РАЗРАБОТКА ВЫСОКОНАГРУЖЕННОЙ РАСПРЕДЕЛЕННОЙ СИСТЕМЫ

Влацкая И.В., канд. техн. наук, доцент, Пятаева Е.В.
Оренбургский государственный университет

С каждым годом неуклонно возрастает число пользователей услугами сети Интернет. Востребованные веб-приложения за короткий срок набирают высокую посещаемость, в результате чего у таких систем могут возникать проблемы с отказоустойчивостью. В результате встает задача усовершенствования аппаратных ресурсов, поиска новых подходов для организации взаимодействия с базами данных и разработки оптимального программного кода.

В настоящее время нет устоявшегося определения высоконагруженной системы, но из существующей литературы можно выделить некоторые особенности, присутствующие у таких систем. Обычно высоконагруженные системы реактивные - это значит, что они не просто должны произвести вычисления, а на полученный запрос отправить ответ. Они должны делать это быстро, и если в такой системе что-то ломается пользователи не должны этого замечать. Нагрузки у таких систем обычно начинаются с 1000 RPS/QPS (Requests per second/Query per second) и 99% всех запросов должны получать ответ не более чем за 300мс. Еще одной особенностью таких систем является высокая утилизация ресурсов (50%-70%). Высоконагруженными ресурсами, в первую очередь, выступают многопользовательские приложения, многие из которых являются распределенными системами, работающими более чем на одном сервере. Таким образом высоконагруженные системы – это системы безостановочного доступа, т.е. те структуры, запрос данных которых дает возможность получать информацию без длительного перерыва при постоянной работе [1].

Актуальность данной работы состоит в том, что возрастает значимость информационных технологий в современном мире, соответственно возрастает численность посетителей веб-приложений и вопрос обеспечения отказоустойчивости при высоких нагрузках встает на передний план. Целью данной работы является построение типового приложения с высоконагруженной распределенной архитектурой.

Были определены требования к разрабатываемой системе. Разрабатываемая система должна выдерживать высокие нагрузки, то есть обеспечивать отказоустойчивость. Будем считать, что система выдерживает нагрузку, если она не выходит из строя при 1000 запросов в секунду. Система должна иметь подсистему управления доступом. Для защиты разрабатываемой системы следует реализовать защитные механизмы: аутентификация пользователей в системе, разграничение доступа пользователей к ресурсам и авторизация пользователей, регистрация и оперативное оповещение о событиях, защита от SQL-инъекций, защита от CSRF-инъекций, защита от XSS-атак. Для этого важное значение имеет использование надежных технологий, в которых уже реализована часть

защиты от основных атак. Также безопасность будет обеспечиваться за счет использования контейнерной технологии Docker.

Проанализировав типовую архитектуру высоконагруженных систем и микросервисную архитектуру, было принято решение объединить данные подходы. На рисунке 1 представлена архитектура для разрабатываемой системы. Основные причины для использования микросервисов – это аппаратные преимущества, недостижимые с помощью единой архитектуры, скорость и простота разработки одного узла системы, удобное и эффективное тестирование [2].

Чтобы повысить безопасность, а также для автоматизации механизма доставки (Continuous Delivery), развертывания (Continuous Deployment), непрерывного тестирования (Continuous Testing) и управления разрабатываемого приложения будем использовать открытую платформу для разработки, доставки и эксплуатации приложений Docker. Основные преимущества использования Docker контейнеров это легкость в обновлении и использовании образов, распределение ресурсов, изолированные среды выполнения и простота масштабирования [3].

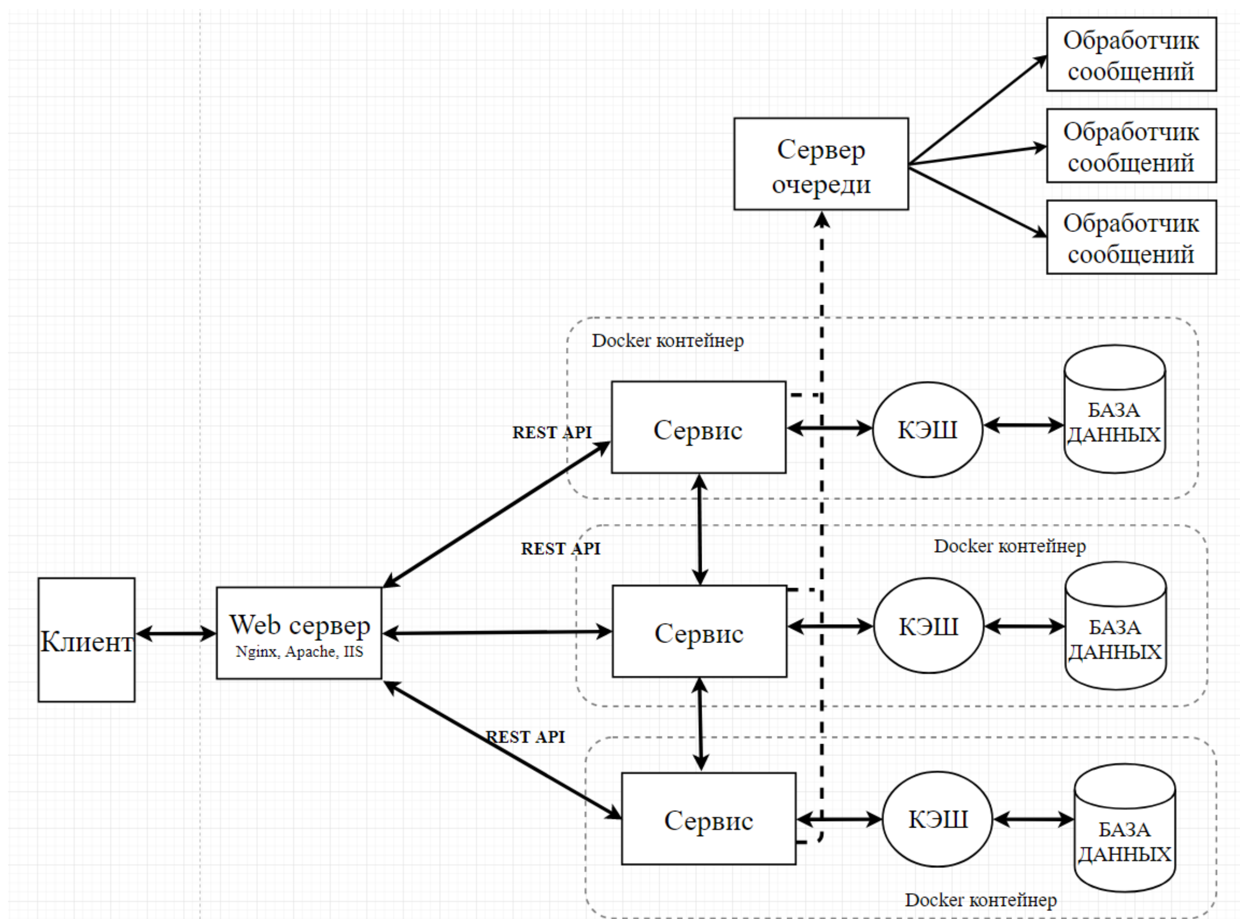

Рисунок 1 – Микросервисная архитектура высоконагруженной системы

Каждый из сервисов запускается, как отдельное приложение через Docker контейнер и не знает о существовании других, общение между ними происходит через REST.

Поскольку задача логирования является достаточно важной, и при одновременном использовании приложения несколькими десятками или сотнями клиентов, все сообщения должны быть обработаны и сохранены, то необходимо использовать очередь сообщений, чтобы избежать выполнения задачи сохранения логов непосредственно после отправки запроса. Для этого была реализована очередь сообщений посредством AMQP-брокера RabbitMQ.

Очень важной частью нашей работы является нагрузочное тестирование, поскольку разрабатываем высоконагруженную систему. Чтобы определить, является ли разработанное приложение отказоустойчивым, необходимо провести нагрузочное тестирование. Нагрузочное тестирование (Load Testing) или тестирование производительности – это автоматизированное тестирование, имитирующее работу определенного количества бизнес пользователей на каком-либо общем ресурсе [4].

В качестве инструмента для проведения нагрузочного тестирования использовали Apache JMeter, разрабатываемый Apache Software Foundation. В таблице 1 приведена зависимость времени отклика от количества пользователей, данные для которых были получены использую программу JMeter.

Таблица 1 – Таблица зависимости времени отклика от количества пользователей.

Кол-во одновременных пользователей	Минимальное время отклика, мс	Максимальное время отклика, мс	Среднее время отклика, мс
1	21	64	24
25	34	71	42
100	35	102	48
1000	38	332	97
10000	45	2845	922

На рисунке 2 представлено время отклика при пользовании приложения 1000 пользователями.

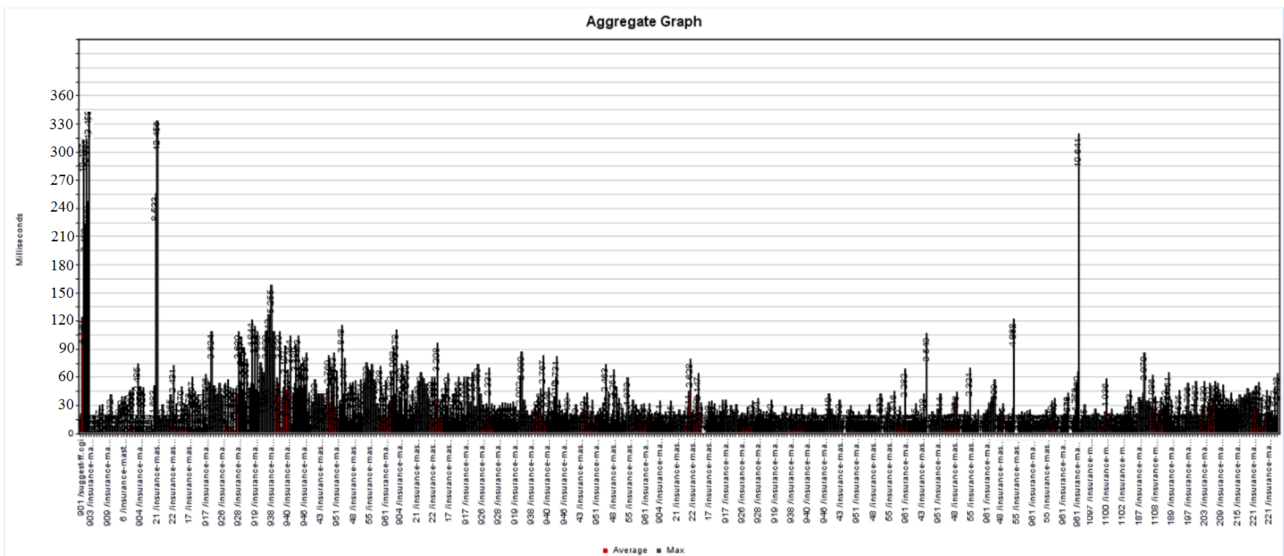

Рисунок 2 - Время отклика при 1000 одновременных запросов

Таким образом, были получены результаты, удовлетворяющие требованиям. Система при максимальной нагрузке в 10000 пользователей не вышла из строя, а максимальное время отклика при этом составило 2845 мс. Отметим, что мы запускали все контейнеры на одном сервере локально, поскольку не было возможности запускать каждый контейнер на отдельном сервере. В последнем случае показатели были бы значительно выше за счет выделения каждому сервису большее количество ресурсов.

Для мониторинга ресурсов серверной части приложения под нагрузкой использовали jvisualvm (Java VisualVM). Эта утилита находится в составе JDK и позволяет снимать дампы, анализировать производительность, состояние потоков и памяти. На рисунке 3 показана нагрузка на ЦП, на рисунке 4 – количество используемой памяти, на рисунке 5 – количество потоков при 1000 различных запросов.

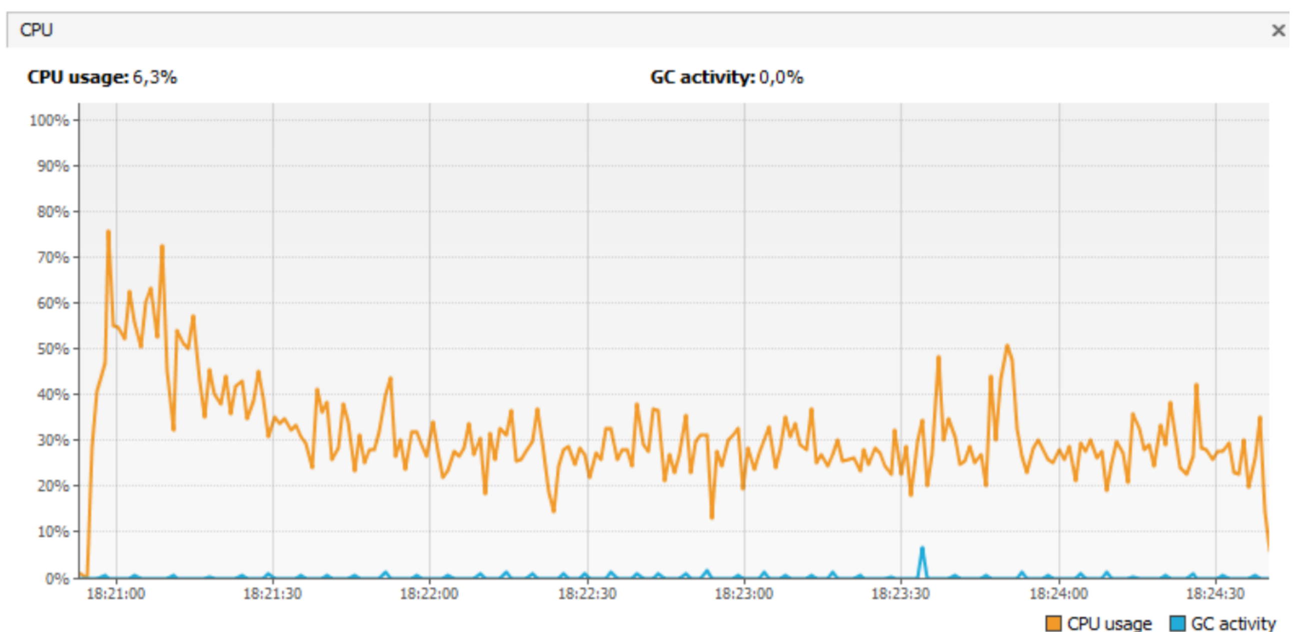

Рисунок 3 – Нагрузка ЦП при 1000 различных запросов

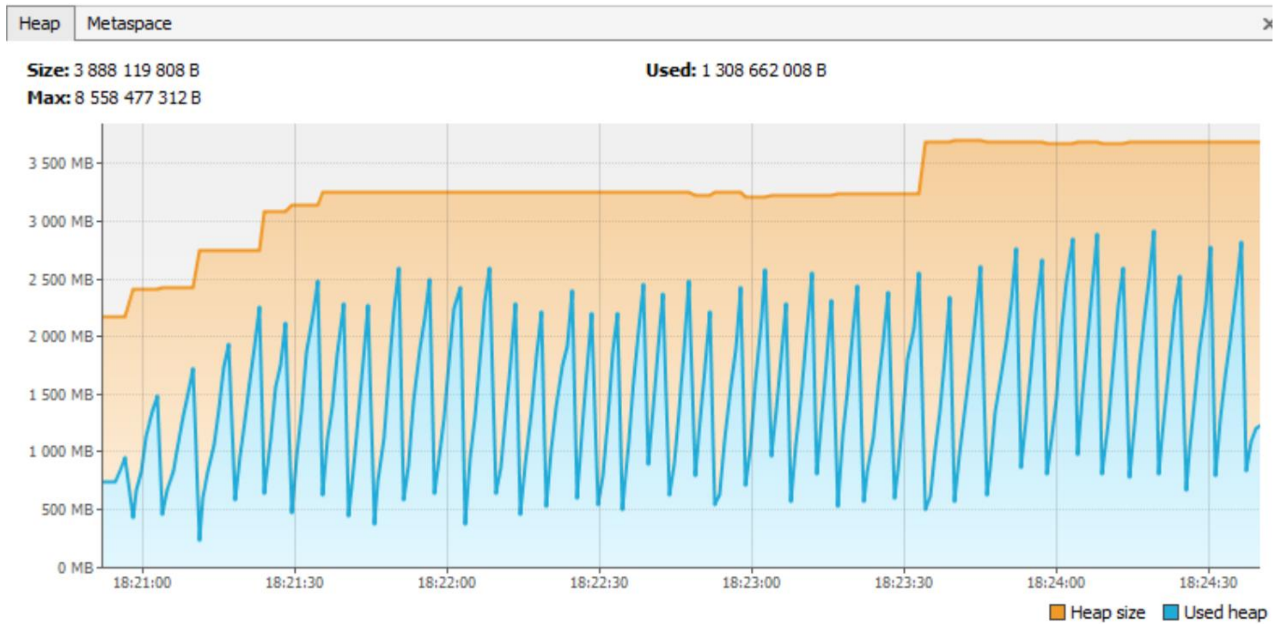

Рисунок 4 – Количеств используемой памяти при 1000 различных запросов

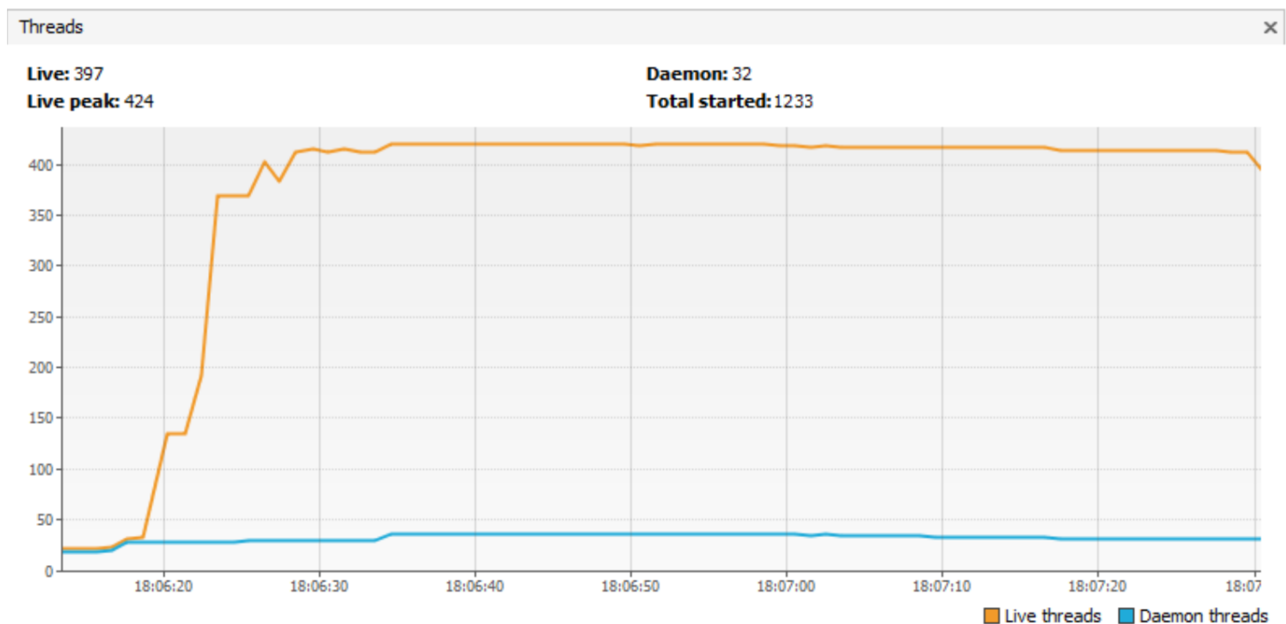

Рисунок 5 – Количество потоков при 1000 различных запросов

Исследование данной зависимости также было проведено для 1, 25, 100 и 10000 запросов. По полученным результатам была сформирована сводная таблица, результаты которой представлены в таблице 2.

Таблица 2 - Таблица зависимости показателей ресурсов от количества запросов.

Количество запросов	Средняя нагрузка ЦП	Количество используемой памяти	Количество потоков
1	6%	750 Мб	30
25	9%	800 Мб	48
100	19%	980 Мб	167
1000	67%	1,6 Гб	1256
10000	89%	2,4 Гб	15678

Данный анализ не выявил аномальных результатов в использовании ресурсов сервера под нагрузками. Начиная со 100 запросов в секунду пиковые показатели процессора примерно одинаковые, но есть различие в их продолжительности. Это обусловлено тем, что процессор тратит больше времени на обслуживание большего количества запросов. Java из коробки предоставляет несколько различных возможностей для организации работы Garbage Collector. Для анализа его работы рассмотрим частоту и длительность срабатывания сборки неиспользуемых объектов в памяти приложения. По графику на рисунке 4 видно, что при возрастании использования heap-памяти, Garbage Collector достаточно результативно освобождает память и не дает приложению упасть с ошибкой Out Of Memory Error, сигнализирующей о недостатке памяти. При увеличении одновременного количества запросов возрастает количество потоков, что объясняется тем, что для каждого запроса используется свой поток. Таким образом можно сделать вывод, что ресурсы сервера расходуются эффективно.

Список литературы

1 Henderson, C. Building Scalable Web Sites / C. Henderson – California : O'Reilly Media, 2008. – 352 c.

2 Newman, S. Building Microservices: Designing Fine-Grained Systems / S. Newman – Sebastopol : O'Reilly Media, 2015. – 280 c.

3 Matthias, K. Docker: Up & Running / J. Allspaw., S. Kane. – Sebastopol: O'Reilly Media, 2015. – 232 c.

4 Schlossnagle, T. Scalable Internet Architectures / T. Schlossnagle - California: O'Reilly Media, 2006. – 244 c.