержку, а также поддерживает строго однократную доставку в очередях FIFO.

**В каких случаях очереди неэффективны**

Конечно, очереди не являются универсальным средством для любых приложений. Рассмотрим варианты, когда очереди не будут самым эффективным решением:

* **У вашего приложения простая архитектура и функции, и вы не ожидаете его роста**. Важно понимать, что очереди сообщений — это дополнительная сложность. Эту систему также необходимо настраивать, поддерживать, осуществлять мониторинг ее работы и так далее. Да, можно использовать Managed-решение, но вряд ли это будет оправдано для небольших приложений. Добавление очередей должно упрощать архитектуру, а не усложнять ее.
* **Вы используете монолитное программное обеспечение, в котором развязка (Decoupling) невозможна или не приоритетна**. Если вы не планируете разбивать монолит на микросервисы, но вам требуется асинхронность — для ее реализации обычно достаточно стандартной многопоточной модели. Очереди могут оказаться избыточным решением до тех пор, пока не возникнет явная необходимость в разделении приложения на автономные компоненты, способные независимо выполнять задачи.

Выбирая инструмент для будущего приложения, обязательно взвесьте все за и против. Не стоит использовать очереди сообщений для задач, которые могут быть решены другим, более простым в настройке и обслуживании способом. Но в тех случаях, когда запланирован переход на микросервисы и бизнес-логика допускает возможность асинхронной обработки, очереди сообщений могут стать лучшим выбором для повышения производительности и надежности вашего продукта.

**Что такое RabbitMQ**

RabbitMQ — это распределенный и горизонтально масштабируемый брокер сообщений. Он позволяет разным программам взаимодействовать с помощью протокола AMQP, а через дополнительные модули — и с помощью некоторых других протоколов: MQTT, HTTP и так далее.

Устройство брокера упрощенно можно описать так:

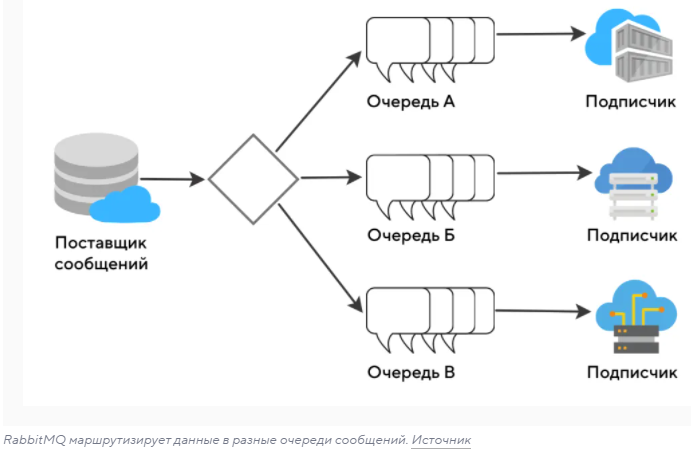
* есть продюсер, или поставщик сообщений, отправляющий события;
* очередь сообщений — своего рода «‎почтовый ящик», где хранятся сообщения;
* подписчики, то есть программы — получатели сообщений.

В очереди может храниться любое количество сообщений от неограниченного количества поставщиков, а получать их может неограниченное число подписчиков.

RabbitMQ принимает сообщения от поставщиков и отправляет им подтверждение о приеме, а затем перенаправляет их подписчикам. Получатели подтверждают, что сообщение доставлено, либо сигнализируют о неудаче. Во втором случае сообщение остается в очереди, пока не будет доставлено. А после доставки оно удалится из системы

Основная фишка RabbitMQ — это гибкая маршрутизация сообщений между различными поставщиками и потребителями событий. Решение не ограничивается созданием простой очереди данных между двумя сторонами. В сервере реализована концепция принимающих события узлов (эксчейнджей) — они маршрутизируют данные в разные очереди сообщений RabbitMQ.

Например, одно и то же сообщение должны получить три подписчика. Оно попадет на узел, который отправит три одинаковых сообщения в три очереди для всех подписчиков, которым оно должно быть доставлено.



**Особенности RabbitMQ:**

* Механизм маршрутизации дополняют ключи маршрутизации (Routing Keys). Они позволяют создавать гибкие правила пересылки сообщений между источниками и потребителями. Благодаря этому RabbitMQ подходит для широкого спектра задач по реализации нетривиальных бизнес-процессов в коде.
* Обширные инструменты маршрутизации позволяют разработчикам и администраторам настраивать сложные системы с тысячами источников и приемников сообщений. Решение подходит для систем бронирования билетов, логистических программ и микросервисных приложений.
* RabbitMQ реализует концепцию push-доставки: поставщик может направить новые события в сеть, но получатель не может их запросить у поставщика. При этом система не гарантирует порядок доставки сообщений.

**Kafka**

Apache Kafka — это тоже распределенный брокер сообщений, но работает он по другому принципу. Он реализует архитектурную концепцию распределенного лога, в который заносят информацию разные поставщики данных. Затем информацию из этого лога читают другие программы и микросервисы.

Сообщения сохраняются в брокере, подписчики подписываются на них, при этом гарантируется, что сообщения находятся в той же последовательности, в которой поступили.

Kafka не отслеживает, получил ли подписчик сообщение, он просто хранит его в течение установленного времени. Подписчики сами отправляют запросы к брокеру о новых сообщениях и указывают, что им нужно прочитать. Сообщения после прочтения не удаляются, поэтому есть возможность повторного их получения.



**Особенности Apache Kafka:**

* Apache Kafka славится способностью поглощать и пересылать титанические объемы данных. В нем есть все, что нужно для работы с высокими нагрузками: репликация, горизонтальное масштабирование, параллельная обработка потоков сообщений сразу на нескольких серверах.
* Параллелизм и распределенная архитектура хорошо сказываются на надежности: даже выход из строя части кластера не нарушит доставку сообщений.
* В пересылке данных принимают участие поставщики и потребители данных. Поставщики пишут сообщения в Apache Kafka, потребители их читают. Для организации систем со сложной топологией применяют темы — своего рода разделы для категоризации различных сообщений по назначению.
* Для маршрутизации сообщений могут применяться Routing Keys, похожие на те, что используются в RabbitMQ. Но, в отличие от RabbitMQ, Apache Kafka гарантирует порядок доставки сообщений.

**RabbitMQ vs Apache Kafka: что выбрать?**

**RabbitMQ**

* Вам важна гибкость в маршрутизации сообщений внутри системы. Инструментарий для построения путей доставки данных в RabbitMQ способен решить даже самые хитрые сценарии в организации потоков событий.
* Вам важен сам факт доставки сообщений, но порядок доставки не принципиален. Настройки RabbitMQ его не гарантируют: если вы сначала отправили сообщение №1, а затем сообщение №2, то подписчику они могут прийти в обратном порядке.

**Apache Kafka**

* Apache Kafka однозначно подходит, если вы работаете с big data. Репликация и параллельная обработка теоретически могут масштабировать вашу систему до бесконечности. Плюс Kafka выигрывает по производительности: может достичь пропускной способности в миллионы сообщений в секунду даже при ограниченных ресурсах.
* Решение позволяет программистам извлекать из очереди сообщения (Message Queuing) за любой момент времени.
* В некоторых системах порядок совершения событий имеет критическое значение — например, груз не может быть доставлен до того, как был отправлен. В таких системах необходимо использовать Apache Kafka: он гарантирует порядок доставки сообщений.

Выбирайте RabbitMQ, если вам нужна надежность и гибкость маршрутизации, а порядок доставки сообщений безразличен.

Apache Kafka подойдет, если работаете с большими нагрузками, вам важна масштабируемость, доставка сообщений в правильном порядке и возможность просматривать историю сообщений.