Проектирование отказоустойчивых систем с микросервисной архитектурой

Т. М. Татарникова

Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В.И. Ульянова (Ленина)

tm-tatarn@yandex.ru

Е. Д. Архипцев

Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В.И. Ульянова (Ленина)

lokargenia@gmail.com

Аннотация. **Обсуждается проблема отказоустойчивости распределённых систем на микросервисной архитектуре. Рассматриваются шаблоны к проектированию систем для повышения надёжности. Описываются критерии, которым должны соответствовать микросервисы. На примерах показано, как осуществляется использование инфраструктуры для повышения отказоустойчивости. Обсуждается проблема распределенного хранения больших данных.**

Ключевые слова: микросервисы, отказоустойчивость микросервисов, пул запросов, распределённые ресурсы

I. Введение

Отказоустойчивость системы означает, что система способна сохранять свою работоспособность даже при отказе отдельных компонентов или связанных систем, и восстанавливать работоспособность после восстановления отказавших компонентов или связанных систем. Это достигается путем расчета обеспечения деградации работоспособности системы пропорционально серьезности отказа конкретных компонентов. То есть, система должна продолжать работать стабильно даже при отказе не критичных компонентов, минимизируя влияние на остальные функциональные возможности. Кроме того, стабильность системы предполагает ее способность автоматически восстанавливать работоспособность после сбоя как отдельных компонентов, так и всей системы в целом [1]. Например, при временном отключении сети стабильная система автоматически восстанавливает связь и продолжает работу без вмешательства со стороны операторов.

Рассмотрим пример построения сервера в монолитной архитектуре. Как видно из рис. 1 система состоит из 3 компонент: балансировка нагрузки, сервер и база данных.

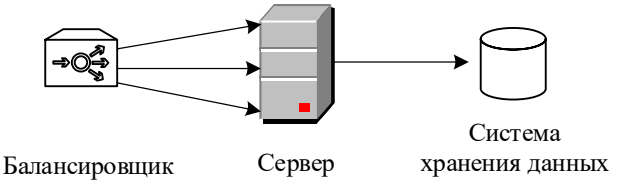


Рис. 1. Пример балансировки нагрузки в монолитной архитектуре

Монолитная архитектура является централизованной, поскольку все части приложения, включая пользовательский интерфейс, бизнес-логику и слой данных, развертываются и запускаются в одном процессе или на одном сервере.

При централизованном подходе один узел или группа узлов управляют передачей служебной информации, на основании которой происходит сбалансированное распределение нагрузки в системе. Недостатком такого подхода заключается в возможной перегрузке узла с централизованным принятием решения – наличие так называемой точки отказа [2].

В микросервисной архитектуре сервер и база данных не являются единым целым, а разбиваются на составляющие, объединенные областью применения (рис. 2). В такой архитектуре, например сервис по продаже компьютеров может быть разделен на следующие микросервисы: сервис-каталог, сервис-заказов, сервис-оплаты. Каждый из сервисов взаимодействует со своей базой данных, тем самым повышая надежность: при выходе из строя одного из сервисов. Архитектура приложения должна частично регрессировать, при этом сохранив оставшийся рабочий функционал [3].

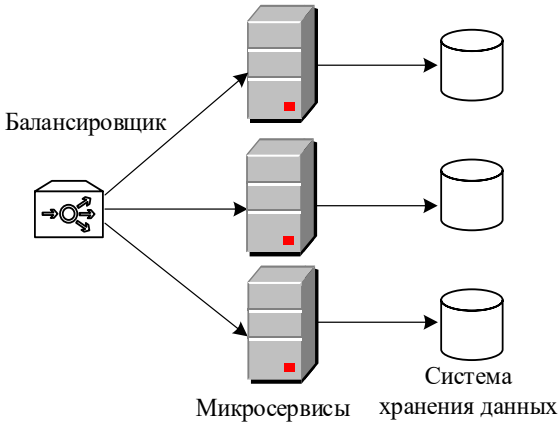


Рис. 2. Пример балансировки нагрузки в микросервисной архитектуре

Все участвующие в реализации функций балансировки нагрузки узлы обмениваются служебной информацией, на основании которой локально принимается решение с учетом собственных ресурсов узла.

На рис. 3 и рис. 4 приведены гистограммы среднего времени обработки запросов пользователей и длины очереди для монолитной и микросервисной архитектуры соответственно. Расчеты выполнены по моделям M|M|1 и M|M|3 при времени обслуживания сервера 0,1 с. Очевидно, что микросервисная архитектура является более эффективной с ростом нагрузки.

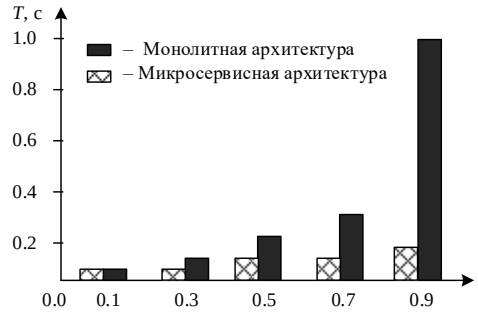


Рис. 3. Среднее время обработки запросов пользователей в монолитной и микросервисной архитектурах

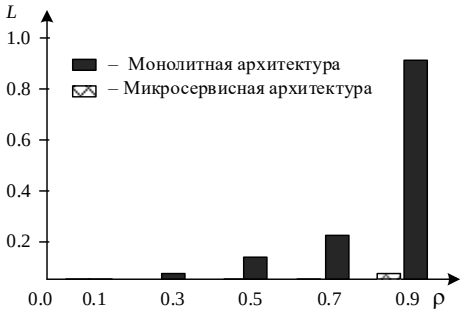


Рис. 4. Средняя длина очереди в монолитной и микросервисной архитектурах

Учитывая особенности архитектуры, можно сформулировать критерии, которым должна соответствовать система:

- надежность и отказ в каскаде (Cascading Failures) – один отказ не должен привести к другим отказам в системе;
- восстановление (Backup and Restore) – система должна иметь возможность к быстрому восстановлению.

Единственным способом создания системы, устойчивой к сбоям, является проведение стресс-тестов или хаос-инжиниринга. После анализа результатов тестов делаются выводы, которые влияют на изменения в архитектуре и подходах к написанию кода. Этот цикл повторяется до достижения нужного уровня отказоустойчивости. Количество успешно пройденных тестов становится метрикой для оценки степени отказоустойчивости как отдельных компонентов, так и всей системы в целом. Создание таких тестов представляет собой инженерную задачу, начиная с описания и автоматизации сценариев отказа, таких как потеря сети, нехватка места на диске, ошибки от внешних сервисов и т. д.

A. Отсутствие единой точки отказа

Отсутствие единой точки отказа (No SPoF) является важным принципом для обеспечения надежности и отказоустойчивости системы на всех уровнях, от отдельных компонентов до целых дата-центров. Для реализации этого принципа сервисы должны быть готовы к запуску в нескольких экземплярах и не зависеть от сохранения состояния (stateless).

Рассмотрим сценарий, когда сервис хранит состояние и не готов к горизонтальному масштабированию. Примером такого состояния может быть сессия пользователя, хранимая на конкретном микросервисе.

При горизонтальном масштабировании до нескольких экземпляров сервиса возникает проблема балансировки запросов, что может привести к тому, что запрос авторизованного пользователя попадет на микросервис, где отсутствует его сессия [3]. Для решения этой проблемы можно вынести сессии в отдельное хранилище, например, Redis, где данные хранятся в виде пар «ключ-значение» (рис. 5).

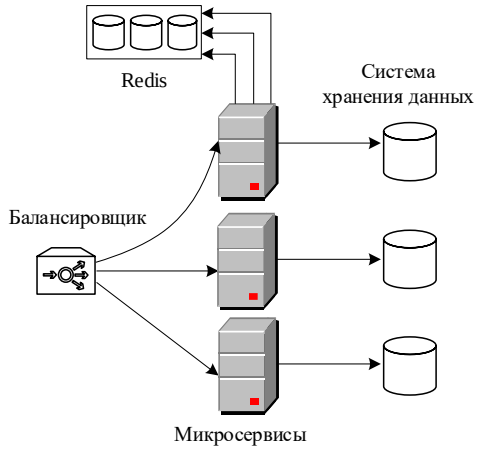


Рис. 5. Использование Redis в качестве хранилища общих данных (сессии пользователя)

Из рис. 2 и рис. 5 заметно, что для каждого микросервиса (группы эквивалентных микросервисов) используется именное хранилище. Данный подход соответствует шаблону проектирования «База данных на сервис» (Database Per Service).

Один из основных принципов при переходе на архитектуру микросервисов – обеспечить каждому сервису свою собственную область данных, чтобы избежать сильных зависимостей на уровне хранилища. Это означает логическое разделение данных, где микросервисы могут работать с общей физической базой данных, но используют отдельные схемы, коллекции или табл. [4].

Шаблон проектирования «Database Per Service» основан на этом принципе и направлен на увеличение автономности микросервисов и снижение связности между командами, разрабатывающими эти сервисы. Однако у этого подхода есть и недостатки: усложняется обмен данными между сервисами и обеспечение транзакционной согласованности ACID. Кроме того, этот шаблон не рекомендуется для небольших приложений, а скорее подходит для крупномасштабных

проектов с множеством микросервисов, где каждая команда должна иметь полный контроль над своими данными для ускорения разработки и лучшего масштабирования.

B. Создание пула соединений

В небольших проектах наиболее простым способом взаимодействия с микросервисами является прямое обращение от клиента к каждому сервису по отдельности. Однако в корпоративных приложениях с множеством микросервисов рекомендуется использовать паттерн API Gateway. API Gateway – это шлюз, расположенный между клиентским приложением и микросервисами, предоставляющий единую точку входа для клиента.

Таком образом схема приложения обретает следующий вид, приведенный на рис. 6.

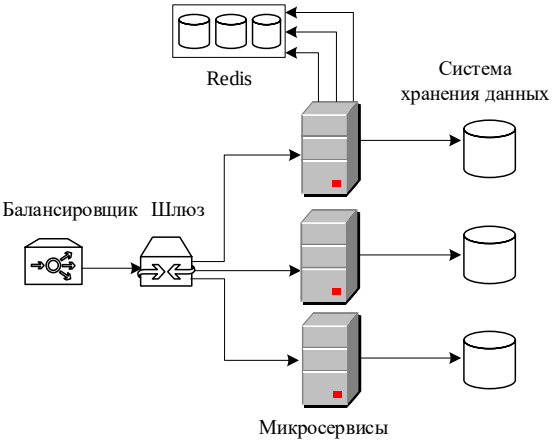


Рис. 6. Использование пула запросов

Использование пула http-соединений для обработки запросов могут вызвать следующую проблему: переполнение пула одним из сервисов. В случае, если один микросервис медленно (с задержкой) обрабатывает запросы, то это может переполнить пул соединений. Когда сервис полностью выходит из строя, запрос моментально отклоняется, что не способствует переполнению. Для решения данной проблемы необходимо завести отдельный пул на каждый из сервисов, как показано на рис. 7.

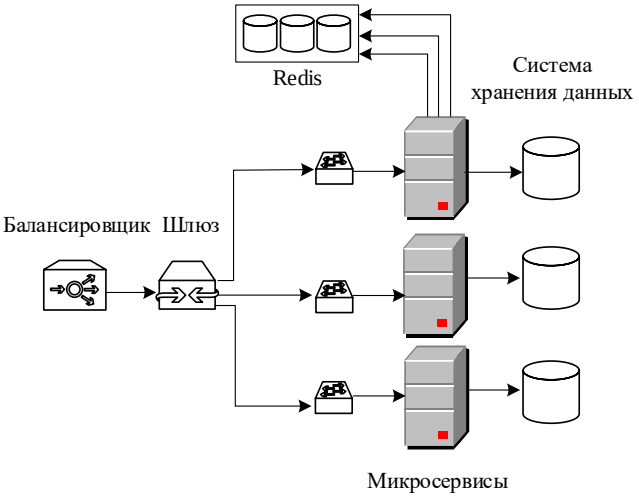


Рис. 7. Использование именной очереди для каждого сервиса

C. Повышение отказоустойчивости

В рассмотренном выше разделе для каждого микросервиса выделен отдельный пул соединений. Это решает проблему в случае с медленными и избыточными запросами, поскольку они не перегружают оставшиеся микросервисы. Однако даже в таком случае не учитываются ресурсы хоста, на котором были развёрнуты сервисы. Если один из сервисов будет полностью использовать ресурсы на машине, это приведёт к сбою и отказу системы.

Для решения данной проблемы в систему необходимо добавить контроль за ресурсами. Данный принцип описан в шаблоне «Переборка» (Bulkhead). Этот шаблон получил свое название благодаря принципам, используемым в судостроении для защиты кораблей от полного затопления в случае повреждения. Точно так же, в архитектуре приложений, он позволяет изолировать различные компоненты приложения, чтобы при сбое одного из них остальные продолжали функционировать. Этот шаблон позволяет эффективно управлять ресурсами, гарантируя, что использование ресурсов для одного сервиса не оказывает влияния на ресурсы, используемые для других сервисов.

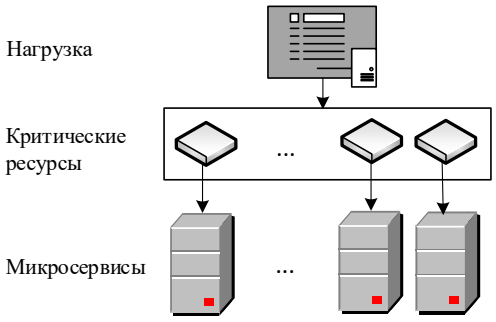


Рис. 8. Использование критических ресурсов на каждый микросервис

Как видно из рис. 8, для каждого сервиса выделено часть ресурсов хоста, с которыми может взаимодействовать лишь указанный микросервис.

В данном примере ограничения наложены на микросервисы. Такой же подход можно применить и к пользователям – назначение каждому клиенту своего собственного экземпляра сервиса. Таким образом, если один клиент генерирует слишком много запросов, перегружая свой экземпляр, это не повлияет на работу других клиентов, которые смогут продолжить свою работу независимо (рис. 9).

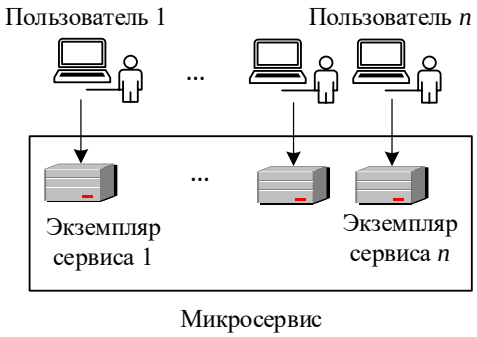


Рис. 9. Распределение ресурсов на каждого пользователя

Каждый пользователь получает свой собственный экземпляр сервиса, что уменьшает вероятность отказа системы из-за перегрузки отдельного пользователя. Это помогает изолировать проблемы, возникающие с одним пользователем, от влияния на других. Однако при использовании отдельных экземпляров могут возникать сложности с согласованием данных между ними, особенно если между ними требуется обмен информацией или синхронизация состояния. Это может потребовать дополнительных механизмов синхронизации и управления состоянием.

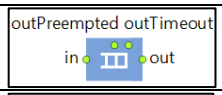
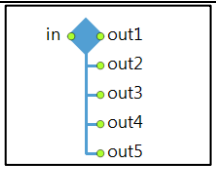
II. Сравнение микросервисных архитектур с разными найстройками надежности

Выше были рассмотрены подходы к повышению отказоустойчивости микросервисов. Произведём сравнение первоначальной настройки и архитектуры с повышенной надежностью. Под повышенной надежностью подразумевается следующее:

- Использование ApiGetway и отдельных пулов-соединений на каждом из сервисов, как на рис. 7.
- Распределение критических ресурсов на каждый микросервис.

Для выполнения сравнения микросервисной архитектуры с первоначальными настройками (сценарий 1) с микросервисной архитектурой с повышенной надежностью (сценарий 2) разработана имитационная модель в системе AnyLogic 8.7 [5]. В табл. I приведены компоненты имитационной модели.

ТАБЛИЦА I. Компоненты имитационной модели

Компонент имитационной модели	**Блок**	**Назначение**
Агент		Запрос пользователя
Блок «Приемник»		Терминал пользователя
Блок «Узел»		Микросервис
Блок «Очередь»		Очередь на обработку микросервисом
Блок «Отказ»		Блокировка потока агентов
Блок «Распределитель»		Распределение запросов по микросервисам в соответствии с вероятностями переходов
Блок «Пул ресурсов»		Моделирование модуля Redis

На рис. 10 приведен график зависимости отклоненных запросов, в % от общего количества запросов, сгенерированных в эксперименте.

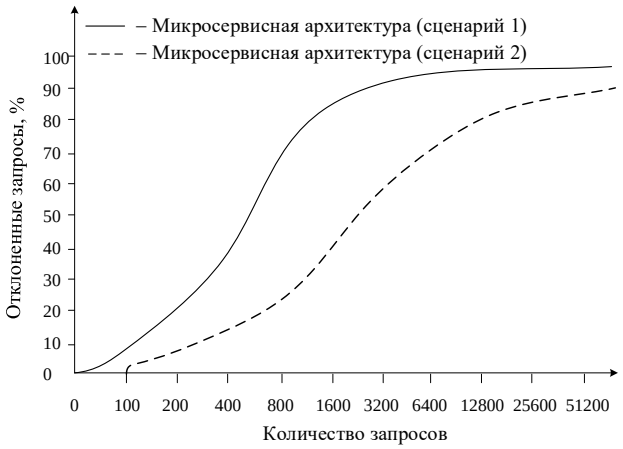


Рис. 10. Процентное соотношение отклоненный запросов

На рис. 10 видно, что сервис, организованный по сценарию 2, справляется лучше с нагрузкой, а число отклонённых запросов никогда не достигает 100 %. При таком подходе сервис не перестаёт полностью отвечать на запросы и сохраняет частичную работоспособность. В микросервисной архитектуре, организованной по сценарию 1, сервис полностью утрачивает возможность функционировать и требует восстановления.

III. Заключение

Основным критерием, которому должна соответствовать информационная система является надежность предоставляемого сервиса – обработка запросов пользователей. Надежность информационной системы обеспечивается отказоустойчивостью микросервисови, что определяет актуальность данного направления в развитии современных архитектурах информационных систем.

Рассмотрены известные способы повышения отказоустойчивости микросервисов. Приведены примеры шаблонов, который повышают отказоустойчивость микросервисов.

Эксперимент на имитационных моделях показывает, что реализация микросервисной архитектурой с повышенной надежностью позволяет сохранить частичную работоспособность информационной сиситемы.

Список литературы

[1] Ричардсон К. Микросервисы. Паттерны разработки и рефакторинга. СПб: Питер. 544 с.

[2] Советов Б.Я., Колбанёв М.О., Татарникова Т.М. Диалектика информационных процессов и технологий // Информация и космос. 2014. № 3. С. 96-104.

[3] Определение числа реплик распределенного хранения больших данных / Т.М. Татарникова, Е.Д. Архипцев // Международная конференция по мягким вычислениям и измерениям. 2023. Т. 1. С. 305-308.

[4] Татарникова Т.М., Архипцев Е.Д. Алгоритм контроллера нечеткой логики для размещения файлов в системе хранения данных // Научно-технический вестник информационных технологий, механики и оптики. 2023. Т. 23. № 6. С. 1171-1177. doi: 10.17586/2226-1494-2023-23-6-1171-1177

[5] Кутузов О.И., Татарникова Т.М. К анализу парадигм имитационного моделирования // Научно-технический вестник информационных технологий, механики и оптики. 2017. Т. 17. № 3. С. 552-558.