УДК: 004.77, 004.62

Использование систем API Gateway для оптимизации автомасштабирования приложений в оркестраторе Kubernetes

Микросервисный подход является лидирующим современным принципом построения архитектуры приложений. Такой принцип имеет такие преимущества, как слабосвязанность и малая зависимость различных компонентов приложения друг от друга, а также гибкость в масштабировании отдельных узлов системы. В данной статье предложено использование системы API Gateway в качестве единой точки доступа к компонентам приложения. Такой подход позволяет уменьшить количество удаленных вызовов между всеми компонентами приложений и упрощает их взаимодействие друг с другом. Предложен принцип работы и алгоритм системы автоматической балансировки и масштабирования API Gateway на основе платформы Kubernetes с использованием Prometheus-сервера как системы агрегации и анализа метрик. Выполнено нагрузочное тестирование предложенной системы автомасштабирования. Результаты тестирования подтверждают ее эффективность.

А. В. ШУЛЯК,
ведущий инженер по стабильности и производительности платформы

Eagle Eye Networks, Остин, США

А. Г. САВЕНКО,
магистр технических наук,
ученый секретарь,
старший преподаватель кафедры информационных систем и технологий

Институт информационных технологий БГУИР

Ключевые слова:
масштабирование высоконагруженных систем, система оркестрации Kubernetes, система мониторинга и анализа метрик Prometheus, масштабирование контейнеризированных приложений, микросерверная архитектура.

Введение. Микросервисная архитектура – вариант сервис-ориентированной архитектуры программного обеспечения, направленный на взаимодействие насколько это возможно небольших, слабо связанных и легко изменяемых модулей – микросервисов, получивший распространение в середине 2010-х годов в связи с развитием практик гибкой разработки и DevOps. Сервисы взаимодействуют друг с другом через различные протоколы такие как HTTP, RPC, GraphQL и т. д.

В микросервисном подходе, метод, вызываемый внутри процесса, является удаленным вызовом, обычно реализованным как прямая коммуникация двух компонентов приложения. Однако такой метод взаимодействия порождает большое количество дополнительных удаленных вызовов, что может привести к перегрузке одного из компонентов приложения. С точки зрения процессов разработки, с развитием бизнес-логики, может потребоваться разделить один из сервисов на дополнительные суб-сервисы, выполняющие отдельные функции. В классическом подходе коммуникаций это может привести к существенному изменению компонента приложения, который обращается к разделяемому, в связи с необходимостью изменить паттерны коммуникаций к большему числу компонентов.

Используя систему API Gateway можно скрыть разделение компонентов приложений от использующих его клиентских приложений, что может помочь минимизировать количество удаленных вызовов, а так же упростить разработку [1]. Главным минусом подобной системы является тот факт, что она становится единой точкой отказа всей системы, поскольку в случае ее выхода из строя, все сервисы приложения будут недоступны. Поэтому уже на этапе разработки архитектуры системы должна быть гарантирована высокая доступность API Gateway.

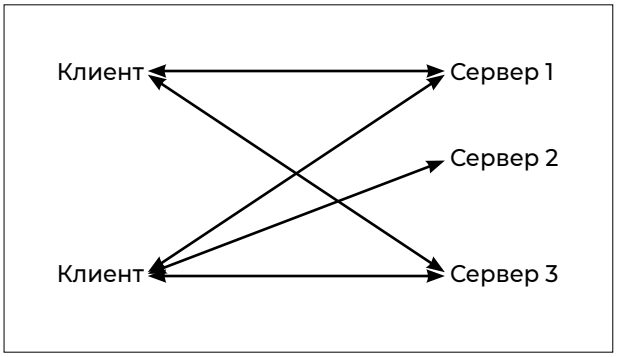

Рисунок 1 – Прямое взаимодействие клиент-серверных компонент

Основное исследование данной статьи направлено на разработку системы автоматического масштабирования для API Gateway, способную в зависимости от нагрузки динамический изменять количество экземпляров прокси-компонента API Gateway для обработки всего объема поступающего трафика и обеспечения высокой доступности.

Коммуникации в микросервисной архитектуре. В микросервисной архитектуре стандартный паттерн коммуникаций состоит из двух частей: клиента и сервера (при этом клиент может быть как внешним пользователем, так и одним из компонентов приложения) [2]. Сервер выполняет некий набор специфичных для него функций, которые возможно вызвать по определенному адресу. Клиент в свою очередь является потребителем функций сервера.

Прямая коммуникация между клиентом и сервером (рис. 1) является самым простым и гибким вариантом для взаимодействия компонентов приложения, однако множество синхронных удаленных вызовов могут повлечь за собой задержки ответов на запрос «сервер-компонент» и их сопутствующую обработки клиентом. К тому же, по мере роста бизнес-логики на любом запросе «сервер-компонент», может потребоваться его разделения на суб-сервисы, что может повлечь за собой масштабные изменения клиентов.

Конфигурация системы для проведения исследования. Для проведения исследования API Gateway использовалась система на основе сервиса Consul компании Hashicorp. Компонент API Gateway используется для проксирования и маршрутизации запросов, инструмент Consul – для автоматического обнаружения сервисов (service discovery). При данной архитектуре сервисов клиентам не требуется знать о структуре и состоянии коммуникации «сервер-компонент», у них есть единая точка входа доступа к сервису. Сконфигурированная архитектура для проведения исследования представлена на рисунке 2.

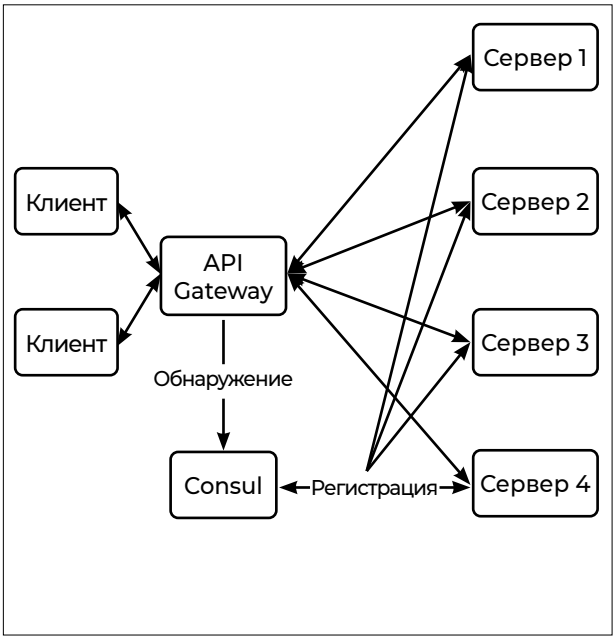

Рисунок 2 – Взаимодействие клиент-серверных компонентов приложения через API Gateway

Как видно из рисунка 2, в данной конфигурации клиент и сервер обмениваются данными только через API Gateway. Сервер при запуске регистрируется в Consul-сервисе и после этого API Gateway, при наличии обращения к данному серверу, проверяет в сервисе Consul наличие доступных экземпляров «сервер-компонент» и отправляет запрос на активный экземпляр и затем возвращает результат запроса клиенту.

Система шлюза API Gateway состоит из трех подсистем: основной подсистемы, подсистемы мониторинга и подсистемы администрирования. Архитектура системы API Gateway представлена на рисунке 3.

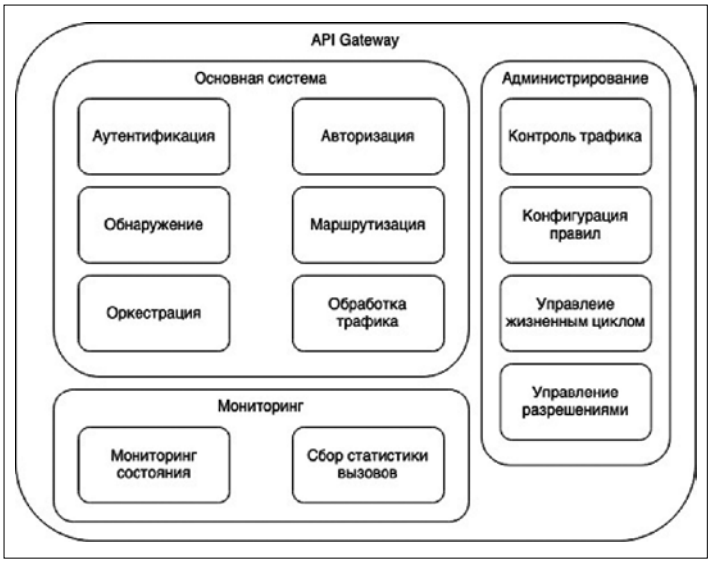

Рисунок 3 – Архитектура API Gateway

Как видно из рисунка 3, основная система представляет собой несколько главных компонентов: аутентификация и авторизация пользователей (клиентских приложений), обнаружение (service discovery), оркестрация, маршрутизация и обработка трафика. Обработка запроса сначала проходит через аутентификацию и авторизацию, затем API Gateway проверяет доступные сервера через систему service discovery при помощи Consul и затем перенаправляет его требуемому серверу.

Подсистема мониторинга направлена на отслеживание состояния внутренних компонент и производительности системы, а также сбора статистики, относящейся к трафику, проходящему через систему.

Подсистема управления позволяет настраивать правила основной системы, аутентификацию и авторизацию, контроль трафика и жизненный цикл API Gateway.

Разработка системы автомасштабирования на основе оркестратора Kubernetes и системы мониторинга Prometheus. В микросервисной архитектуре система API Gateway является единой точкой доступа к сервер-компонентам. Большое количестве запросов к API Gateway может перегрузить службу или вызвать ее отказ что приведет к недоступности всей системы одновременно. Для решения данной проблемы предлагается использовать оркестратор Kubernetes как унифицированную платформу для управления микросервисами и систему мониторинга Promethues как систему сбора и анализа метрик для корректного масштабирования API Gateway.

Kubernetes – это платформа для оркестрации контейнеров стандарта OCI (Open Container Initiative), представляющая собой систему, которая имплементирует паттерн «контейнер как сервис» [3]. Базовая архитектура оркестратора Kubernetes представлена на рисунке 4.

В оркестраторе Kubernetes минимальной единицей управления является «под» (англ. pod), который может включать в себя несколько контейнеров. Также Kubernetes как платформа исполняет роль мониторинга, супервайзинга, сбора логов и метрик запущенных контейнеров. Как видно из рисунка 4, основными элементами архитектуры Kubernetes являются:

1. Мастер-сервер. Выполняет управляющую роль и включает в себя следующие субкомпоненты:
   a) API Server – компонент, отвечающий за коммуникации между всеми остальными частям Kubernetes, а также между администратором и кластером;
   b) контроллер-менеджер – компонент, отвечающий за управление и выполнение внутренней автоматизации;
   c) планировщик – подсистема, отвечающая за распределение полезной нагрузки на исполнительные сервера;
   d) etcd – отказоустойчивое хранилище формата «ключ-значение».
2. Исполнитель-сервер. Выполняет полезную нагрузку кластера. Имеет следующие субкомпоненты:
   a) kubelet – контролирующий компонент оркестратора Kubernetes, который отвечает за настройку и мониторинг остальных компонентов на данном исполнительном узле (ноде);

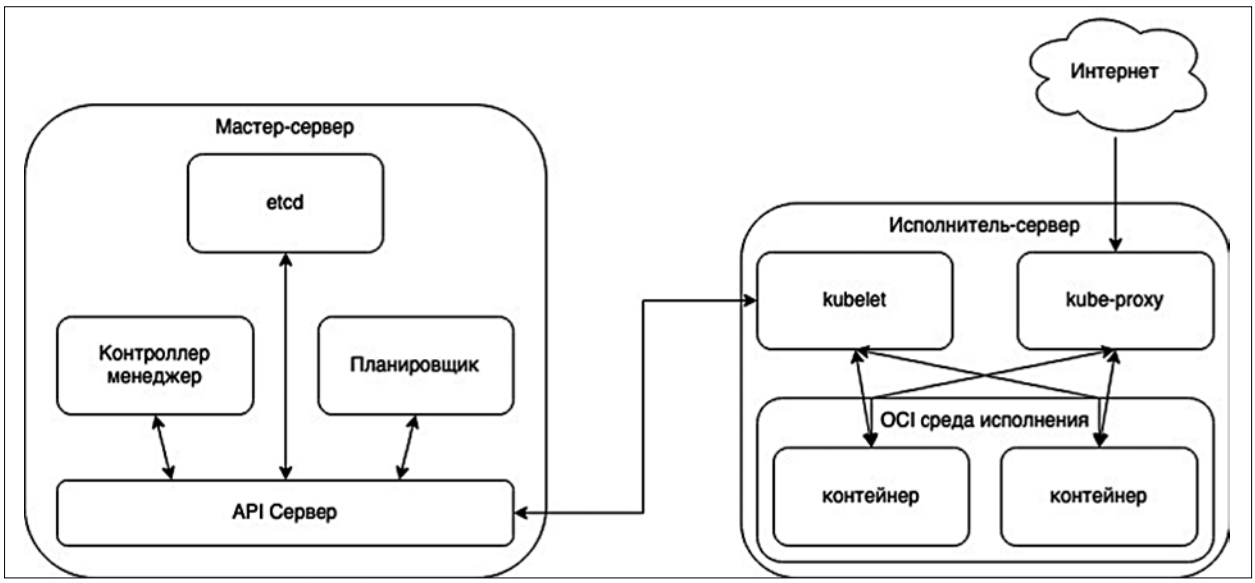

Рисунок 4 – Базовая архитектура оркестратора Kubernetes

b) kube-proxy – сетевой компонент оркестратора Kubernetes, отвечающий за сетевое взаимодействие между всеми узлами (нодами) и сетью интернет;
c) OCI среда исполнения – среда исполнения контейнерных приложений, соответствующая стандарту Open Container Initiative.

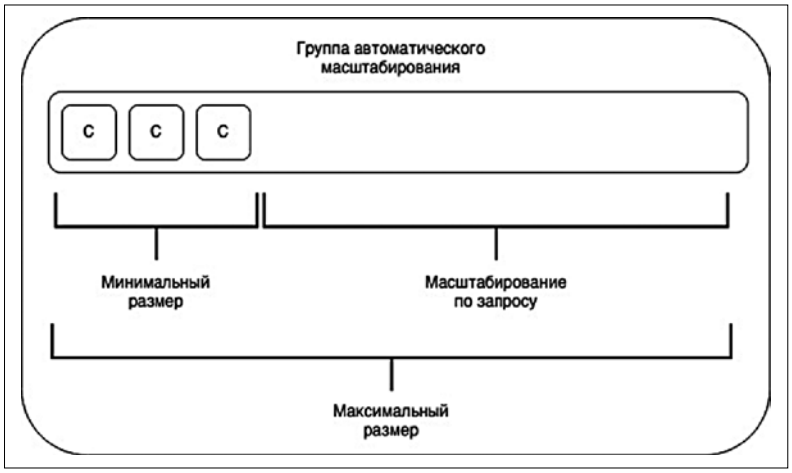

Рисунок 5 – Принцип автоматического масштабирования

Для сбора и анализа метрик используется сервис Prometheus. Сбор данных осуществляет по протоколу HTTP через сборщик метрик Node-exporter, который собирают доступную информацию от приложений через HTTP-интерфейсы и локальные API-интерфейсы системы.

Система автоматического масштабирования должна динамически регулировать количество экземпляров приложения в соответствии с рабочей нагрузкой приложения. Это может обеспечить высокую доступность приложения и оптимизировать использование системных ресурсов. Общий принцип автоматического масштабирования представлен на рисунке 5.

Таким образом, максимальный и минимальный размер группы автоматического масштабирования задается в параметрах развертываемого компонента в оркестраторе Kubernetes. Когда нагрузка на API Gateway низкая, количество экземпляров системы будет так же минимальным [4]. С ростом нагрузки система автоматического масштабирования будет увеличивать количество экземпляров в зависимости от метрик до тех пор, пока не достигнет максимально допустимого значения. В случае отказа любого из экземпляров API Gateway-системы, он будет заменен оркестратором Kubernetes на новый. Таки образом, будет обеспечена высокая доступность и автоматическое масштабирование системы API Gateway при помощи платформы Kubernetes. Иллюстрация предложенного подхода автомасштабирования представлена на рисунке 6.

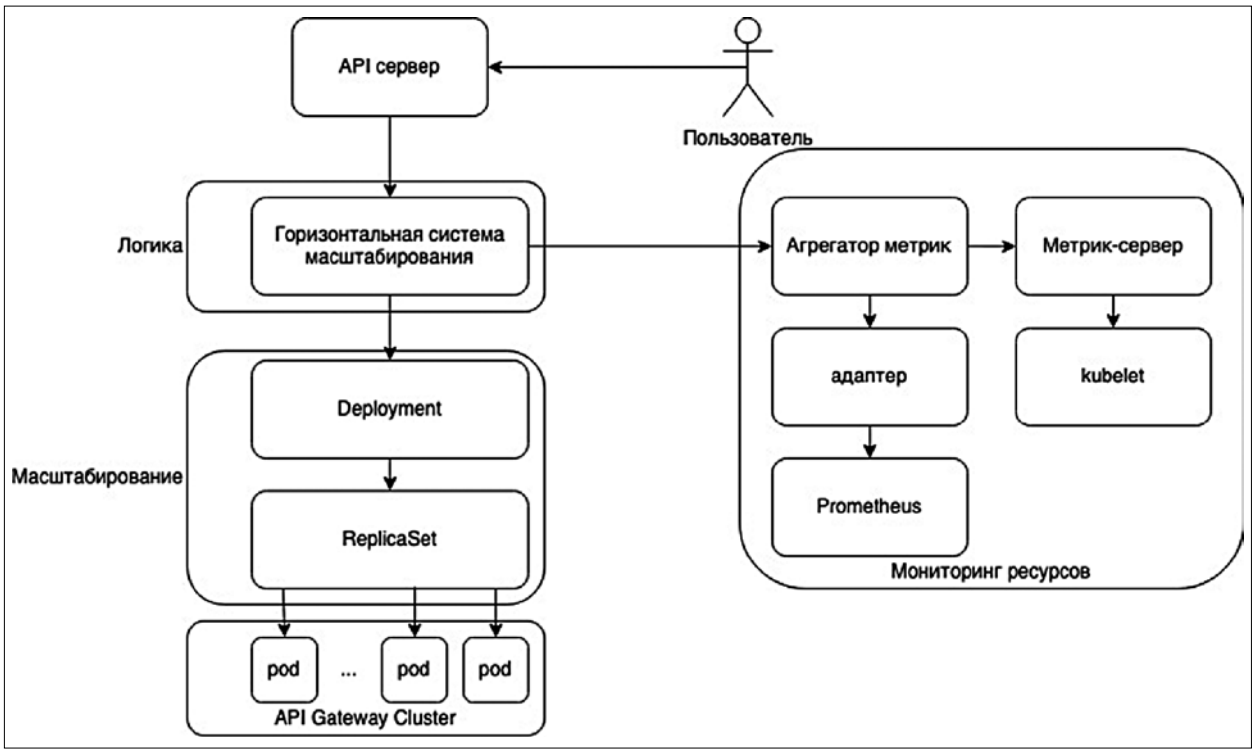

Рисунок 6 – Автоматическое масштабирования с использованием Kubernetes

Таким образом, система автомасшатабирования включает в себя 4 главных компонента:

- API Gateway Cluster – кластер API Gateway, который выполняет проксирование запросов к сервер-сервисам и предоставляет метрики по нагрузке для сервиса Prometheus через HTTP интерфейс;
- мониторинг ресурсов – подсистема мониторинга ресурсов, которая собирает, агрегирует и анализирует метрики, поступающие из API Gateay и kubelet для дальнейшего использования подсистемой логики горизонтального автомасштабирования (Horizontal Pod Autoscaling (HPA));
- логика горизонтального автомасштабирования (HPA) – анализирует метрики, поступающие от сервиса Prometheus и, по достижению пороговых границ заданных значений, начинает масштабирование кластера API Gateway;
- подсистема масштабирования – внутренняя подсистема оркестратора Kubernetes, которая отвечает за масштабирование экземпляров полезной нагрузки в соответствии с указаниями подсистемы HPA и поддержание их работоспособности.

Алгоритм работы такой системы автомасштабирования следующий:

1. Через систему горизонтального масштабирования (HPA) создается кластер API Gateway и указываются квоты на минимальное и максимальное значения по количеству выделенных ресурсов;
2. HPA выделяет ресурсы и создает требуемые экземпляры приложений, с некоторой периодичностью, опрашивая сервис Prometheus, для получения новых метрик;
3. HPA на основе полученных метрик рассчитывает количество экземпляров приложения, требуемых в данный момент времени;
4. HPA масштабирует систему в соответствии с расчетами, как в большую, так и в меньшую стороны;
5. Повторяются шаги 2–4.

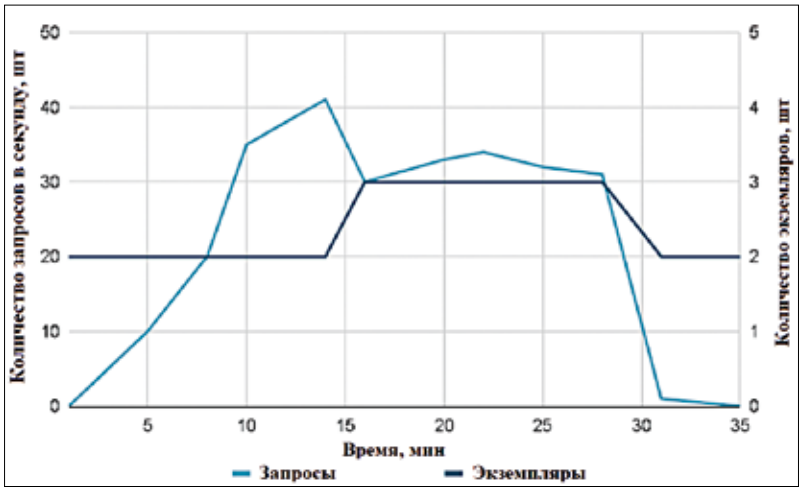

Рисунок 7 – Совмещенный график зависимости количества запросов в секунду от количества экземпляров API Gateway приложения

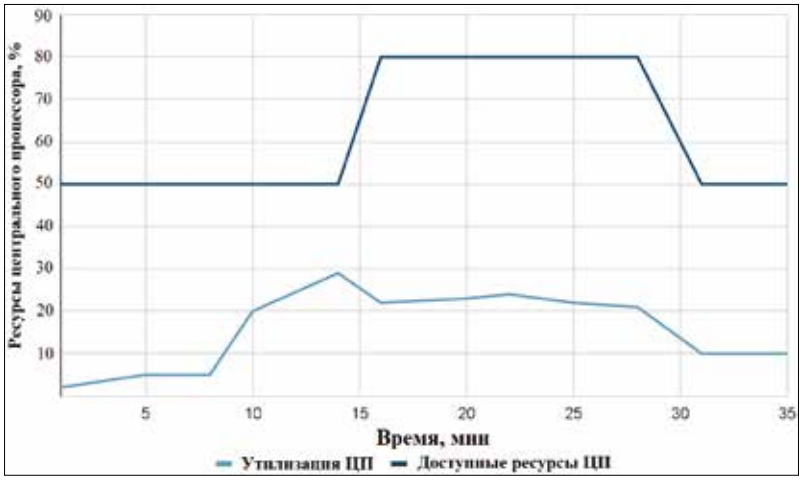

Рисунок 8 – Совмещенный график зависимости утилизации ресурсов ЦП от доступных ресурсов

Эксперимент. Тестирование предложенной системы автомасштабирования проводилось с использование пяти виртуальных машин. Конфигурация оркестратора Kubernetes представляет собой один мастер-узел (master-node) и пять рабочих узлов (node). В качестве операционной системы было использовано Flatcar Container Linux, предназначенный для оптимального использования ресурсов в контейнерной среде [5]. Характеристики виртуальных машин: 24 ядра с тактовой частой 3.2Гц и 64 гигабайта оперативной памяти.

На платформу Kubernetes были установлены два экземпляра системы API Gateway в стандартном пространстве имен. Для анализа производился сбор двух основных метрик: уровень использования ЦПУ и количество запросов в секунду. Продолжительность тестирования составила 35 минут. Был использован линейный рост запросов для моделирования постепенной нагрузки. Начальное количество запросов было равно нулю, а максимальное достигало 110 запросов в секунду. Для моделирования запросов были использованы

тестовые программы, эмулирующие поведения пользователя.

Результаты тестирования (графики зависимости количества запросов в секунду от количества экземпляров API Gateway приложения и зависимости утилизации ресурсов центрального процессора (ЦП) от доступных ресурсов) представлены на рисунках 7 и 8.

Из рисунка 7 видно, когда загрузка шлюза API достигает 45 запросов в секунду примерно за 12 минут, начинается автоматическое масштабирование и создается дополнительный экземпляр API Gateway. В свою очередь, из рисунка 8 видно, что после автомасштабирования нагрузка на систему API Gateway (утилизация ресурсов ЦП) снизилась, так как запросы распределились на дополнительный экземпляр. Затем нагрузка начала расти, но остановилась на уровне 34 запросов в секунду на каждый из экземпляров. После остановки приложения по нагрузочному тестированию нагрузка стала снижаться, и произошло автоматическое масштабирование в меньшую сторону и более не требующийся экземпляр был удален. Результаты эксперимента наглядно демонстрируют, что система может динамически регулировать количество экземпляров в зависимости от выставленных пороговых значений нагрузки.

Заключение. В данной статье был предложен подход использования сервиса API Gateway в качестве точки коммуникации с микросервисной архитектурой, разработана система автоматического масштабирования для API Gateway на основе платформы Kubernetes и системы сбора и анализа метрик Prometheus. Произведено нагрузочное тестирование предложенной системы масштабирования, результаты которого подтверждают эффективность предложенного решения.

Результаты тестирования показывают, что при использовании системы API Gateway можно уменьшить количество удаленных вызовов и упростить вызовы серверных служб друг к другу. Приложения и серверные службы взаимодействую только с системой API Gateway, что позволяет динамически менять логику серверных служб без изменения логики взаимодействия с клиентами.

Кроме того, система автоматического масштабирования может динамически регулировать количество экземпляров службы в соответствии с нагрузкой шлюза API. Он использует наименьшее количество экземпляров для балансировки рабочей нагрузки, когда нагрузка низкая. При превышении нагрузкой заданного порога, система динамически создает дополнительные экземпляры для балансировки рабочей нагрузки. Предложенный подход и разработанная система автомасштабирования может повысить рациональное использование системных ресурсов, обеспечивая при этом высокую доступность и качество обслуживания шлюза API.

ЛИТЕРАТУРА

1. **Манойло В. Е.** Паттерн «API Gateway» для эффективного взаимодействия с микросервисами / В. Е. Манойло // Научно-технические инновации и веб-технологии. – 2022. – №1. – с. 47–54.
2. **Календарев А.** Современная веб-архитектура. От мониторинга к микросервисам / А. Календарев // Системный администратор. – 2017. – № 1–2 (170-171). – с. 80–83.
3. Тарасов И. А. Разработка веб-приложения для непрерывной интеграции микросервисов на платформе контейниризации Docker / И. А. Тарасов, Д. В. Борисенков // Актуальные проблемы прикладной математики, информатики и механики. Сборник трудов Международной научной конференции ФГБОУ «Воронежский государственный университет». – Воронеж, 2021. – с. 632–638.
4. **Тусупова Б. Б.** Трехуровневая облачная инфраструктура для коммерческого приложения с использованием бессерверности в AWS / Б. Б. Тусупова, А. И. Ким, Е. Р. Ким // Polish Journal of Science. – 2021. – № 39-1 (39). – с. 51–54.
5. **Espe L.** Performanse evaluation of container runtimes / L. Espe, V. Podolskiy, M. Getndt // Closer 2020 – Proceedings of the 10th International Conference on Cloud Computing and Services Science. – 2020. – с. 273–281.

The microservice approach is the leading modern application architecture principle. This principle has such advantages as loose coupling and low dependence of various application components on each other, as well as flexibility in scaling individual nodes of the system. This article proposes using the API Gateway system as a single point of access to application components. This approach reduces the number of remote calls between all application components and simplifies their interaction with each other. The principle of operation and the algorithm of the API Gateway automatic balancing and scaling system based on the Kubernetes platform using the Prometheus server as a system for aggregating and analyzing metrics are proposed. Load testing of the proposed autoscaling system was performed. Test results confirm its effectiveness.

***Keywords**: scaling of high-load systems, Kubernetes orchestration system, Prometheus metrics monitoring and analysis system, scaling of containerized applications, microserver architecture.*

Получено 15.09.2022.