УДК 004.27

М. Н. Карпович
Белорусский государственный технологический университет

ОСОБЕННОСТИ ПРОЕКТИРОВАНИЯ МИКРОСЕРВИСНО-СОБЫТИЙНЫХ АРХИТЕКТУР ДЛЯ ВЫСОКОНАГРУЖЕННЫХ РАСПРЕДЕЛЕННЫХ СИСТЕМ ОБРАБОТКИ ИНФОРМАЦИЙ

В статье рассматриваются характеристики различных архитектурных решений, сложности, проблемы, ключевые особенности и лучшие практики при их использовании. Выполнено сравнение монолитной, микросервисной и событийно-ориентированной архитектуры. При создании архитектуры для обработки больших потоков данных разработчики сочетают архитектуру, управляемую событиями, и архитектуру микросервисов для создания систем, которые являются чрезвычайно масштабируемыми, доступными, отказоустойчивыми, параллельными и простыми в разработке и обслуживании. Описываются и сравниваются подходы к работе с данными в микросервисной и монолитной архитектурах. Рассматриваются паттерны маршрутизации запросов, такие как API Gateway и Service Discovery. Указаны их преимущества, недостатки и особенности реализации. Сравниваются два подхода поиска адресов на стороне сервера и клиента. Приводится пример реализации событийно-ориентированной архитектуры на основе таких паттернов, как посредник и брокера. Описываются способы их реализации, особенности, сильные и слабые стороны каждого из них. Приведены сравнения подходов для межпроцессорного общения микросервисов, использующих синхронный и асинхронный тип взаимодействия. Даны сравнения брокеров сообщений по различным параметрам. В заключении обосновывается выбор архитектуры приложения для распределенной обработки больших объемов данных.

Ключевые слова: микросервисная архитектура, событийно-ориентированная архитектура, паттерн посредник, паттерн брокера.

Для цитирования: Карпович М. Н. Особенности проектирования микросервисно-событийных архитектур для высоконагруженных распределенных систем обработки информаций // Труды БГТУ. Сер. 3, Физико-математические науки и информатика. 2023. № 1 (266). С. 89–95. DOI: 10.52065/2520-6141-2023-266-1-15.

M. N. Karpovich
Belarusian State Technological University

FEATURES OF DESIGNING MICROSERVICE-EVENT ARCHITECTURES FOR HIGHLY LOADED DISTRIBUTED INFORMATION PROCESSING SYSTEMS

The article discusses the characteristics of various architectural solutions, complexities, problems, key features and best practices in their use. Comparison of monolithic, microservice and event-oriented architecture. When creating an architecture for processing large data flows, developers combine an event-driven architecture and a microservices architecture to create systems that are extremely scalable, affordable, fault-tolerant, parallel, and easy to develop and maintain. Approaches for working with data in microservice and monolithic architecture are described and eliminated. The patterns for routing requests, such as API Gateway and Service Discovery, are considered, their advantages, disadvantages and implementation features are described. Two approaches for searching addresses on the server and client side are compared. An example of the implementation of event-oriented architecture based on such patterns as: intermediary and message broker is given. The ways of their implementation, the strengths and weaknesses of each of these approaches are described. Comparison of approaches for interprocessor communication of microservices using synchronous and asynchronous type of interaction. Describes the comparison of message brokers by various parameters. In conclusion, conclusions are drawn on the choice of the optimal architecture for processing large amounts of data.

Keywords: microservice architecture, event-driven architecture, mediator pattern, broker pattern.

For citation: Karpovich M. N. Features of designing microservice-event architectures for highly loaded distributed information processing systems. *Proceedings of BSTU, issue 3, Physics and Mathematics. Informatics*, 2023, no. 1 (266), pp. 89–95. DOI: 10.52065/2520-6141-2023-266-1-15 (In Russian).

Введение. Современное программное обеспечение должно обрабатывать большие потоки данных и соответствовать сложным бизнес-требованиям.

Архитектура программ обеспечивает удобство обслуживания и расширяемость приложений. Однако выбор подходящей архитектуры – нетривиальная задача. Важными являются тщательное рассмотрение требований к функциям приложения, временным рамкам проекта, бизнес-требований и опыт команды разработчиков. Хотя микросервисы естественным образом подходят для сложных приложений и облачной среды, во многих случаях может быть лучше начать разработку монолитной архитектуры, а затем перенести приложение на микросервисную архитектуру.

Микросервисы – это архитектурный стиль, в котором одно приложение строится с использованием нескольких небольших сервисов [1]. Каждый сервис работает в своем собственном процессе и взаимодействует с другими службами с помощью облегченного механизма, часто синхронных веб-сервисов REST. Каждый сервис представляет собой небольшой компонент, который выполняет один бизнес-процесс – аутентификацию, уведомление, обработку платежей и т. д. В разных сервисах могут использоваться разные технологические стеки, базы данных, фреймворки и языки программирования. Они также могут быть независимо развернуты и масштабируемы [2, 3].

Связь между различными микросервисами в приложении обычно реализуется с помощью веб-сервисов REST или архитектуры, управляемой событиями (EDA – Event Driven Architecture), и потоковой обработки. Архитектура, управляемая событиями, это слабосвязанная архитектура, основанная на асинхронной обработке потока событий в реальном времени. EDA – популярный шаблон распределенной асинхронной архитектуры, который может быть использован для решения проблемы работы с распределенными данными.

Он обладает высокой масштабируемостью и гибкостью. В EDA каждый микросервис публикует событие, когда происходит какое-то действие, т. е. служба заказов будет публиковать новое событие, когда заказ был создан или изменен. Остальные микросервисы подписываются на интересующие их события. Например, служба инвентаризации подпишется на события нового заказа, так как ей необходимо уменьшить количество соответствующих товаров в базе данных инвентаризации. Событие можно определить как «существенное изменение состояния» [3].

Хотя концепция EDA возникла в начале 2000-х гг., в последние годы она привлекла большое внимание. EDA является предпочтительным архитектурным стилем для многих облачных приложений, требующих высокой доступности и быстрой пропускной обработки данных.

Цель данной статьи – анализ и выбор подходящей архитектуры для разработки высоконагруженного распределенного приложения обработки большого количества данных.

Основная часть. Сравним микросервисы с традиционной монолитной архитектурой. Монолитные приложения имеют следующие преимущества [2]:

– простота разработки и тестирования. Если приложение относительно небольшое: большинство корпоративных приложений имеют монолитную многоуровневую архитектуру (уровень представления, уровень интеграции (сервиса), уровень бизнес-логики, база данных и слой доступа);

– удобство развертывания приложений (есть только одно приложение, которое необходимо развернуть и настроить);

– масштабирование приложения (при необходимости балансировщик нагрузки может распределить нагрузку между несколькими экземплярами приложения);

– модульная структура приложений (модули приложений взаимодействуют друг с другом через удаленные вызовы процедур).

Многие из существующих успешных приложений проектировались как монолитные. Однако по мере роста размера приложений следует учитывать некоторые недостатки монолитной архитектуры [2]:

– если приложение становится слишком большим и сложным, разработчикам значительно труднее внедрять новые функции или быстро вносить изменения;

– тестирование становится намного сложнее, поскольку изменения могут привести к дефектам регрессии в любом модуле;

– снижение надежности (проблемы в любом модуле приложения могут привести к остановке всего приложения);

– внедрение новых технологий становится сложнее, поскольку обновление фреймворков или языковой версии влияют на все приложение;

– непрерывное развертывание больших монолитных приложений затруднено – обновление в одном модуле потребует повторного развертывания всего приложения, что может быть дорогостоящим с точки зрения аппаратных ресурсов и времени.

В микросервисной архитектуре приложение разбивается на набор небольших, слабо связанных сервисов, каждый сервис из которых отвечает за один бизнес-процесс. Например, приложение для онлайн-покупок может состоять из сервиса каталога товаров, сервиса инвентаризации, сервиса заказов, сервиса доставки, сервиса пользователей, сервиса рекомендаций по продуктам и т. д.

Каждый микросервис использует собственную базу данных. Хотя такой подход может привести к дублированию данных, он обеспечивает слабую связанность сервисов.

Преимущества микросервисов заключаются в следующем [2]:

– проблема сложности больших монолитных приложений решается путем разбиения приложения на много мелких сервисов. Каждый сервис разрабатывается и поддерживается одной выделенной командой;

– простое внедрение новых технологий: небольшая команда, поддерживающая соответствующий сервис, несет ответственность для выбора технологий реализации;

– простое и быстрое развертывание: каждую службу можно развертывать независимо. Непрерывное развертывание также намного проще организовать по сравнению с большими монолитными приложениями;

– упрощенная масштабируемость: каждый сервис можно масштабировать независимо;

– отказоустойчивость: приложение работает, даже если один сервис выходит из строя.

Тем не менее следует отметить следующие недостатки микросервисов [2]:

– первоначальная стоимость внедрения: архитектура микросервисов хорошо работает только для сложных приложений. Разработка и реализация взаимосвязанных сервисов требует много времени в начале процесса разработки, а также есть необходимость в опытном архитекторе и команде;

– существует значительная нагрузка на связь из-за частых вызовов микросервисов;

– бизнес-транзакции, которые требуют обновления нескольких бизнес-объектов, должны обновлять разные базы данных, принадлежащие разным сервисам;

– трудно вносить изменения, затрагивающие несколько сервисов.

В разработке программного обеспечения шаблоны проектирования [3] являются хорошим общим решением перечисленных проблем. Gateway API и Service Discovery – два наиболее часто используемых шаблона в микросервисной архитектуре.

При использовании монолитной архитектуры клиенту, скорее всего, потребуется сделать только один вызов REST сервиса, который соберет необходимую информацию. Однако при использовании микросервисов существуют следующие варианты:

– вызвать каждый сервис напрямую, чтобы получить необходимую информацию. В предложенном простом примере клиенту нужно сделать небольшое количество вызовов. Однако для больших приложений клиенту может потребоваться вызвать сотни сервисов. Некоторые сервисы могут использовать разные протоколы связи, из-за чего процесс доступа к данным усложняется;

– процесс по обновлению данных становится намного сложнее, если клиенты обращаются к ним напрямую.

Для маршрутизации запросов в микросервисах используют два паттерна: API Gateway и Service Discovery.

API Gateway похож на шаблон объектно-ориентированного проектирования Facade [4]. Он инкапсулирует внутреннюю архитектуру системы и предоставляет каждому клиенту определенный API. Этот один адрес указывает на систему и часто обрабатывает входящие запросы, вызывая несколько служб и объединяя полученные результаты. Такой подход упрощает клиентский код и сокращает количество взаимодействий между клиентом и приложением [4].

Важно отметить потенциальные недостатки Gateway API:

– это дополнительный компонент приложения, который необходимо разработать, развернуть и которым нужно управлять;

– это может стать узким местом при использовании в большой системе. Разработчики должны обновить API Gateway для изменения или обновления функциональных возможностей системы.

В традиционных корпоративных приложениях сетевые расположения различных сервисов являются статическими, и могут храниться в файлах конфигурации приложений. Однако в облачных приложениях IP-адреса различных служб назначаются динамически и часто меняются. Паттерн Service Discovery предлагает решение проблемы динамических адресов. Он использует реестр служб, где хранится информация обо всех доступных конечных точках сервиса.

Существует два шаблона для обнаружения микросервисов:

– поиск адресов на стороне сервера: клиенты используют балансировщик нагрузки для

отправки запросов. Балансировщик нагрузки запрашивает реестр службы и направляет запросы к соответствующему экземпляру службы [5];

– поиск адресов на стороне клиента: клиенты несут ответственность за поиск сетевых адресов доступных сервисов и для запросов балансировки нагрузки между сервисами. Когда сервисы запускаются, они регистрируют свои адреса в реестре услуг.

Реализовав поиск на стороне клиента, разработчики могут реализовывать приложение с конкретными инфраструктурными решениями по балансировке нагрузки [6].

В монолитных приложениях управление данными относительно простое – имеется только одна база данных [7]. При использовании реляционных баз данных приложения могут использовать ACID (atomicity (атомарность), consistency (согласованность), isolation (изоляция), durability (надежность)) и выполнять изменение нескольких строк и таблиц за одну транзакцию. Однако в микросервисной архитектуре данные каждого их них являются частными для этого сервиса и могут быть доступны только через API, который он предоставляет, что приводит к следующим основным проблемам распределенных данных:

– извлечение данных из нескольких сервисов;

– внедрение бизнес-транзакций, которые поддерживают согласованность данных между несколькими сервисами.

При организации событийно-ориентированной архитектуры сервисы рассчитаны на развертывание во множестве экземпляров. Поскольку состояние хранится в журналах событий, сам сервис не сохраняет состояние, что обеспечивает масштабирование любого сервиса.

Событие может состоять из двух частей:

– заголовок (имя, временная метка и тип события);

– тело (описывает то, что в действительности произошло).

Можно использовать события для реализации бизнес-транзакций, охватывающих несколько сервисов. Транзакции представлены серией шагов, где каждый шаг – это микросервис, который обновляет или создает бизнес-сущность и публикует событие, запускающее следующий шаг. EDA имеет две основные топологии: посредник (рис. 1) и брокера (рис. 2):

Для каждого начального шага посредник отправляет обрабатываемое событие и ждет, когда данное событие будет обработано соответствующим обработчиком. Этот процесс продолжается до тех пор, пока все события не будут обработаны. Обработчики прослушивают каналы событий и выполняют определенную бизнес-логику для работы данных событий.

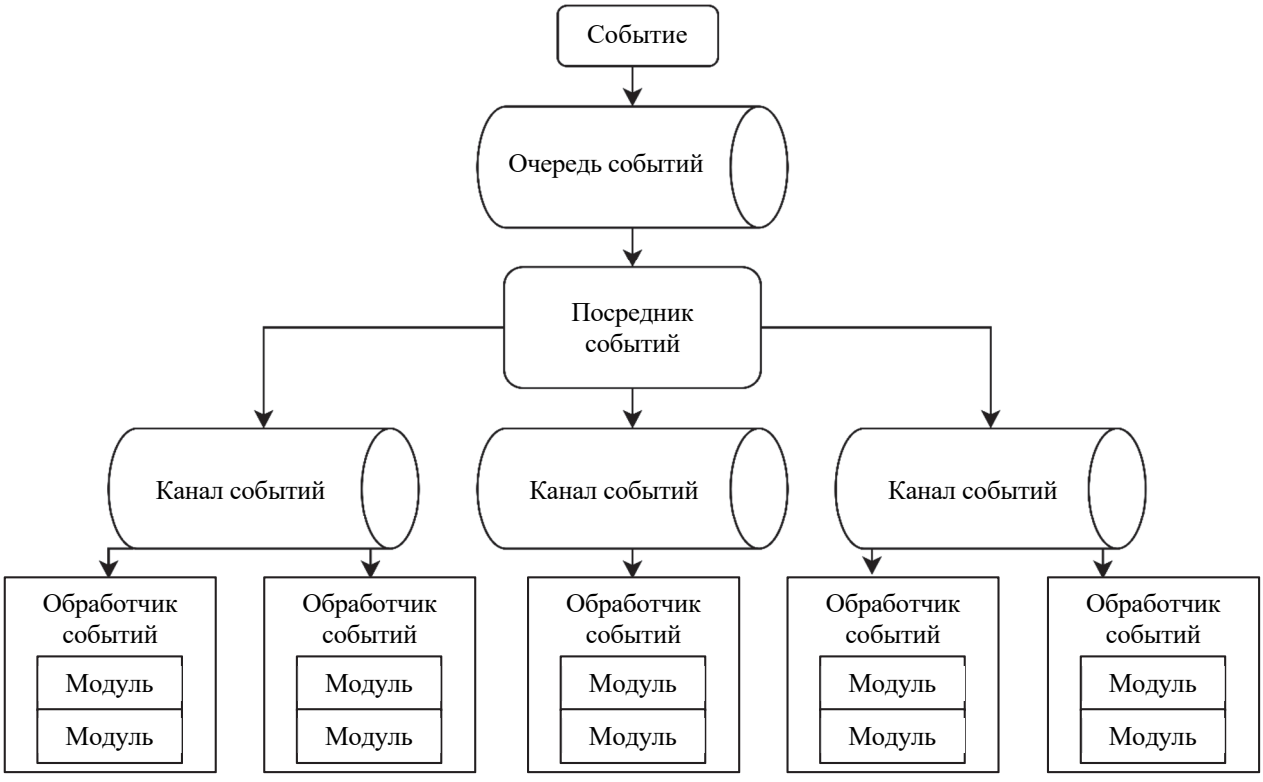

Рис. 1. Топология посредник

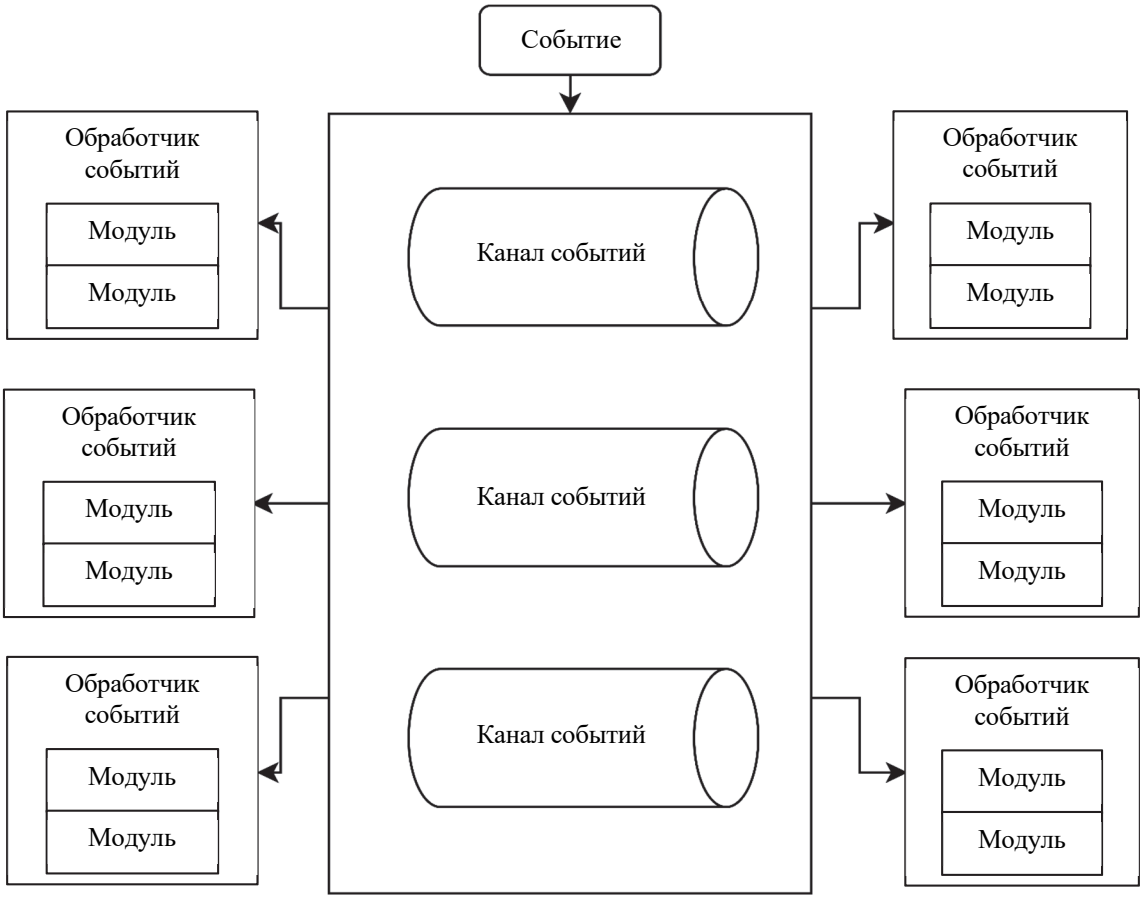

Рис. 2. Топология брокера

В топологии брокера (broker topology) нет центрального посредника событий. Брокеры сообщений наиболее широко используются для того, чтобы события могли обрабатываться несколькими потребителями событий (каждый из которых выполняет свою задачу).

В модели существует два типа компонентов: брокер и обработчики событий. Каждый обработчик отвечает за получение и публикацию нового события, чтобы уведомить других о выполненном действии. Брокер предоставляет следующие компоненты: очереди сообщений, темы сообщений или их комбинацию.

Очереди сообщений – довольно известный шаблон при проектировании системы, в которой необходимо асинхронное общение между компонентами с гарантированной доставкой, даже в случае недоступности обработчика событий [8].

Для каждой конкретной задачи определяются собственные требования к системе очередей. Основные требования, предъявляемые к очередям:

– пропускная способность;
– задержка сообщений;
– масштабируемость;
– поддержка протокола AMQP;
– упорядоченность отправки сообщений.

Основные преимущества использования очередей сообщений:

– дают возможность избежать неэффективного использования ресурсов;
– позволяют горизонтально масштабировать приложения; распределяют процессы обработки информации;
– позволяют балансировать нагрузку;
– дают возможность выдерживать пиковые нагрузки;
– повышают отказоустойчивость сервисов;
– обеспечивают порядок доставки, очередь работает по системе FIFO.

В случае сбоев в работе микросервисов механизмы асинхронной связи предоставляют различные методы восстановления данных и, как правило, лучше справляются с подобными ошибками [3].

Кроме того, при использовании брокеров сообщений вместо синхронного протокола REST, участвующего в обмене данными между сервисами, используется асинхронный способ, при котором сервисам не нужно знать друг о друге. Сравнение синхронного и асинхронного способов взаимодействия представлено в табл. 1.

Таблица 1

Сравнение режимов взаимодействия сервисов

Режим взаимодействия	Достоинства	Недостатки
Синхронный	Работает в режиме реального времени; простая обработка ошибок	Может занять много времени при обработке запроса
Асинхронный	Запросы не блокируют друг друга; несколько действий могут выполняться параллельно	Сложнее отследить ошибки; разное время отклика на одинаковые события

Асинхронный режим расширяет возможности создания центрального средства обнаружения, мониторинга и выравнивания нагрузки, а также программ контроля соблюдения политик. При написании кода и создании системы он предоставляет больше возможностей, включая адаптивность и масштабируемость.

Из вышеперечисленного утверждения следует, что асинхронная связь, по определению, не блокируемая. Она поддерживает лучшее масштабирование по сравнению с синхронным режимом.

Асинхронная передача данных обычно выполняется через брокера сообщений. Другие способы, такие как AsyncIO, имеют ограниченные возможности [9].

Сравнение брокеров сообщений с точки зрения эффективности обмена сообщений по различным критериям представлено в табл, 2.

Таблица 2

Сравнение брокеров сообщений

Брокер сообщений	Масштабирование (количество запросов)	Поддерживаемые платформы	Режим one-to-one	Режим one-to-many
RabbitMQ	50 тыс./с	10	+	+
Kafka	1 млн/с	18	–	+
Pulsar	500 тыс./с	7	–	+
NATS	9 млн/с	48	+	+

При выборе брокера асинхронного режима следует учитывать следующие требования:

– масштабирование брокера – количество сообщений, отправляемых в системе, в секунду;

– количество поддерживаемых платформ;

– способность управлять клиентами в режиме one-to-one и/или one-to-many.

Как видно из табл. 2, при больших объемах данных лучший выбор – Kafka, NATS, Pulsar. В этом случае требуется распределенная очередь с высокой пропускной способностью, созданная для длительного хранения больших объемов данных. Перечисленные варианты идеально подходят в тех случаях, где требуется персистентность.

Для сложной маршрутизации подойдет RabbitMQ. Это давно известный, популярный брокер со множеством функций и возможностей, поддерживающих сложную маршрутизацию при незначительном объеме трафика (несколько десятков тысяч сообщений в секунду) [10].

Заключение. Согласно проведенному анализу, наиболее эффективной с точки зрения производительности при обработке большого количества данных является микросервисная архитектура, основанная на событиях.

Разработчики могут комбинировать архитектуру, управляемую событиями, и микросервисную архитектуру для разработки распределенных, высокодоступных, отказоустойчивых и высокопроизводительных приложений. Приложения могут обрабатывать большие объемы данных и при этом будут широко масштабируемыми. Однако, разработчики должны учитывать множество архитектурных проблем и сложностей, на основе которых нужно принимать ключевые архитектурные решения.

В данной статье рассмотрены архитектурные решения для разработки высоконагруженных приложений и факторы, которые необходимо учитывать при принятии решений. Приведено сравнение синхронного и асинхронного способов взаимодействия микросервисов. Обоснованы рекомендации использования асинхронного режима.

Сформулированы основные требования, предъявляемые к решениям коммуникаций в микросервисах на основе очереди сообщений. Выполнен анализ основных провайдеров обмена сообщениями: RabbitMQ, Kafka, NATS, Pulsar по критериям масштабируемости, управляемости, персистентности, гарантии доставки и количеству поддерживаемых платформ.

Исходя их этого можно сделать вывод, что наиболее эффективным с точки зрения пропускной способности, масштабируемости и отказоустойчивости является событийно-ориентированная микросервисная архитектура, использующая топологию брокера и асинхронный способ взаимодействия сервисов между собой.

Список литературы

1. Карпович М. Н. Проектирование микросервисных архитектур информационных систем // Информационные технологии: материалы 86-й науч.-техн. конф. проф.-препод. состава, науч.

сотрудников и аспирантов (с междунар. участием), Минск, 31 января – 12 февраля 2022 года. Минск: БГТУ, 2022. С. 82–85.

2. Newman S. Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith. USA: O'Reilly Media Publ., 2019. P. 255.

3. Bellemare A. Building Event-Driven Microservices. USA: O'Reilly Media, 2020. 304 p.

4. Richardson C. Microservices Patterns: With examples in Java. USA: Manning, 2018. 520 p.

5. Gough J., Bryant D., Auburn M. Mastering API Architecture. USA: O'Reilly Media, 2021. 268 p.

6. Christudas B. Practical Microservices Architectural Patterns. USA: Apress, 2019. 934 p.

7. Bruce M., Pereira P. Microservices in Action. USA: Manning, 2018. 392 p.

8. Filipe H., Rocha O. Practical Event-Driven Microservices Architecture. USA: Apress, 2021. 166 p.

9. Stopford B. Designing Event-Driven Systems. USA: O'Reilly Media, 2018. P. 75–80.

10. Mertcan A. Redis vs Kafka vs RabbitMQ // Buy me a coffee. URL: https://buymeacoffee.com/argucmertcan/redis-vs-kafka-vs-rebbitmq (дата обращения: 22.12.2022).

References

1. Karpovich M. N. Designing microservice architectures of information systems. *Informatsionnyye tekhologii: materialy 86-y nauchno-tekhnicheskoy konferentsii professorsko-prepodavatel'skogo sostava, nauchnykh sotrudnikov i aspirantov* [Information technologies: materials of the 86th scientific-technical conferences of teaching staff, researchers and postgraduates (with international participation)], Minsk, January 31 – February 12, 2022. Minsk, 2022, pp. 82–85 (In Russian).

2. Newman S. Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith. USA, O'Reilly Media Publ., 2019. 255 p.

3. Bellemare A. Building Event-Driven Microservices. USA, O'Reilly Media Publ., 2020. 304 p.

4. Richardson C. Microservices Patterns: With examples in Java. USA, Manning Publ., 2018. 520 p.

5. Gough J., Bryant D., Auburn M. Mastering API Architecture. USA, O'Reilly Media Publ., 2021. 268 p.

6. Christudas B. Practical Microservices Architectural Patterns. USA, Apress Publ., 2019. 934 p.

7. Bruce M., Pereira P. Microservices in Action. USA, Manning Publ., 2018. 392 p.

8. Filipe H., Rocha O. Practical Event-Driven Microservices Architecture. USA, Apress Publ., 2021. 166 p.

9. Stopford B. Designing Event-Driven Systems. USA: O'Reilly Media Publ., 2018, pp. 75–80.

10. Mertcan A. Redis vs Kafka vs RabbitMQ: Buy me a coffee. Available at: https://buymeacoffee.com/argucmertcan/redis-vs-kafka-vs-rebbitmq (accessed 22.12.2022).

Информация об авторе

Карпович Максим Николаевич – магистрант кафедры программной инженерии. Белорусский государственный технологический университет (220006, г. Минск, ул. Свердлова, 13а, Республика Беларусь). E-mail: karpovich@belstu.by

Information about the author

Karpovich Maksim Nikolaevich – Master's degree student, the Department of Software Engineering. Belarusian State Technological University (13a, Sverdlova str., 220006, Minsk, Republic of Belarus). E-mail: karpovich @belstu.by

Поступила после доработки 30.01.2023