DOI: 10.15514/ISPRAS–2020–32(6)–6



# Проектирование высоконагруженных систем

*В.А. Рудометкин, ORCID: 0000-0003-2718-7373 <vasiliy.rudometkin@gmail.com>*
*ООО СТРИМ,*
*115470, Россия, г. Москва, Проектируемый проезд № 4062, д. 6 стр. 2*

**Аннотация**. В настоящее время большинство сервисов переходят в онлайн, что позволяет пользователям получать услугу в любое время. Высокая доступность услуги ведет к росту количества пользователей, что влечет за собой повышение нагрузки на систему. Высокая нагрузка оказывает негативное влияние на компоненты системы, что может привести к сбоям функционирования и потери данных. В статье рассмотрено несколько подходов к проектированию и мониторингу, следование которым поможет предотвратить неправильное функционирование системы. Описан наиболее популярный способ распределения области ответственности каждого сервиса, в соответствии с паттерном DDD, применение которого позволит разделить компоненты системы логически по использованию и физически при масштабировании системы. Данных подход будет полезен также и при масштабировании команды, позволяя разработчикам независимо работать над разными компонентами системы, не мешая друг другу. Интеграция новых людей в проект также будет занимать кратчайшие сроки. При проектировании архитектуры системы стоит уделить внимание и схеме взаимодействия сервисов между собой. Использование паттерна CQRS позволяет разнести чтение и запись в разные компоненты, что в дальнейшем позволяет пользователю быстро получать ответ от системы. Особое внимание в статье уделено мониторингу системы, так как при увеличении размера системы время поиска ошибок в системе занимает большое время, приводя к долгой недоступности системы, что может повлечь за собой потерю клиентов. Все описанные в статье способы применены на многих проектах, например, МТС ПОИСК. Благодаря правильно спроектированной системе удалось сократить время ожидание ответа сервиса с двух минут до нескольких секунд без потери качества результата, а сложная система мониторинга системы позволяет в режиме реального времени отслеживать все процессы внутри системы и предотвращать аварии. В итоге, в начале проектирования системы следует особое внимание уделить архитектуре, вопросу мониторинга и тестирования системы. Впоследствии эти временные вложения позволят снизить риски потери данных и недоступности работы системы.

**Ключевые слова**: высоконагруженная система; DDD; REST; сервера очередей; socket; ELK; CQRS; МТС ПОИСК, проектирование

**Для цитирования:** Рудометкин В. А. Проектирование высоконагруженных систем. Труды ИСП РАН, том 32, вып. 6, 2020 г., стр. 79-86. DOI: 10.15514/ISPRAS–2020–32(6)–6

## Designing Highly Loaded Systems

*V.A. Rudometkin, ORCID: 0000-0003-2718-7373 <vasiliy.rudometkin@gmail.com>*
*STREAM LLC,*
*6c2, Projected Drive 4062, Moscow, 115470, Russia*

**Abstract.** Nowadays, most of the services are moving online, which allows users to receive the service at any time. The high availability of the service leads to an increase in the number of users, which entails an increase in the load on the system. High load has a negative impact on system components, which can lead to malfunctions and data loss. To avoid this, the article discusses several design and monitoring approaches, the observance of which will help prevent system malfunctioning. The article describes the most popular way to distribute the area of responsibility of each service, in accordance with the DDD pattern, the use of which will allow you to separate the components of the system logically by use and physically when scaling the system.





This approach will also be useful when scaling a team and allow developers to work independently on different system components without interfering with each other. The integration of new people into the project will also take the shortest possible time. When designing the system architecture, it is worth paying attention to the scheme of interaction between services. Using the CQRS pattern allows you to separate reading and writing into different components, which later allows the user to quickly receive a response from the system. Particular attention in the article is paid to monitoring the system, since with an increase in the size of the system, the time to search for errors in the system reaches a large amount of time, which can lead to a long unavailability of the system, which will entail the loss of clients. All the methods described in the article have been applied on many projects, for example, MTS POISK. Thanks to a properly designed system, it was possible to reduce the waiting time for a service response from two minutes to several seconds without losing the quality of the result, and a sophisticated system monitoring system allows you to monitor all processes within the system in real time and prevent accidents. As a result, at the beginning of the system design, special attention should be paid to the architecture, the issue of monitoring and testing the system. Subsequently, these temporary investments will reduce the risks of data loss and system unavailability.

**Keywords:** highly loaded system; DDD; REST; queue server; socket; ELK; CQRS; MTS SEARCH, system design

**For citation:** Rudometkin V.A. Designing highly loaded systems. Trudy ISP RAN/Proc. ISP RAS, vol. 32, issue 6, 2020, pp. 79-86 (in Russian). DOI: 10.15514/ISPRAS-2020-32(6)-6

## *1. Введение*

В 21-м веке все больше сервисов переходят в онлайн, позволяя выполнять большинство каждодневных операций, не выходя из квартиры. Сегодня можно заказать еду, уборку, доставку одежды, найти работу и даже купить машину или квартиру, сидя за компьютером. Эти сервисы становятся с каждым днем сложнее, охватывая все больший спектр услуг, привлекая большое количество пользователей, что приводит к повышению нагрузки на систему. Высокая нагрузка на систему приводит к долгой обработке запросов пользователей, повышает процент потери данных и отказа системы, вследствие чего клиент выберет другого поставщика услуг. Для решения этой проблемы необходимо особое внимание уделить проектированию архитектуры системы.

Высоконагруженной системой называется система, нагрузку которой не может выдержать один сервер или у которой имеются тысячи или миллионы пользователей [1]. При проектировании новой архитектуры необходимо выбрать способ управления данными, способ ограничения контекста компонентов системы и способ их взаимодействия.

В статье рассмотрено несколько подходов к проектированию и мониторингу, соблюдение которых поможет предотвратить неправильное функционирование системы. Описан наиболее популярный способ распределения области ответственности каждого сервиса, в соответствии с паттерном DDD, применение которого позволит разделить компоненты системы логически по использованию и физически при масштабировании системы.

Использование паттерна CQRS позволяет разнести чтение и запись в разные компоненты, что в дальнейшем позволяет пользователю быстро получать ответ от системы. Особое внимание в статье уделено мониторингу системы, так как при увеличении размера системы время поиска ошибок в системе достигает большого количества времени, что может привести к долгой недоступности системы, которое повлечет за собой потерю клиентов.

## *2. Родственные работы*

В [2] в общих чертах рассмотрен паттерн DDD и кеширование данных, которые меняются редко, что в высоконагруженной системе бывает крайне редко. В этой статье не рассматриваются способы кеширования данных, но подробно раскрывается способы правильного использования паттерна DDD.

В статье [3] приведен пример использования паттерна CQRS и доказана эффективность такого подхода, но не рассмотрена проблема интеграции паттерна в высоконагруженную систему. Этот вопрос необходимо рассматривать при проектировании системы, так как

неправильная интеграция в существующую систему может значительно усложнить поддержку системы, а внедрение может занять продолжительное время.

В работе [4] подробно описано проектирование систем с использованием паттернов и различных подходов, но паттерн CQRS отделен от событий. В настоящей статье предлагается рассмотреть совмещения этих двух подходов, что принесет дополнительное повышение производительности и снижение нагрузку на систему.

## 3. Управление данными

Архитектуру системы необходимо проектировать, опираясь на основные данные в системе, например, при проектировании файлового хранилища стоит определить, как будут храниться файлы – на диске с доступом по протоколу FTP или в базе данных.

Наиболее удобным способом хранения файлов является хранение ссылок на файлы в базе данных. При реализации такого решения в базе данных будут находиться идентификаторы файлов и их метаданные. На дисках или в облачных хранилищах будут храниться сами файлы. Для использования, у пользователя или другого ресурсах, которым необходим файл, хранится только идентификатор, по которому можно получить как файл, так и всю связанную с ним информацию. Основное преимущество такого подхода в том, что не важно, где хранить файл – или в базе данных в бинарном виде, например, или на диске, или в облачном хранилище.

Использовать базы данных не рекомендуется для хранения файлов, так как база данных из-за хранения файлов начинает занимать в разы больше дискового пространства, что приводит к долгому процессу создания бекапов и их восстановления.

Для задач, в которых структура данных недостаточно хорошо известна, подойдут NoSQL базы данных [5]. Основные типы нереляционных баз данных представлены на рис. 1.

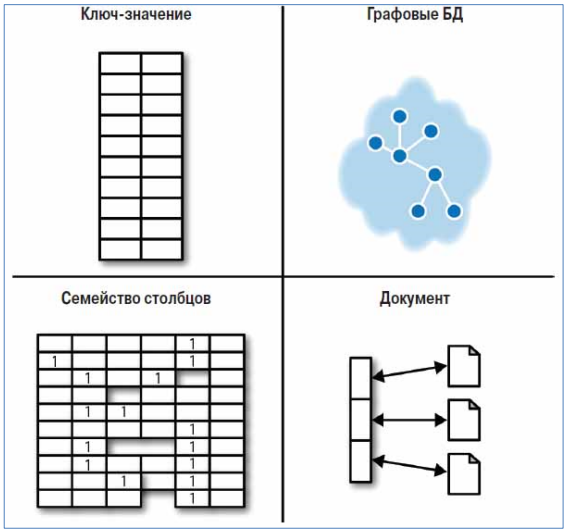


*Рис. 1 Секторы NoSQL-хранилищ [6]*
*Fic.1 NoSQL storage sectors [6]*

- *Ключ-значение* – такой тип нереляционной базы данных будет полезен, когда необходимо получить объект по идентификатору. Например, это может быть кеширование данных – с помощью уникального ключа можно быстро получать любые данные. Реализацией такого подхода может служить, например, база данных Redis [7].





- *Графовые системы* – данная структура будет полезна, если в системе используются связи между объектами одной коллекции, например, для получения корректного трека *n* точек GPS точек должны идти последовательно. Примером является база данных Arango [8].
- *Семейство столбцов* – отлично походит для метрик, когда есть большой поток данных от сервиса, а отображать его необходимо на разных графиках, удобно делать выборку по столбцам и отображать необходимые данные. Пример – ClickHouse [9].
- *Документ* – данный тип нереляционной базы данных позволяет хранить динамическую структуру значения. Такой тип базы данных будет полезен для структур данных, в которых невозможно предсказать полный набор полей. Пример – MongoDB [10].

Для гарантии целостности данных подойдут реляционные базы данных, такие как PostgreSQL, MS SQL Server, MySQL и другие. Основное, на что необходимо обратить внимание, – индексирование таблиц для быстрого доступа к данным по их уникальным идентификаторам. Необходимо позаботится о шардировании базы данных для повышения отказоустойчивости и о репликации для гарантии сохранности данных. При выборе реляционной базы данных также следует учитывать их особенности; например, база данных PostgreSQL [11] позволяет хранить тип данных JSONB – двоичную разновидность формата JSON, у которой пробелы удаляются, сортировка объектов не сохраняется, вместо этого они хранятся оптимальным образом и сохраняется только последнее значение для ключей-дубликатов.

## 4. Паттерн DDD

При разработке сервисов нового приложения часто придерживаются монолитной архитектуры, при которой один сервис обрабатывает практически любые задачи – чтение и запись в базу данных, бизнес-логика. Такой поход имеет свои преимущества при старте проекта – экономится время и ресурсы на первый запуск приложения, но развитие такой системы затруднено из-за невозможности масштабирования. Для решения этой проблемы разработана микросервисная архитектура, которая позволяет разделить монолит на небольшие отдельные компоненты. На этапе проектирования микросервисов появляется проблема разграничения логики каждого сервиса, так как если определить неправильный контекст сервиса, то выигрыша от такого подхода не будет. Наиболее эффективным способом является разграничение области ответственности сервиса в соответствии с бизнес-моделью; например, один сервис обрабатывает все потоки данных, которые касаются персональных данных, другой поддерживает транзакции пользователей и т.д. Такой подход описан в наборе принципов Domain Driven Design (DDD) [12].

Основными принципами, которыми оперирует DDD являются следующие.

1) *Ограниченные связи.* Так как сервис работает в ограниченном контексте, то и связей будет минимальное количество.
2) *Целостность.* При любом исходе в базе данных не останется несогласованных данных. В случае, если происходит ошибка распределенных транзакций, и данные портятся, проблема решается на другом уровне.
3) *Взаимосвязь.* При проектировании системы по методу DDD получится четкая структура, в которой каждый отдельный компонент имеет связанные, но независимые элементы. Это значит, что данные в одном сервисе зависят от данных в другом сервисе, например, пользователь и его счет, но изменение одной из сущностей не потребует изменения другой.

Для максимальной эффективности такого подхода базы данных должны быть также изолированы друг от друга – одна база на один сервис. Такое решение позволит сократить время на восстановление базы данных и снизить нагрузку на сервер.

## 5. Взаимодействие компонентов и паттерн CQRS

При реализации приложения с архитектурой по DDD паттерну приводит к большому количеству сетевых взаимодействий между компонентами системы, поэтому стоит выбрать способы взаимодействия каждого компонента системы.

Выбор способа взаимодействия между компонентами зависит от нескольких факторов:

1) количество и частота передаваемых данных;
2) ответная реакция на вызов компонента.

В первом случае, если необходимо передать/получать большое количество данных, или если вызов компонента осуществляется часто, то стоит выбрать способ непрерывного взаимодействия – *socket*. При таком подходе открывается непрерывное взаимодействие между двумя компонентами на серверной части приложения или клиент-серверном взаимодействии, и осуществляется непрерывная передача данных. При реализации взаимодействия через *socket* не требуется ответная реакция вызываемого компонента. Один из недостатков данного решения состоит в том, что поскольку под каждое соединение открывается свой порт, количество соединений для одного серверного компонента ограничено величиной 65535, но при таком подходе сервис может не успевать обрабатывать все входящие данные, поэтому стоит ориентироваться на сложность обрабатываемых задач.

В случае, когда взаимодействий не так много и передаваемых данных меньше, можно использовать более простую синхронную реализацию взаимодействия компонентов и клиента с серверов – *rest*.

При проектировании архитектуры системы, если ответ не нужен сразу или необходимо провести изменения в большом количестве компонентов, то стоит использовать событийную модель – отправлять событие в сервер очередей, используя, например, *kafka* [13] и другие сервисы. Слушатели должны подписаться на это событие и провести необходимые операции. Основное преимущество данного подхода – минимизация нагрузки на систему, так как слушатель прочитает столько сообщений, сколько может обработать, возможность провести изменения в большом количестве компонентов практически одновременно. Основной недостаток такого подхода – невозможна простая реализация, при которой можно вернуть ответ клиенту, возникают дополнительные точки отказа и появляется нагрузка на поддержку сервера очередей.

Для получения результата обработки события клиентом следует решить следующую проблему: компонент, который получил данные, занят их обработкой, и дополнительные запросы на выдачу информации приведут к созданию дополнительных соединений, лишней нагрузки на компонент и базу данных. Поэтому процессы добавления данных и их выдачи необходимо разделить – создать разные компоненты, один из которых будет принимать данные от клиента, а второй их отдавать.

Соответствующий паттерн проектирования называется CQRS (Command and Query Responsibility Segregation, разделение ответственности команд и данных) [14]. Этот паттерн принято кратко характеризовать следующим образом: метод должен быть либо командой, выполняющей какое-то действие, либо запросом, возвращающим данные, но не одновременно тем и другим. В основе CQRS лежит принцип CQS (Command and Query Segregation, разделение команд и запросов), введенный Бертраном Мейером (Bertran Meyer) в его книге [15].

Принцип императивного программирования позволяет снизить нагрузку на базу данных и разделить потоки чтения и записи данных. В результате система гарантирует быстрый ответ пользователю на получение данных, а запись их не затрагивает работу пользователя за счет выполнения операций чтения и записи над разными базами данных (рис. 2).





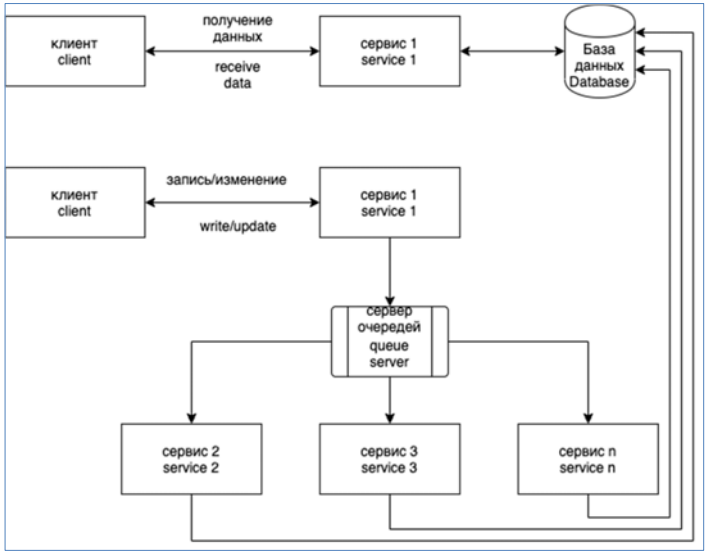


*Рис. 2. Схема взаимодействия сервисов в высоконагруженной системе*
*Fig. 2. Service interaction scheme in a highly loaded system*

## 6. Примеры практического применения паттернов

Описанные в статье способы применены во многих проектах, например, МТС ПОИСК [16], открытие нового счета в Альфа-Банке и т.д.

Приложение МТС Поиск – сервис для поиска людей, основанных на геоданных. Трек пользователя представляет из себя набор связанных точек – граф, поэтому применение графовой нереляционной базы данных позволило хранить маршруты пользователя в базе данных и быстро получать его. Для этого решения была выбрана графовая NoSQL база данных Arango.

Получение в приложении некоторых сущностей, которые меняются редко, занимало много времени, поэтому была интегрирована нереляционная база данных «ключ-значение» Redis для хранения готовых объектов. Время получения сущностей по уникальным идентификаторам удалось сократить в разы, также удалось уменьшить количество сетевых взаимодействий, нагрузку на базу данных и другие компоненты системы, которые задействованы в процессе сбора данных.

В проекте Новый счет для юридических лиц в компании Альфа-Банк необходимо было обрабатывать большое количество файлов и разное количество метаданных, связанных с каждым из них, поэтому было выбрано NoSQL решение MongoDB. В результате удалось решить проблему хранения файлов и всей связанной с ними информации, хранить разные форматы данных, так как в некоторых случаях метаданных может быть значительно больше. За счет шардирования удалось легко масштабировать базу данных и получить хорошую производительность.

Паттерн CQRS применялся в приложении МТС Поиск. Получение всех новых данных выведено в отдельные процессы, которые работают отдельно от компонентов, которые выдают информацию пользователю. За счет этого получилось упростить разработку приложения.

### *7. Заключение*

При проектировании высоконагруженной отказоустойчивой системы с возможностью как вертикального, так и горизонтального масштабирования следует разделять чтение и запись в системе, разрабатывая архитектуру по паттерну CQRS и соблюдая правила разделения сервисов по паттерну DDD. Требуется выбрать подходящую базу данных, возможно, даже несколько, так как сервисы получатся логически независимыми друг от друга, а для их быстрой совместной работы следует стоит выбрать правильный тип их взаимодействия.

## Список литературы / References


[1]. High-load Systems. URL: https://flyoutsourcing.com/high-load-systems.html, accessed 17.06.2020.
[2]. Игумнов А.О., Сонькин Д.М. Об одном из подходов к оптимизации высоконагруженных систем на примере системы диспетчерского управления таксомоторным парком. Науковедение, 2013 г., №2(15), 7 стр. / Igumnov A.O., Sonkin D.M. One approache to optimize highload systems on example of dispatching taxi system. Naukovedenie, №2(15), 7 p. (in Russian).
[3]. Rajković P., Janković D., Milenković A. Using cqrs pattern for improving performances in medical information systems. In Proc. of the 6th Balkan Conference in Informatics, 2013, pp. 86-91.
[4]. Pacheco V. F. Microservice Patterns and Best Practices: Explore patterns like CQRS and event sourcing to create scalable, maintainable, and testable microservices. Packt Publishing, 2018, 366 p.
[5]. What are NoSQL databases? URL: https://aws.amazon.com/ru/nosql/, accessed 17.06.2020.
[6]. Обзор NOSQL баз данных / NOSQL Database Overview. URL: https://oracle-patches.com/db/3688-обзор-nosql-баз-данных, accessed 05.09.2020 (in Russian).
[7]. Redis. URL: https://redis.io/, accessed 28.10.2020.
[8]. ArangoDb, URL: https://www.arangodb.com/, accessed 28.10.2020.
[9]. What Is ClickHouse? URL: https://clickhouse.tech/docs/en/, accessed 28.10.2020.
[10]. MongoDb. URL: https://www.mongodb.com/, accessed 17.07.2020.
[11]. Postgres Professional. URL: https://habr.com/ru/company/postgrespro/blog/458186/, accessed 06.09.2020 (in Russian).
[12]. Design a DDD-oriented microservice. URL: https://docs.microsoft.com/ru-ru/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/ddd-oriented-microservice, accessed 06.09.2020.
[13]. Apache Kafka. URL: https://kafka.apache.org/, accessed 28.10.2020.
[14]. Pattern: Command Query Responsibility Segregation (CQRS). URL: https://microservices.io/patterns/data/cqrs.html, accessed 25.06.2020.
[15]. Meyer B. Object-Oriented Software Construction. Prentice Hall, 1994, 534 p.
[16]. МТС ПОИСК / MTS Search. URL: https://poisk.mts.ru/, accessed 25.06.2020 (in Russian).


## Информация об авторе / Information about the author


Василий Андреевич РУДОМЕТКИН – аспирант в области вычислительной техники и информатики, эксперт в области разработки высоконагруженных систем, сервисов геолокации, банковских систем. Сфера научных интересов: разработка и тестирование высоконагруженных систем.

Vasilii Andreevich RUDOMETKIN – postgraduate student in the field of computer technology and informatics, an expert in the development of high-load systems, geolocation services, banking systems. Research interests: development and testing of high-load systems.