DOI 10.37882/2223-2966.2023.7-2.06

ОПТИМИЗАЦИЯ ПЕРЕДАЧИ ДАННЫХ В АРХИТЕКТУРЕ ВЕБ-СЕРВИСОВ

OPTIMIZATION OF DATA TRANSFER IN WEB SERVICES ARCHITECTURE

E. Vakhromeeva
I. Trifonov

Summary: This scientific article explores the optimization of transmitted data in a client-server architecture in order to increase service availability and improve system performance. As a result of the optimization carried out, it was possible to reduce the amount of transmitted data by almost five times, which is a significant achievement. This has a positive impact on the availability of the service, especially in conditions of limited mobile network bandwidth or outdated equipment of providers. It is important to note that the size of the data stored on the server remains virtually unchanged.

The use of separating data into «hot» and «cold» categories allows the server to work more efficiently, providing high performance and system reliability. This ensures that the functions of the application perform efficiently and reduce the risk of data loss. The authors of the article emphasize the need to take into account the specifics of each system when choosing optimization methods and data storage mechanisms. Regular monitoring and analysis of application performance are essential components to achieve the best results in optimizing the size of the transferred data.

Keywords: client-server architecture, data reduction, data sharing, performance monitoring, data optimization methods.

Вахромеева Екатерина Николаевна
канд. техн. наук, доцент, Российский государственный университет им. А.Н. Косыгина (Технологии. Дизайн. Искусство), г. Москва
9287701@mail.ru
Трифонов Илья Александрович
Российский государственный университет им. А.Н. Косыгина (Технологии. Дизайн. Искусство), г. Москва
ilya.trifonov.official@vk.com

Аннотация. Данная научная статья исследует оптимизацию передаваемых данных в клиент-серверной архитектуре с целью повышения доступности сервиса и улучшения производительности системы. В результате проведенной оптимизации удалось сократить объем передаваемых данных практически в пять раз, что представляет значительное достижение. Это оказывает положительное влияние на доступность сервиса, особенно в условиях ограниченной пропускной способности мобильных сетей или устаревшего оборудования провайдеров. Важно отметить, что размер данных, хранимых на сервере, практически не изменяется.

Применение разделения данных на «горячие» и «холодные» категории позволяет серверу работать более эффективно, обеспечивая высокую производительность и надежность системы. Это обеспечивает эффективное выполнение функций приложения и снижение рисков потери данных. Авторы статьи подчеркивают необходимость учета специфики каждой системы при выборе оптимизационных методов и механизмов хранения данных. Регулярный мониторинг и анализ производительности приложения являются важными компонентами для достижения наилучших результатов в оптимизации размера передаваемых данных.

Ключевые слова: клиент-серверная архитектура, сокращение объема данных, разделение данных, мониторинг производительности, методы оптимизации данных.

С развитием веб-приложений и увеличением объема передаваемых данных в сети, оптимизация процесса передачи становится важным аспектом для обеспечения эффективной работы веб-приложений. Один из ключевых факторов, влияющих на производительность и пользовательское взаимодействие с веб-приложениями, заключается в оптимизации размера передаваемых данных между клиентом и сервером. В современном информационном обществе, множество приложений, с которыми мы взаимодействуем, основаны на клиент-серверной архитектуре. Эта модель взаимодействия между компьютерными приложениями разделяет систему на две ключевые компоненты: клиентскую и серверную. Клиентское приложение функционирует на устройстве пользователя, а, в свою очередь, серверное приложение, как понятно из названия, располагается на сервере.

Процесс взаимодействия начинается с пользовательского взаимодействия с клиентским приложением, которое отправляет запросы на сервер для получения данных или выполнения определенных операций. Сервер обрабатывает эти запросы и отвечает, предоставляя запрошенную информацию или результаты операций, выполненных на сервере. Таким образом, клиентское приложение обращается к серверу с целью получения данных и осуществления функций, в то время как сервер отвечает за обработку запросов и хранение данных.

Такое взаимодействие между клиентом и сервером обеспечивает эффективную работу приложений, позволяя пользователю получать необходимую информацию и выполнять требуемые задачи, что повышает производительность.

Основное преимущество клиент-серверной архитектуры заключается в возможности распределения задач между клиентом и сервером. Клиент выполняет только то, что связано с интерфейсом пользователя и взаимодействием с пользователем, а все остальные задачи по обработке данных, хранению и передаче выполняются на сервере. Такой подход позволяет разделить нагрузку между двумя компонентами, сократить время отклика системы и повысить ее надежность.

Кроме того, клиент-серверная архитектура обеспечивает масштабируемость. Надежность приложения также повышается, поскольку если один из серверов перестает работать, клиенты могут переключиться на другой сервер. В результате, все эти преимущества делают клиент-серверную архитектуру привлекательным выбором при разработке приложений, обеспечивая эффективное распределение задач, масштабируемость и надежность системы.

В рамках клиент-серверной архитектуры, взаимодействие между клиентами и серверами происходит посредством разнообразных протоколов. Эффективное управление объемом передаваемых данных является важной задачей, требующей тщательного изучения при разработке таких приложений. Корректное управление объемом передачи данных играет ключевую роль в обеспечении высокой производительности и качества работы приложения, обеспечивая быструю загрузку страниц и плавное взаимодействие с пользователями. Это также позволяет сократить использование пропускной способности сети и затраты на хранение данных. Контроль объема данных и оптимизация способствуют безопасной передаче информации и эффективному использованию ресурсов сервера, таких как процессорное время и память, улучшая обработку запросов и общую производительность системы.

При выборе методов оптимизации и механизмов хранения данных важно учитывать особенности конкретной системы. Различные факторы, такие как объем и частота запросов, характер данных, доступные ресурсы и требования к отзывчивости, должны быть учтены при применении современных методов оптимизации, соответствующих потребностям системы.

Оптимизация размера передаваемых данных в клиент-серверной архитектуре является актуальной проблемой, которая может быть решена с помощью различных методов и технологий. Один из эффективных подходов включает разделение данных на «горячие» и «холодные» категории. Горячие данные, которые часто запрашиваются и активно изменяются, могут быть сохранены в быстродействующих кэшах на сервере для быстрого доступа клиентского приложения. Холодные данные, с другой стороны, могут быть сохранены в базе данных и запрашиваться по мере необходимости. Такой подход помогает сократить объем передаваемых данных, уменьшить нагрузку на сервер и повысить общую производительность и отзывчивость системы.

Решение о классификации данных как горячих или холодных требует тщательного анализа и учета особенностей конкретной системы, включая частоту запросов и изменений данных, а также их влияние на функциональность и взаимодействие с пользователем. Использование соответствующих механизмов хранения данных, основанных на классификации, играет важную роль в обеспечении эффективного функционирования системы.

Однако, следует отметить, что процесс определения горячих и холодных данных является динамическим и может изменяться в зависимости от изменений в поведении пользователей и требованиях системы. Поэтому регулярный мониторинг и анализ производительности приложения позволяют добиться наилучших результатов в оптимизации размера передаваемых данных.

Итак, оптимизация размера передаваемых данных в клиент-серверной архитектуре с помощью классификации на горячие и холодные данные является эффективным подходом, позволяющим снизить объем передачи информации, улучшить производительность системы и повысить отзывчивость веб-приложений.

Одним из таких методов является сжатие данных, которое позволяет уменьшить их размер перед передачей. Сжатие может быть применено к различным типам данных, таким как текст, изображения, аудио или видео. Это позволяет сократить объем передаваемых данных, уменьшить время передачи и снизить нагрузку на сеть.

Другим важным аспектом оптимизации размера передаваемых данных является минимизация избыточности информации. При разработке протоколов и форматов передачи данных следует избегать повторяющейся информации или излишней структуры, которая может увеличить объем передаваемых данных. Использование компактных форматов данных, таких как JSON или Protocol Buffers, может значительно сократить объем передаваемой информации.

Кроме того, оптимизацию можно достичь путем кэширования данных на клиентской стороне. Это позволяет избежать повторных запросов к серверу за одними и теми же данными, если они не изменились. Кэширование позволяет снизить объем сетевого трафика и ускорить отклик системы.

Также стоит учитывать влияние сетевых условий на производительность и передачу данных. Использование сжатия данных, оптимизация протоколов связи

и выбор эффективных алгоритмов передачи данных могут помочь справиться с проблемами низкой пропускной способности сети и снизить задержки при передаче данных.

Наконец, постоянное исследование и применение новых методов оптимизации данных в клиент-серверной архитектуре позволяет достичь лучших результатов и улучшить производительность системы. Развитие технологий и появление новых инструментов позволяют решать проблемы передачи данных более эффективно.

Объектом исследования является создание серверной архитектуры клиент-серверного приложения почтового веб-сервиса с реализацией разделения данных на горячие и холодные.

В ходе разработки сервера была использована среда выполнения Node.js, которая является открытой и кроссплатформенной, предоставляющей возможность выполнения JavaScript на стороне сервера. Node.js базируется на мощном JavaScript-движке «V8», который применяется в браузере Google Chrome. Однако, в отличие от браузера, Node.js обладает расширенными возможностями доступа к системным ресурсам, таким как файловая система, сеть и оперативная память, что позволяет создавать высокопроизводительные и масштабируемые серверные приложения.

Node.js обладает несколькими существенными преимуществами по сравнению с другими технологиями серверной разработки. Одно из ключевых преимуществ состоит в его способности обрабатывать запросы асинхронно, что позволяет эффективно обрабатывать большое количество запросов одновременно без блокировки исполнения потоков. Это достигается с использованием механизмов обратных вызовов или асинхронных функций.

Это асинхронное выполнение запросов в Node.js основывается на событийно-ориентированной архитектуре, где операции выполняются неблокирующим образом и их завершение сопровождается вызовом соответствующего обратного вызова. Такой подход позволяет эффективно использовать ресурсы сервера и снизить задержки в обработке запросов.

Кроме того, Node.js предоставляет разработчикам богатый набор модулей и библиотек, которые упрощают процесс разработки и расширяют функциональность серверных приложений.

При постановке задачи было принято решение ввода тестовых данных на сервер посредством передачи ему JSON-файла. Формат JSON используется для передачи структурированных данных между клиентом и сервером в сети. Он обладает читаемостью как для человека, так и для компьютера, кроме этого, позволяет элегантно представлять сложные иерархические структуры данных.

В качестве тестовых данных для почтового клиента был использован массив писем (объектов) в виде JSON-файла. Каждое из писем содержит в себе данные об авторе, получателях, заголовок, текст письма и некоторые другие служебные параметры. Наиболее популярным запросом к такому файлу может быть запрос списка писем, в котором не отображаются получатели и текст письма целиком. Полные же данные о письме запрашиваются только когда пользователь открывает определённое письмо.

Для обработки тестовых данных на сервере требуется прочитать исходный JSON-файл и создать переменный для хранения данных. Исходные данные из JSON-файла сохраняются в переменную hotData, переменная coldData является пустым объектом.

JavaScript-код:

```
const rawData = fs.readFileSync('db.json');
let hotData = JSON.parse(rawData);
let coldData = {};
```

На следующем этапе требуется выполнить индексирование писем и разделить данные на две переменные. В соответствии с целью задачи было решено выделить полный текст письма и список адресатов в холодные данные, поскольку они не являются необходимыми для частых операций. В горячих данных остается только начало текста письма, ограниченное до 100 символов, чтобы отобразить его в общем списке.

JavaScript-код:

```
hotData.forEach((letter, index) => {
 letter.id = index;
 coldData[index] = {
 to: letter.to,
 text: letter.text
 }
 letter.text = letter.text.slice(0, 100);
 letter.to = null
});
```

Для обеспечения целостности данных и явного способа их получения в качестве ключа к объекту с данными переменной coldData был выбран идентификатор письма, что упрощает поиск и убирает необходимость сортировки данных в переменной coldData.

В ходе исследования были проведены измерения объема передаваемых данных до и после применения оптимизации с использованием инструмента Chrome

DevTools. Chrome DevTools представляет собой набор инструментов для разработки и отладки веб-приложений, который поставляется с браузером Google Chrome. Эти инструменты предоставляют разработчикам доступ к содержимому веб-страниц и обладают мощными возможностями для отладки и анализа производительности веб-приложений. В ходе исследования мы изучили изменения в объеме передаваемых данных до и после применения оптимизации, результаты которых показаны на рис. 1–2.

Для работы с тестовыми данными на сервере сначала необходимо прочитать исходный JSON-файл и создать переменные для хранения данных. Изначально все данные будут занесены в переменную hotData, а coldData будет проинициализировано пустым объектом.

JavaScript-код, выполняющий данные операции:

```
const rawData = fs.readFileSync('db.json');
let hotData = JSON.parse(rawData);
let coldData = {};
```

Следующим шагом необходимо выполнить индексирование писем и разделение данных в две переменные. Исходя из поставленной задачи, было принято решение вынести в холодные данные полный текст письма и список адресатов, так как они не являются необходимыми при частой операции — получении списка писем. В горячих данных остался текст письма до 100 символов, чтобы отобразить начало письма в общем списке. Помимо этого, во время индексирования происходит проверка того, имеются ли у письма вложения и в случае их наличия происходит сохранение вложенного изображения на сервере, а в данных сохраняется ссылка на файл, по которому в интерфейсе будет загружаться изображение и отображаться пользователю.

Также проверяется список получателей письма и наличие у них изображений профиля, все обнаруженные аватары (изображения профиля) сохраняются на сервере и при передаче данных отправляется только ссылка на это изображение, что значительно уменьшает количество передаваемых данных.

JavaScript-код, реализующий данные манипуляции:

```
hotData.forEach((letter, index) => {
letter.id = index;
if (letter.hasOwnProperty('doc')) {
```

Name	Status	Type	Initiator	Size
inbox?limit=20&offset=0&isUnre...	200	fetch	MailService.js?...	59.8 kB
inbox?limit=20&offset=20&isUnr...	200	fetch	MailService.js?...	68.9 kB
inbox?limit=20&offset=40&isUnr...	200	fetch	MailService.js?...	56.8 kB
inbox?limit=20&offset=60&isUnr...	200	fetch	MailService.js?...	59.3 kB
inbox?limit=20&offset=80&isUnr...	200	fetch	MailService.js?...	59.2 kB
inbox?limit=20&offset=100&isUn...	200	fetch	MailService.js?...	63.2 kB
inbox?limit=20&offset=120&isUn...	200	fetch	MailService.js?...	65.8 kB
inbox?limit=20&offset=140&isUn...	200	fetch	MailService.js?...	25.0 kB

Рис. 1. Измерения размера передаваемых данных до проведения оптимизации

Name	Status	Type	Initiator	Size
inbox?limit=20&offset=0&isUnre...	200	fetch	MailService.js?...	12.0 kB
inbox?limit=20&offset=20&isUnr...	200	fetch	MailService.js?...	11.9 kB
inbox?limit=20&offset=40&isUnr...	200	fetch	MailService.js?...	12.3 kB
inbox?limit=20&offset=60&isUnr...	200	fetch	MailService.js?...	11.0 kB
inbox?limit=20&offset=80&isUnr...	200	fetch	MailService.js?...	12.0 kB
inbox?limit=20&offset=100&isUn...	200	fetch	MailService.js?...	11.9 kB
inbox?limit=20&offset=120&isUn...	200	fetch	MailService.js?...	11.2 kB
inbox?limit=20&offset=140&isUn...	200	fetch	MailService.js?...	5.6 kB

Рис. 2. Измерения размера передаваемых данных после проведения оптимизации

```
const docImg = letter.doc.img;
if (Array.isArray(docImg)) {
const imagesUrls = [];
docImg.forEach(image => {
imagesUrls.push(makePictureAndUrl(image, pictureTypes.attachments));
});
letter.doc.img = imagesUrls;
} else {
letter.doc.img = makePictureAndUrl(docImg, pictureTypes.attachments);
}
}
if (letter.hasOwnProperty('author')
&& letter.author.hasOwnProperty('avatar')) {
letter.author.avatar = makePictureAndUrl(letter.author.avatar, pictureTypes.avatars);
}
if (letter.hasOwnProperty('to')) {
letter.to.forEach(person => {
if (person.hasOwnProperty('avatar')) {
person.avatar = makePictureAndUrl(person.avatar, pictureTypes.avatars);
}
});
}
coldData[index] = {
to: letter.to,
text: letter.text
}
letter.text = letter.text.slice(0, 100);
letter.to = null
});
hotData.sort(sortFunctionByDate).reverse();
```

В результате оптимизации, объем передаваемых данных был сокращен практически в пять раз, что является значительным достижением. Это улучшает доступность сервиса даже при ограниченной пропускной способности мобильных сетей или устаревшего оборудования у провайдеров. При этом размер данных, хранимых на сервере, почти не изменяется. Разделение данных позволяет серверу работать более эффективно, обеспечивая высокую производительность и надежность. Это, в свою очередь, позволяет приложению эффективно выполнять свои функции.

С полным кодом проекта можно ознакомиться в репозитории по ссылке https://github.com/IlyaTrifonov/vk-cup-s1.

ЛИТЕРАТУРА

1. Клиент-серверная архитектура | Введение в интернет: [Электронный ресурс]. URL: https://ru.hexlet.io/courses/internet-fundamentals/lessons/client-server/theory_unit (дата обращения: 16.02.2023).
2. URL: https://appmaster.io/ru/blog/masshtabiruemost-vazhna (дата обращения: 28.02.2023).
3. Протоколы Интернета и электронной почты: [Электронный ресурс]. URL: https://professorweb.ru/my/csharp/web/level1/1_6.php (дата обращения: 02.03.2023).
4. Documentation | Node.js: [Электронный ресурс]. URL: https://nodejs.org/en/docs (дата обращения: 03.03.2023).
5. Working with JSON — Learn web development | MDN: [Электронный ресурс]. URL: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON (дата обращения: 04.03.2023).
6. Chrome DevTools — Chrome Developers: [Электронный ресурс]. URL: https://developer.chrome.com/docs/devtools/ (дата обращения: 09.03.2023).

Журнал «Современная наука: актуальные проблемы теории и практики»