УДК 004.27

М.Н. Карпович, маг. (БГТУ, Минск)

ПРОЕКТИРОВАНИЕ МИКРОСЕРВИСНЫХ АРХИТЕКТУР ИНФОРМАЦИОННЫХ СИСТЕМ

Микросервисная архитектура – подход к разработке программного обеспечения, при котором приложение разбивается на небольшие автономные компоненты (микросервисы) с четко определенными интерфейсами.

Монолиты – это приложения, построенные как единое целое, где вся логика по обработке запросов помещается внутрь одного процесса. Разумеется, монолиты могут иметь модульную структуру – содержать отдельные классы, функции, пространства имен. Но связи между этими модулями настолько сильны, что изменение каждого из них неизбежно отражается на работе приложения в целом [1].

Недостатки монолитной архитектуры:

1. избыточность сборок и развертывания;
2. невозможность масштабирования части приложения;
3. увеличение цены сбоя;
4. сложность внедрения новых технологий;
5. организационные сложности;
6. редкость обновлений;
7. сильная зависимость от модели данных.

Для небольших и редко обновляемых приложений такая архитектура может работать прекрасно, но по мере наращивания функциональности межмодульные связи в монолите будут неизбежно увеличиваться и усложняться, изменения в одних модулях будут все больше влиять на другие – в итоге дальнейшее развитие таких систем становится крайне затруднительным. В этом случает лучшим решением будет использование микросервисов.

В отличие от монолитов, в микросервисной архитектуре приложение строится как набор небольших и слабосвязанных компонентов (микросервисов), которые можно разрабатывать, развертывать и поддерживать независимо друг от друга.

Ключевые преимущества микросервисов по сравнению с монолитами:

1. простота развертывания;
2. оптимальность масштабирования;
3. устойчивость к сбоям;
4. возможность выбора технологий;
5. небольшие команды разработки;

6. уменьшение дублирования функциональности;

7. упрощение замены сервисов при необходимости;

8. независимость моделей данных.

Несмотря на множество преимуществ, микросервисы далеко не всегда оказываются оптимальным вариантом, у них есть несколько особенностей, которые стоит учитывать при выборе архитектуры.

Микросервисы по своей природе распределены, а это, как известно, имеет свои недостатки: удаленные вызовы медленнее и чаще подвержены сбоям. Если ваш микросервис обращается к десятку других микросервисов, а те, в свою очередь, вызывают еще несколько, то итоговое время отклика значительно возрастает. Также по мере увеличения взаимодействий микросервисов друг с другом возрастает и число возможных точек отказа [2].

Известны несколько путей решения этой проблемы:

– изменить детализацию своих вызовов таким образом, чтобы сократить их количество;

– использовать асинхронность: при параллельном выполнении нескольких вызовов конечное время отклика будет определяться самым медленным из них, а не суммой всех задержек.

Оба метода усложняют модель программирования и увеличивают требования к квалификации разработчика. Рост числа небольших независимых сервисов неизбежно увеличивает операционную сложность. Возрастает роль непрерывной интеграции и доставки, ведь невозможно обрабатывать десятки услуг без автоматизации их тестирования и развертывания. Повышаются требования к мониторингу, особенно в силу технологической разнородности сервисов.

Чтобы справиться с возросшей нагрузкой, компании нужно овладеть целым рядом новых навыков и инструментов, и важнейший из них – внедрение культуры DevOps. Необходимо обеспечить тесное сотрудничество программистов, тестировщиков, инженеров сопровождения и прочих участников разработки продукта на всех этапах его жизненного цикла. Далеко не все организации смогут справиться с таким количеством изменений, но изменения во взаимодействии команды необходимы.

Если разработка монолитных приложений без повышения квалификации и овладения новыми навыками со временем становится затруднительной, то разработка микросервисов – вовсе невозможна.

Микросервисы порождают возможные проблемы с согласованностью из-за применяемого в них децентрализованного управления данными. В монолитном приложении можно выполнить множество

связанных изменений за одну транзакцию, и вы будете уверены, что в случае сбоя произойдет откат и согласованность данных сохранится.

Микросервисам же требуется несколько ресурсов для выполнения цепочки изменений, распределенные транзакции не приветствуются – поэтому может возникнуть ситуация, когда при обновлении одного компонента временно перестанет отвечать другой, ожидая завершения операции на первом.

При разработке определенных сервисов можно отдать предпочтение не согласованности, а доступности: чтобы в случае обновления или выхода из строя одного сервиса, другие продолжали работу. Но делать это нужно крайне осторожно, чтобы бизнес-логика не принимала решений на основе противоречивой информации [3].

Признаками того, что для приложения необходимо применять микросервисную архитектуру следующие:

1. большая команда: если у вас работает больше 10 человек, команда растет, все сложнее погружать новичков целиком в предметную область, то микросервисы помогут стандартизировать разработку и упростить командную работу;

2. множество взаимодействующих модулей в приложении: если их количество измеряется десятками, однозначно стоит задуматься о микросервисах;

3. объемный код: приложение с многомиллионными строками кода со временем все тяжелее поддерживать и развивать как монолит;

4. долгое время запуска приложений (полчаса и более): переход на микросервисы позволит устранить вынужденные простои и эффективно использовать время разработчиков;

5. различные требования к ресурсам в рамках одного приложения: использование микросервисов идеально в случаях, когда у разных компонентов отличаются требования к центральному процессору, памяти и так далее;

6. если необходимо обеспечить своевременный выход

обновлений, микросервисы являются предпочтительным вариантом, так как по скорости развертывания они значительно лучше монолитных приложений;

7. высокий трафик со склонностью к периодическим всплескам нагрузки: микросервисы отлично сочетаются с автомасштабированием и облачной моделью, которые позволят использовать ресурсы только тогда, когда в них есть необходимость.

ЛИТЕРАТУРА

1. Ньюмен С. Создание микросервисов 2-е издание. – СПб.: Питер, 2021. – 625 с.

2. Microservice Architecture [Электронный ресурс]. – Режим доступа: http://microservices.io/. – Дата доступа: 30.01.2021.

3. Ричардсон К. Микросервисы. Паттерны разработки и рефакторинга. – СПб.: Питер, 2019. – 544 с.

УДК 316.776

Я.Ю. Навроцкий, асп.; Н.В. Пацей, доц., канд. техн. наук
(БГТУ, г. Минск)

ПРИНЦИПЫ ГИПЕРБОЛИЧЕСКОЙ МАРШРУТИЗАЦИИ В ИНФОРМАЦИОННО-ОРИЕНТИРОВАННЫХ СЕТЯХ

Информационно-ориентированная сеть (*ICN – Information Centric Networking*) – интернет-архитектура, построенная на основе именованных объектов данных (*NDO – Named Data Object*), в которой адресным элементом протокола является контент (информация), а не конечные точки хоста, предоставляющие к ней доступ. Основным действующим проектом реализации *ICN* сети является *NDN* (*Named Data Network*). В маршрутизаторах (*forwarders*) *NDN* используется таблица маршрутизации *FIB* (*Forwarding Interest Base*) на основе которой, определяется следующий адресат сообщения запроса. Из-за того, что в *FIB* ключом является название *NDO*, а также из-за неограниченного количества имен в сети, происходит быстрое увеличение количества записей в *FIB*, что в свою очередь требует множества ресурсов [1].

Гиперболическая маршрутизация – это жадный алгоритм маршрутизации, основанный на полярных координатах узлов (r – радиус, θ – угол), расположенных в неевклидовой плоскости, которые отображают сетевую геометрию [2]. При гиперболической маршрутизации, каждый узел знает свои собственные координаты, координаты своих соседей и координаты *NDO*, которые несет запрос –сообщение *Interest*. На основе данных координат маршрутизатор вычисляет расстояние между ним, соседями, конечным получателем и отправляет сообщение соседнему узлу с оптимальным путем. При использовании такого подхода, маршрутизаторам не нужно поддерживать полноценный *FIB*, достаточно хранить небольшой кэш рассчитанных маршрутов или «мертвых» маршрутов – маршрутов, которые не способны предоставить запрашиваемы данные, что значительно сократит количество ресурсов затрачиваемых на маршрутизацию [2–4].

Вопрос присвоения гиперболических координат именам *NDN* все еще остается открытым. Основная проблема решения данного вопроса заключается в том, что в настоящее время нет крупномасштаб-