

Министерство науки и высшего образования Российской Федерации  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ  
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО  
ITMO University

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА  
GRADUATION THESIS

Разработка высоконагруженной системы синхронизации данных пользователей для  
аукциона рекламных объявлений

Обучающийся / Student Кириенко Никита Алексеевич

Факультет/институт/кластер/ Faculty/Institute/Cluster факультет программной  
инженерии и компьютерной техники

Группа/Group Р34101

Направление подготовки/ Subject area 09.03.04 Программная инженерия

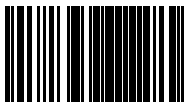
Образовательная программа / Educational program Системное и прикладное  
программное обеспечение 2020

Язык реализации ОП / Language of the educational program Русский

Квалификация/ Degree level Бакалавр

Руководитель ВКР/ Thesis supervisor Гаврилов Антон Валерьевич, Университет ИТМО,  
факультет программной инженерии и компьютерной техники, старший преподаватель  
(квалификационная категория "старший преподаватель")

Обучающийся/Student

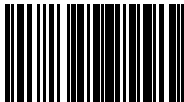
Документ подписан	
Кириенко Никита Алексеевич	
14.05.2024	

(эл. подпись/ signature)

Кириенко  
Никита  
Алексеевич

(Фамилия И.О./ name  
and surname)

Руководитель ВКР/  
Thesis supervisor

Документ подписан	
Гаврилов Антон Валерьевич	
14.05.2024	

(эл. подпись/ signature)

Гаврилов Антон  
Валерьевич

(Фамилия И.О./ name  
and surname)

**Министерство науки и высшего образования Российской Федерации  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ  
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО  
ITMO University**

**ЗАДАНИЕ НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ /  
OBJECTIVES FOR A GRADUATION THESIS**

**Обучающийся / Student** Кириенко Никита Алексеевич  
**Факультет/институт/кластер/ Faculty/Institute/Cluster** факультет программной инженерии и компьютерной техники  
**Группа/Group** P34101  
**Направление подготовки/ Subject area** 09.03.04 Программная инженерия  
**Образовательная программа / Educational program** Системное и прикладное программное обеспечение 2020  
**Язык реализации ОП / Language of the educational program** Русский  
**Квалификация/ Degree level** Бакалавр  
**Тема ВКР/ Thesis topic** Разработка высоконагруженной системы синхронизации данных пользователей для аукциона рекламных объявлений  
**Руководитель ВКР/ Thesis supervisor** Гаврилов Антон Валерьевич, Университет ИТМО, факультет программной инженерии и компьютерной техники, старший преподаватель (квалификационная категория "старший преподаватель")

**Характеристика темы ВКР / Description of thesis subject (topic)**

**Тема в области фундаментальных исследований / Subject of fundamental research:** нет / not

**Тема в области прикладных исследований / Subject of applied research:** да / yes

**Основные вопросы, подлежащие разработке / Key issues to be analyzed**

Техническое задание:

Разработать высоконагруженную систему доставки данных, используемых для аукциона рекламных объявлений, на большое количество конечных машин.

Функции разрабатываемого решения:

1. Обработка входящих HTTP-запросов заданного формата;
2. Доставка полученных в запросе данных на конечные машины;
3. Сохранение данных в конечную базу данных.

Цель:

Обеспечение требуемых времени доставки сообщений и пропускной способности.

Задачи:

1. Исследовать существующие решения, реализующие функционал доставки сообщений;
2. Выделить их преимущества и недостатки;
3. Реализовать собственное решение, соответствующее перечисленным требованиям;

4. Провести анализ, профилирование и тестирование первоначальной реализации решения, исследовать возможности по улучшению производительности;
5. По возможности улучшить решение, затем снова провести анализ, сравнив с первоначальной реализацией.

Содержание работы:

- Введение;
- Обзор предметной области и аналогов;
- Описание предложенного решения;
- Практическая реализация;
- Анализ результатов;
- Заключение.

**Форма представления материалов ВКР / Format(s) of thesis materials:**


Пояснительная записка, программный код, презентация

**Дата выдачи задания / Assignment issued on:** 02.10.2023

**Срок представления готовой ВКР / Deadline for final edition of the thesis** 28.05.2024

**СОГЛАСОВАНО / AGREED:**

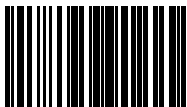
Руководитель ВКР/  
Thesis supervisor

Документ подписан	
Гаврилов Антон Валерьевич	
03.05.2024	

(эл. подпись)

Гаврилов Антон  
Валерьевич

Задание принял к  
исполнению/ Objectives  
assumed BY

Документ подписан	
Кириенко Никита Алексеевич	
03.05.2024	

(эл. подпись)

Кириенко  
Никита  
Алексеевич

Руководитель ОП/ Head  
of educational program

(эл. подпись)

**Министерство науки и высшего образования Российской Федерации  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ  
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО  
ITMO University**

**АННОТАЦИЯ  
ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ  
SUMMARY OF A GRADUATION THESIS**

**Обучающийся / Student** Кириенко Никита Алексеевич  
**Факультет/институт/кластер/ Faculty/Institute/Cluster** факультет программной инженерии и компьютерной техники  
**Группа/Group** P34101  
**Направление подготовки/ Subject area** 09.03.04 Программная инженерия  
**Образовательная программа / Educational program** Системное и прикладное программное обеспечение 2020  
**Язык реализации ОП / Language of the educational program** Русский  
**Квалификация/ Degree level** Бакалавр  
**Тема ВКР/ Thesis topic** Разработка высоконагруженной системы синхронизации данных пользователей для аукциона рекламных объявлений  
**Руководитель ВКР/ Thesis supervisor** Гаврилов Антон Валерьевич, Университет ИТМО, факультет программной инженерии и компьютерной техники, старший преподаватель (квалификационная категория "старший преподаватель")

**ХАРАКТЕРИСТИКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ  
DESCRIPTION OF THE GRADUATION THESIS**

**Цель исследования / Research goal**

Обеспечение требуемых времени доставки сообщений и пропускной способности.

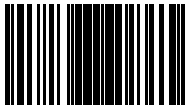
**Задачи, решаемые в ВКР / Research tasks**

1. Исследовать существующие решения, реализующие функциональность доставки сообщений; 2. Выделить их преимущества и недостатки; 3. Реализовать собственное решение, соответствующее перечисленным требованиям; 4. Провести анализ, профилирование и тестирование первоначальной реализации решения, исследовать возможности по улучшению производительности; 5. По возможности улучшить решение, затем снова провести анализ, сравнив с первоначальной реализацией.

**Краткая характеристика полученных результатов / Short summary of results/findings**

Исследованы существующие решения, реализующие функциональность доставки сообщений, выделены их преимущества и недостатки. Сделан вывод, что существующие решения не подходят для решения поставленной задачи, и обоснована необходимость реализации собственного решения. Был спроектирован и реализован первый прототип решения, после чего были проведены тестирование и профилирование этого прототипа. На основе результатов анализа полученных при тестировании данных прототип был улучшен и приведен к финальной реализации, для которой было проведено финальное тестирование, подтверждающее соответствие всем заявленным требованиям.

Обучающийся/Student

Документ подписан	
Кириенко Никита Алексеевич	
14.05.2024	

(эл. подпись/ signature)

Кириенко  
Никита  
Алексеевич

(Фамилия И.О./ name  
and surname)

Руководитель ВКР/  
Thesis supervisor

Документ подписан	
Гаврилов Антон Валерьевич	
14.05.2024	

(эл. подпись/ signature)

Гаврилов Антон  
Валерьевич

(Фамилия И.О./ name  
and surname)

## СОДЕРЖАНИЕ

СПИСОК СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ.....	8
ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ.....	9
ВВЕДЕНИЕ.....	10
1 Обзор предметной области и аналогов.....	12
1.1 Введение в предметную область.....	12
1.2 Предыдущие решения.....	13
1.3 Общий вид разрабатываемой системы.....	18
1.4 Аналоги и существующие решения.....	19
1.4.1 RabbitMQ.....	19
1.4.2 Kafka.....	21
1.4.3 NATS.....	22
1.4.4 Сравнение существующих решений в таблице.....	24
2 Описание предложенного решения.....	26
2.1 Общее описание.....	26
2.2 Язык программирования.....	27
2.3 HTTP-сервер.....	27
2.4 Протокол передачи данных на consumer.....	28
2.5 Внутренняя очередь.....	30
2.6 Прочие инструменты.....	31
3 Практическая реализация.....	33
3.1 Общая схема.....	33
3.2 Спецификация protobuf, gRPC и генерация кода.....	35
3.3 Метрики.....	37
3.4 Реализация компонента producer.....	39
3.4.1 Создание структуры Cookie.....	40
3.4.2 Разработка модуля Handler.....	40
3.4.3 Модуль Exchange.....	42
3.4.4 Разработка модуля Streamer.....	45
3.4.5 Создание Dockerfile и другие особенности.....	47
3.5 Реализация компонента consumer.....	49
3.5.1 Разработка модуля Consumer.....	49
3.5.3 Разработка модуля Storage.....	50
4 Тестирование и профилирование.....	54

4.1 Методика тестирования и профилирования.....	54
4.2 Тестирование передачи по сети.....	58
4.3 Тестирование записи в базу данных.....	61
4.4 Выводы.....	70
ЗАКЛЮЧЕНИЕ.....	72
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ.....	74

## **СПИСОК СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ**

AMQP – Advanced Message Queuing Protocol

DSP – Demand-Side Platform

HTTP – Hypertext Transfer Protocol

RPS – Requests Per Second

SSP – Supply-Side Platform

TTL – Time To Live

БД – База Данных



## **ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ**

Cookie (куки) – в контексте рекламных аукционов (OpenRTB) и данной ВКР: набор пользовательских данных.

DSP – Demand-Side Platform, платформа стороны спроса, рекламодатель, желающий разместить свое рекламное объявление.

OpenRTB 2.0 – протокол, позволяющий автоматизировать торговлю рекламной в интернете: взаимодействие стороны спроса (рекламодателей) со стороной предложения (издателей).

SSP – Supply-Side Platform, платформа стороны предложения, издатель рекламных объявлений, предоставляемых рекламодателями.

Брокер сообщений – приложение-посредник, принимающее сообщения от источника и передающее их потребителю.

Горутина – в контексте языка программирования Go: легковесный поток, обеспечивающий конкурентное выполнение кода с минимальными накладными расходами по сравнению с потоками операционной системы.

Мьютекс – примитив синхронизации, обеспечивающий взаимное исключение выполнения критических участков кода.

Персистентность – свойство или способность приложения сохранять данные и состояние между сеансами работы.

Репликация – механизм синхронизации содержимого нескольких экземпляров базы данных.

## ВВЕДЕНИЕ

Аукцион рекламных объявлений, использующий протокол OpenRTB – один из самых широко используемых механизмов для организации покупки и продажи рекламы, размещаемой в интернете. Разработка инструментов в соответствии с протоколом позволяет получить выгоду от совместимости с ним.

Главными элементами аукциона являются DSP (платформа продажи) и SSP (платформа покупки). Когда SSP встречает нового пользователя, она отдает его на известные DSP, которые регистрируют его у себя, назначают уникальный идентификатор, а затем передают обратно SSP в виде структуры, содержащей параметры для идентификации пользователей в связке SSP-DSP. Эта операция должна проходить за достаточно короткое время, поскольку к моменту получения следующего запроса на показ рекламы на всех SSP в кластере уже должен присутствовать эти параметры.

Обеспечение своевременной доставки данных на SSP – нетривиальная задача, которая требует либо тонкой настройки существующих решений, либо написания собственного механизма.

Кроме того, кластер SSP – быстрорастущая система, поэтому реализованное решение должно соответствовать современным стандартам разработки ПО в аспектах вертикальной и горизонтальной масштабируемости.

**Целью** работы является обеспечение заданных времени доставки сообщений и пропускной способности.

**Требования** к системе:

1. Обеспечение пропускной способности в 40 тыс. запросов в секунду;

2. Обеспечение работоспособности при пиковой нагрузке в 200 тыс. запросов в секунду;
3. Время доставки при пропускной способности 40 тыс. запросов в секунду – меньше 1 секунды;
4. Простота развертывания, расширения, конфигурации;
5. Минимальное количество инфраструктурных элементов;
6. Простота компонентов;
7. Быстрая скорость развертывания в кластере;
8. Гарантия сохранения данных при кратковременных потерях соединения.

**Задачи работы:**

1. Исследовать существующие решения, реализующие функционал доставки сообщений;
2. Выделить их преимущества и недостатки;
3. Реализовать собственное решение, соответствующее перечисленным требованиям;
4. Провести анализ, профилирование и тестирование первоначальной реализации решения, исследовать возможности по улучшению производительности;
5. По возможности улучшить решение, затем снова провести анализ, сравнив с первоначальной реализацией.

Разрабатываемое решение позволит упростить масштабирование кластера, что позволит быстрее увеличивать доходы компании.

# **1 Обзор предметной области и аналогов**

## **1.1 Введение в предметную область**

Протокол OpenRTB, используемый для проведения аукционов рекламных объявлений реального времени, содержит в себе идентификаторы id и buyerguid, которые используются для определения пользователей в рамках аукциона [1], что в свою очередь используется для более точного таргетинга, персонализации и подбора рекламы.

Типичный показ рекламы выглядит следующим образом: размещенный на сайте плеер делает запрос к SSP на получение списка рекламных объявлений. SSP обращается к OpenRTB-аукциону, где DSP делают ставки на показ рекламы. После этого SSP выбирает победившие рекламные объявления и показывает их пользователю.

Однако для того, чтобы точнее делать ставки на показ рекламы, DSP нужно знать, кому они эту рекламу показывают, то есть иметь уникальный идентификатор пользователя. Именно для этого существует механизм сопоставления идентификаторов, который называется сопоставлением или синхронизацией куки (cookie matching или cookie syncing) [2]. Стоит отметить, что в данном контексте термин “куки” не относится к HTTP-куки, а обозначает набор пользовательских данных. Ниже этот термин будет использоваться именно в этом значении, если не сказано иное.

В процессе сопоставления куки формируется связь между собственным идентификатором пользователя на стороне SSP с идентификатором пользователя у каждой DSP.

Сама синхронизация куки производится нетривиально по той причине, что веб-браузеры запрещают доменам читать куки других доменов. Из-за этого участники торгов размещают на сайтах специальные HTML-теги <img>, в которых содержится ссылка на DSP, куда и отправляются куки.

После этого DSP перенаправляет запрос уже к SSP, куда включает полученные куки пользователя, а также свой идентификатор. SSP сохраняет это соответствие, и в следующих запросах рекламы использует его для улучшения таргетинга.

Так как в рамках данной работы рассматривается сторона SSP, то идентификатор пользователя здесь определяется специфичным для компании способом – строкой из 20 символов, по совместительству являющейся и числом из 20 разрядов в шестнадцатеричной системе счисления, например “ae63a07d882b56e10b71”.

## **1.2 Предыдущие решения**

В компании MoeVideo кластер рекламной системы представляет из себя быстрорастущий набор экземпляров SSP (на данный момент – более 50 экземпляров) с входящим потоком около 20 тысяч запросов на синхронизацию куки в секунду в течение дня. Стоит отметить, что этот поток зависит от текущих партнеров компании (DSP), каждый из которых создает примерно 1 тыс. запросов в секунду. В разные моменты времени компания может работать или не работать с отдельными DSP, поэтому реальная нагрузка труднопредсказуема.

Нагрузка сильно варьируется в течение дня, и в пике может кратковременно достигать отметки от 100 до 200 тыс. запросов в секунду – в этом случае важно, чтобы система продолжала функционировать, но допускается увеличение времени доставки и записи в конечную базу данных.

Этот поток данных должен своевременно доходить до всех экземпляров SSP. Для решения этой задачи в прошлом было использовано и испытано множество вариантов механизма доставки и хранения данных. Ниже будут кратко описаны некоторые из них.

До начала разработки было решено разместить базу данных, в которой содержатся куки, на одной машине с использующей её SSP. Это обусловлено тем, что чтение из неё должно занимать минимально возможное время, и таким образом исключается влияние сети на скорость чтения и передачи данных.

На Рисунке 1 продемонстрирована первая реализация, в которой синхронизация производилась горизонтально с помощью встроенной репликации MongoDB. Кластер состоял из машин, каждая из которых содержала один экземпляр MongoDB и SSP, а данные приходили с HTTP-сервера.

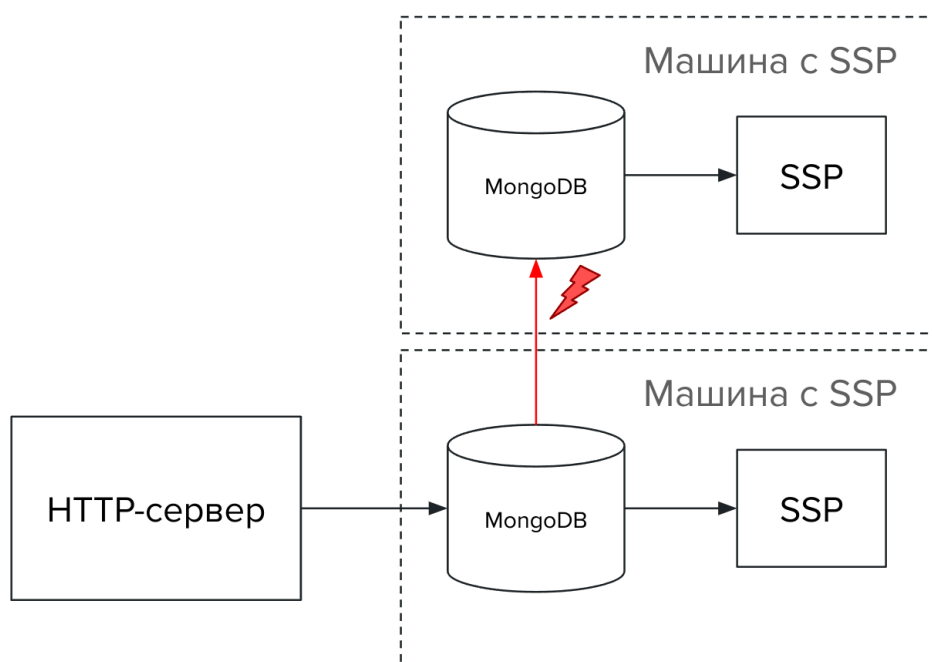


Рисунок 1 – Первый прототип

В результате нагрузочного тестирования у данной реализации были выявлены следующие преимущества и недостатки:

- + Быстрая запись и чтение у MongoDB;
- + Высокая надежность;
- Отсутствие полноценной мульти-мастер репликации у MongoDB;

- Медленная репликация при большом количестве экземпляров;
- Строгий верхний предел в 50 реплик.

Поскольку недостатки сильно перевешивали преимущества, было решено попробовать реализацию с помощью базы данных Tarantool, показанную на Рисунке 2.

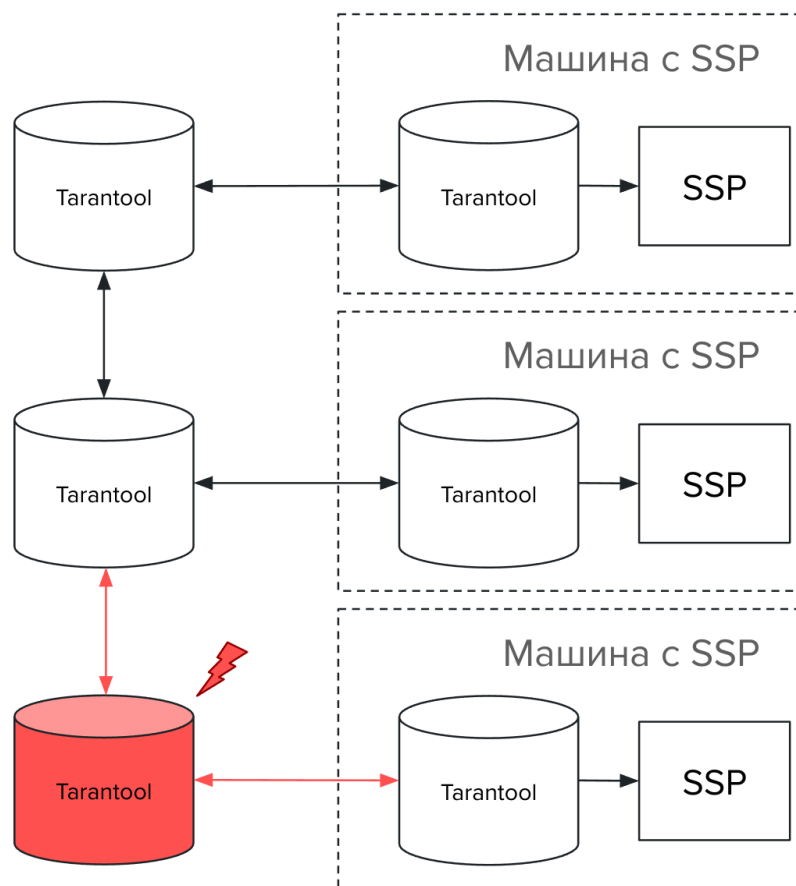


Рисунок 2 – второй прототип

В рамках реализации с Tarantool на каждой машине с SSP находился Tarantool – но также существовал и отдельный набор его экземпляров, находившихся вне машин с SSP, который осуществлял репликацию.

Эта реализация также имела высокую скорость записи и чтения, а также имела преимущество перед MongoDB в том, что у Tarantool имеется

более гибкая система репликации, в рамках которой можно сделать мульти-мастер, и отсутствует строгий верхний предел на количество реплик.

Но и эта реализация оказалась неподходящей, так как она сильно усложняет поддержку инфраструктуры. Кроме того, по неизвестной причине кластер начинал отказывать: master-экземпляры самопроизвольно превращались в readonly-экземпляры, а их время на их восстановление могло достигать до нескольких дней. Техническая поддержка не смогла помочь с решением этой проблемы, из-за чего от решения пришлось отказаться, а в будущем склоняться к независимости от сторонних решений, чтобы такая ситуация не повторялась на промышленном кластере.

Третья (и текущая) реализация показана на Рисунке 3. Здесь решено было оставить MongoDB в качестве базы данных для хранения куки, так как она хорошо показала себя в предыдущих прототипах, не считая репликацию, которая здесь не используется. Также в этой реализации было принято решение отказаться от согласованности данных между базами, так как условия задачи позволяют это сделать.

Так как сроки уже не позволяли сравнивать разные варианты, в качестве брокера для данных был выбран Redis, в котором существует механизм publisher-subscriber. Соответственно, несколько экземпляров Redis здесь выступают в роли брокера.



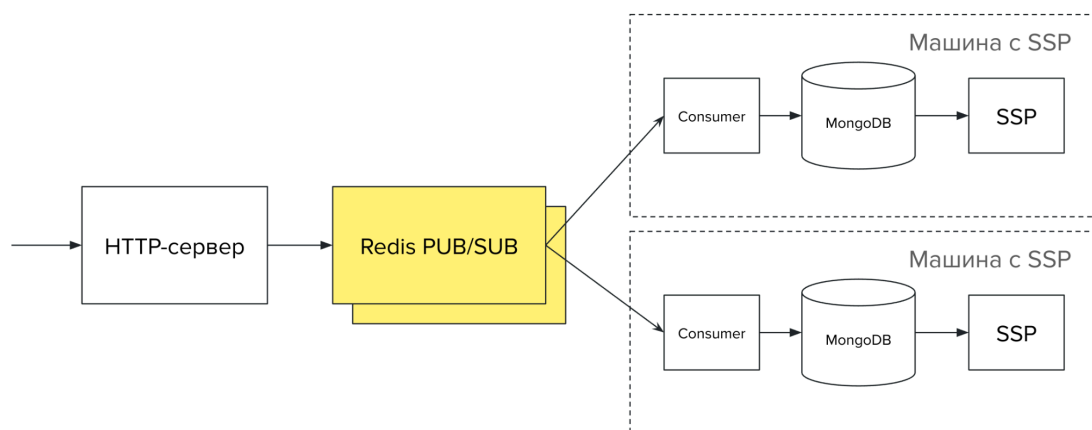


Рисунок 3 – Реализация через Redis в роли брокера

Данная реализация оказалась достаточно работоспособной, и выдерживала необходимую нагрузку, поэтому была развернута на промышленном кластере, и до недавних пор обеспечивала работу системы. Однако у нее оставались некоторые недостатки:

- Неудовлетворительное время доставки данных, достигающее при средней нагрузке до 5 секунд и более – что не критично само по себе, но нежелательно;
- Неудобное масштабирование, при котором нужно увеличивать количество экземпляров Redis для поддержания работоспособности;
- Неэффективное использование ресурсов, так как Redis потребляет много оперативной памяти и процессорного времени, а также имеет много избыточных функций, которые не требуются для решения данной задачи.

В рамках данной ВКР разрабатывается решение, которое избегает перечисленных недостатков, то есть выдерживает большую нагрузку при малом количестве экземпляров, имеет только необходимые функции, а также легко масштабируется.

### 1.3 Общий вид разрабатываемой системы

На рисунке 4 показан общий вид разрабатываемой системы.

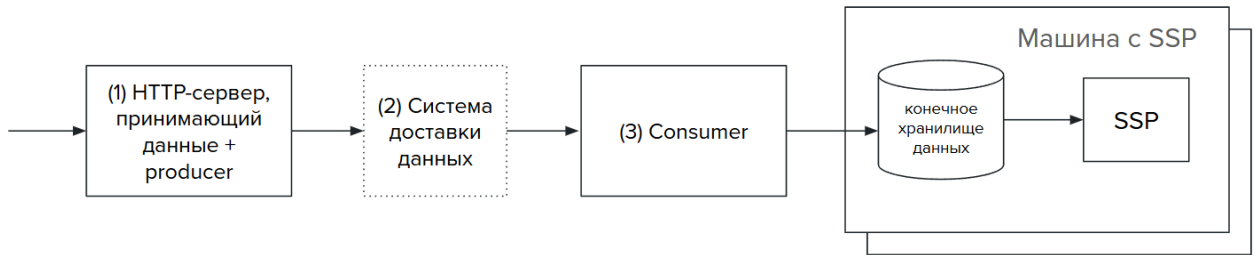


Рисунок 4 – Общий вид разрабатываемой системы

В рамках задачи необходимо разработать следующий набор компонентов, некоторые из которых могут быть как объединены, так и разделены.

Первый компонент – HTTP-интерфейс (сервер), принимающий данные. Данная система разрабатывается на замену внутреннему механизму доставки данных, из-за чего необходимо соответствовать уже используемому протоколу. HTTP-сервер будет иметь один эндпоинт, формат запроса к которому совпадает с существующим (он будет описан в третьей главе). Далее сервер будет преобразовывать полученные данные в необходимый формат, и отправлять в систему доставки данных.

Второй компонент – producer, который отправляет данные в систему доставки данных в соответствии с её интерфейсом.

Третий компонент – сама система доставки данных, которая может быть реализована разными способами (брокер, без брокера и тд).

Четвертый – consumer, то есть потребитель данных, который будет забирать данные из системы доставки, преобразовывать в необходимый формат, а затем записывать их в базу данных на SSP. В качестве базы данных будет использоваться MongoDB, так как она хорошо показала себя в

предыдущих реализациях, и может выдержать требуемую нагрузку. К тому же SSP уже написан под неё, из-за чего нежелательно выбирать другую БД, иначе пришлось бы вносить значительные изменения в SSP.

Эта схема не является финальной – итоговое решение может как объединять некоторые компоненты (что предпочтительно), либо разделять их (что нежелательно). Стоит помнить, что в число требований входит простота инфраструктуры и легковесность. Учитывая эти требования, рассмотрим различные существующие решения, с помощью которых можно реализовать систему доставки данных.

Все рассматриваемые готовые решения так или иначе являются брокерами сообщений – то есть системами, которые обеспечивают доставку сообщений от производителя к потребителю.

## **1.4 Аналоги и существующие решения**

### **1.4.1 RabbitMQ**

RabbitMQ – распределенный и горизонтально масштабируемый брокер сообщений. Он поддерживает протоколы AMQP, MQTT, STOMP (и другие, через отдельные модули) – это делает его подходящим для большого количества сценариев.

Однако для задачи синхронизации куки такое количество разных протоколов избыточно, так как помимо самих пользовательских данных никакие метаданные не передаются.

Внутри он содержит персистентную очередь сообщений, а также поддерживает механизм QoS (quality of service), при котором сообщение не будет удалено, пока принимающая сторона не подтвердит, что она его обработала.

И снова здесь имеется избыточная для задачи функциональность: персистентная очередь не нужна, так как куки должны как можно быстрее доходить до SSP, а запись на диск занимает время. Кроме того, она создает дополнительные факторы – приходится приобретать сервер с диском большой пропускной способности, определенным образом разворачивать приложение (так как оно из stateless становится stateful).

Более того: хранимые на диске куки будут просто терять актуальность к тому моменту, когда, допустим, отключившийся надолго SSP снова подключается и начинает их считывать. В таком случае гораздо важнее доставить на него самые свежие куки – они соответствуют пользователям, которые прямо сейчас должны получить новую рекламу, а значит их данные должны как можно быстрее дойти до SSP.

Персистентность в RabbitMQ можно отключить с помощью параметра `delivery_mode = 1`, но это лишь добавляет сложности развертывания и конфигурации.

Также RabbitMQ поддерживает гибкую маршрутизацию, которая позволяет направлять сообщения разным читателям в зависимости от содержимого или заданного приоритета. Например, можно разделить сообщения по темам, либо доставить их всем получателям через `fanout exchange`. Для данной задачи эти функции не нужны, так как весь входящий поток куки должен дублироваться всем получателям.

Модель получения сообщения получателями – PUSH, то есть RabbitMQ “проталкивает” сообщения, из-за чего получатели могут перегружаться, если не успевают их обрабатывать. В данном случае это нежелательно, так как `consumer` не должен перегружаться, иначе он будет записывать в MongoDB сильно устаревшие данные и нагружать всю машину, на которой также работает база данных и SSP.

При развертывании RabbitMQ требуется проводить настройку либо через графический интерфейс, либо через утилиту `rabbitmqadmin` – всё это нужно для создания сущностей, необходимых для работы брокера [3].

Пропускная способность одного брокера RabbitMQ составляет десятки тысяч сообщений [4], а для ее увеличения пришлось бы увеличивать количество экземпляров – это не соответствует требованиям задачи.

#### **1.4.2 Kafka**

Apache Kafka – распределенный программный брокер сообщений, являющийся распределенным реплицированным журналом фиксации изменений. На данный момент фактически является стандартом в индустрии по части рассылки сообщений.

Kafka в целом внешне похож на RabbitMQ, но работает иначе. Внутри Kafka содержится журнал сообщений, который считывается получателем по указателю на смещение. При этом старые сообщения остаются в системе, и при необходимости можно откатиться и вновь прочесть их. Таким образом, Kafka реализует модель PULL, то есть “вытягивания”, при которой потребитель считывает столько сообщений, сколько может обработать. Это лучше подходит для поставленной задачи, чем модель RabbitMQ.

Kafka обладает большой надежностью и горизонтальной масштабируемостью, что, однако, достигается в ущерб простоте использования – Kafka достаточно сложно разворачивается. До недавних пор для его работы требовался Zookeeper, который координировал экземпляры Kafka. На данный момент Kafka умеет работать без него, но сам факт необходимости координации добавляет сложности при использовании данной технологии.

В рамках данной задачи имеется большое преимущество: между конечными базами данных MongoDB не требуется согласованность. Вполне

приемлемо, что какие-то куки присутствуют в одних экземплярах, но отсутствуют в других. Благодаря этому допущению, здесь не требуются сложные механизмы координации – экземпляры producer и consumer могут быть вообще не связаны друг с другом. Следовательно, функциональность Kafka здесь будет избыточна.

Также Kafka написан на языке программирования Java, и для обеспечения работоспособности всей сложной функциональности требуется много системных ресурсов.

Для общения с Kafka используется собственный бинарный протокол, работающий поверх TCP.

Наконец, в отличие от RabbitMQ, в Kafka нельзя полноценно отключить персистентность, то есть всегда будет использоваться диск. Как уже было заявлено при описании RabbitMQ, это снижает скорость доставки и увеличивает потребление ресурсов.

Пропускная способность одного экземпляра Kafka также составляет десятки тысяч сообщений [5], что аналогичным с RabbitMQ образом не удовлетворяет требованиям задачи.

### **1.4.3 NATS**

NATS – распределенная система доставки сообщений, написанная на языке программирования Go. Создана как более легковесная альтернатива классическим брокерам, таким как RabbitMQ.

В NATS отсутствует персистентность или транзакции, нет гарантий доставки, и вся система в целом проектировалась с упором на простоту развертывания и использования.

В статье “Dissecting Message Queues” описываются результаты тестирования пропускной способности NATS и других брокеров [6] –

Рисунок 5. Как можно заметить, показатели Kafka соответствуют упомянутым ранее, а RabbitMQ имеет гораздо меньшую пропускную способность (скорее всего, из-за отличий методики тестирования, а также того факта, что изначальное число было взято с официального сайта RabbitMQ).

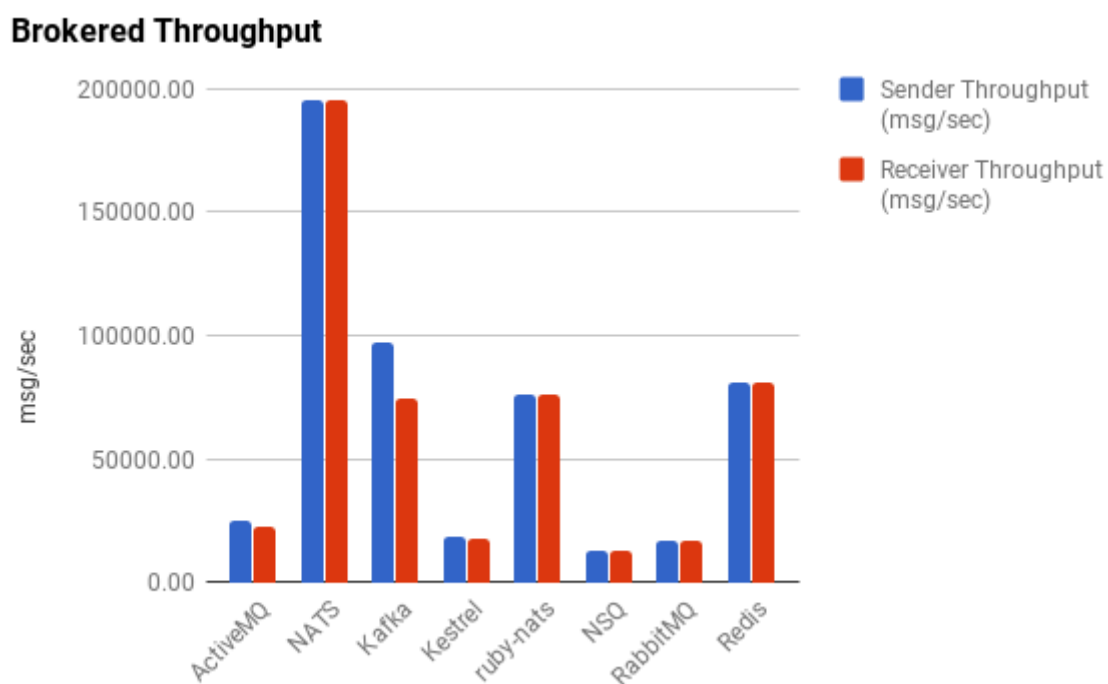


Рисунок 5 – пропускная способность брокеров сообщений

Достигается это, вероятнее всего, за счет отсутствия персистентности, и простоты системы в целом. Пропускная способность в 200 тысяч сообщений в секунду в целом соответствует требованиям, но не может создать проблемы в будущем, если нагрузка сильно увеличится.

NATS имеет достаточно простой и легковесный, однако текстовый протокол, работающий поверх TCP [7]. Также NATS поддерживает как PUSH, так и PULL-модели получения [8].

Достоверных показателей использования системных ресурсов NATS найдено не было, но можно предположить, что оно сильно меньше по сравнению с RabbitMQ и Kafka, поскольку NATS написан на языке программирования Go, и не использует диск.

#### 1.4.4 Сравнение существующих решений в таблице

В Таблице 1 содержится список критериев, и соответствие рассмотренных решений этим требованиям.

Таблица 1 – сравнение существующих решений

Требование \ Решение	RabbitMQ	Kafka	NATS
Простое развертывание и конфигурация	Нет	Нет	Да
Малое потребление ресурсов	Нет	Нет	Да?
Пропускная способность экземпляра	~10-90k RPS	~10-90k RPS	~200k RPS
Модель получения	PUSH	PULL	PUSH/PULL
Протокол (простота и легковесность)	AMQP	Собственный бинарный	Собственный текстовый
Режим без использования диска	Да, с доп. конфигурацией	Нет	Да
Гибкая настройка интерфейса записи	Нет	Нет	Нет

Общим недостатком реализации с использованием брокеров сообщений является большое количество инфраструктурных компонентов в системе, поскольку для реализации все равно понадобятся элементы (1) и (3) с рисунка 1. Также брокеры вносят дополнительную сложность в виде необходимости



конфигурации и развертывания кластера, имеют вероятность возникновения сложно устранимых неполадок (split brain и ему подобных).

Большинство брокеров имеют HTTP-интерфейс, но для данной задачи требуется конкретный формат запроса, а у брокеров отсутствует возможность (по крайней мере, из коробки) гибко настраивать свой HTTP-интерфейс, чтобы он принимал данные в нужном формате. Следовательно, всё равно придется разворачивать дополнительный компонент (producer), который будет служить прослойкой между клиентом и брокером.

Также перечисленные решения обладают избыточной функциональностью, которая вносит сложность в разработку, тратит процессорное время и другие ресурсы. Например, RabbitMQ использует протокол AMQP, поддерживает QoS, различные виды очередей – для данной системы всё это является излишним.

Также персистентность, включенная по умолчанию (а в Kafka она вообще не выключается), которая обычно является преимуществом в других сценариях использования, здесь является недостатком, так как это будет тратить лишние ресурсы – как вычислительные, так и денежные (нужен быстрый диск на машине). К счастью, здесь можно пренебречь персистентностью, так как при длительных неполадках сохраненные на диске данные уже станут достаточно неактуальными (куки наиболее актуальны в течении короткого периода после регистрации и 1-2 дня после него), а вот выигрыш от повышенной производительности принесет больше денег.

Как можно заметить по результатам сравнения, существующие решения не соответствуют всем требованиям, выставленным к решению – поэтому было принято решение о реализации собственного решения.

## 2 Описание предложенного решения

### 2.1 Общее описание

В предыдущей главе был обоснован отказ от существующих брокеров сообщений, поэтому в рамках данного решения будет реализован механизм без брокера.

Поскольку и HTTP-сервер, и producer должны быть реализованы самостоятельно, их можно объединить в один компонент producer, который будет отправлять данные потребителям (consumer). При этом каждый consumer находится на машине с SSP и базой данных, что уменьшает задержку при отправке данных с него в базу данных. В качестве БД, как говорилось ранее, будет использоваться MongoDB. Общая схема реализуемой системы проиллюстрирована на Рисунке 6.

Также в этой модели присутствует балансировщик нагрузки, который будет распределять данные между экземплярами producer, если потребуется увеличить их количество. В качестве балансировщика может выступать любое существующее решение, так как они все имеют аналогичный функционал. Например: haproxy, nginx, Kubernetes.

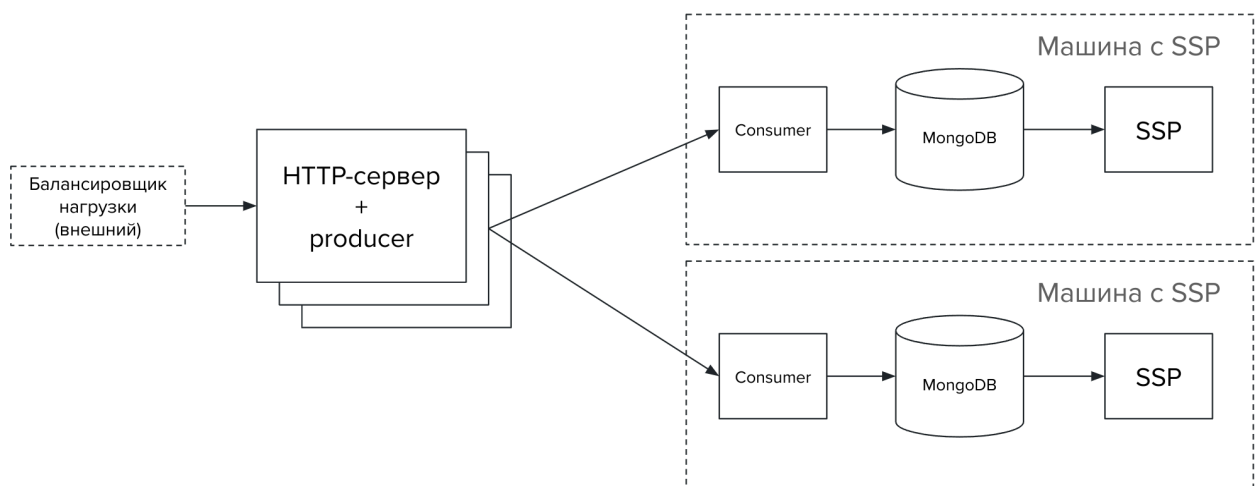


Рисунок 6 – Общая схема системы

Реализация без брокера сильно упрощает инфраструктуру, так как требуется всего дополнительных 2 вида приложений (3, если считать балансировщик нагрузки – но он может присутствовать в любой из реализаций).

Далее необходимо выбрать инструменты и технологии, которые будут использоваться для реализации этих компонентов.

## **2.2 Язык программирования**

В качестве языка программирования выбран Go, так как он совмещает множество качеств, необходимых для разработки надёжных высоконагруженных приложений [9]:

- Простота кода;
- Быстрая компиляция;
- Достаточное количество удобных абстракций (например, интерфейсы);
- Богатая стандартная библиотека;
- Встроенная поддержка многопоточности в виде примитивов синхронизации и горутин, то есть реализации легковесных “зеленых” потоков со стеком переменного размера.
- Высокая производительность;
- Активное сообщество и поддержка.

Далее оставались две основные задачи: выбор библиотеки для HTTP-сервера, протокола передачи данных на consumer, а также второстепенных библиотек.

## **2.3 HTTP-сервер**

Для HTTP-сервера вполне можно было бы взять пакет `net/http` из стандартной библиотеки Go, так как он обладает достаточной функциональностью. Однако известно, что он сильно проигрывает по производительности другим реализациям (например, `fasthttp` [10]), а для

данного решения нужно обеспечить низкое потребление ресурсов процессора и оперативной памяти, и не сделать HTTP-сервер узким местом.

По этой же причине исключены популярные библиотеки, которые предоставляют более удобный интерфейс для разработки RESTful интерфейсов, десериализацию JSON-тела через рефлексию и другие удобства (например, библиотеки `echo` или `gin`). Не использовались и генераторы сервера из OpenAPI-спецификации, поскольку в данном решении реализуется лишь один эндпоинт, и можно пожертвовать удобством разработки ради производительности.

В итоге была выбрана высокопроизводительная библиотека **fasthttp**. Хотя она и не имеет удобного интерфейса с синтаксическим сахаром, она дает возможность максимально оптимизировать работу сервера. Эта библиотека разработана для высокопроизводительных систем, которым требуется обрабатывать тысячи маленьких запросов в секунду и иметь стабильное время ответа на уровне нескольких миллисекунд. По результатам бенчмарка, сервер на fasthttp работает в среднем в 10 раз быстрее сервера на net/http [10].

## 2.4 Протокол передачи данных на consumer

Далее предстоял выбор протокола передачи данных на consumer. Здесь стояло две задачи – выбор транспортного протокола, а также формата сериализации данных. При этом протокол и формат сериализации должны иметь минимальные накладные расходы, так как поток данных будет значительным, и нужно максимально эффективно тратить ресурсы сервера на обработку полезной нагрузки.

Рассмотрение было начато с очевидного – обычный RESTful HTTP. Здесь он неэффективен, так как отдельные HTTP-вызовы имеют большие накладные расходы, а также избыточную в данном случае функциональность.

Опустившись на уровень ниже, можно было бы использовать TCP или UDP. Сразу исключается UDP, так как он не обладает достаточной надежностью, и в таком случае гарантия доставки будет слишком малой даже для данной задачи. Однако на простом TCP писать достаточно сложно, поскольку реализовывать более продвинутый функционал (например, повторную отправку при ошибках и мультиплексирование запросов) придется самостоятельно, что усложняет разработку.

Также еще одним существующим решением для организации работы подобных систем без участия брокера (либо с собственной его реализацией) является библиотека ZeroMQ, которая содержит большое количество различных сокетов, позволяющих реализовывать различные сложные топологии, такие как PUB-SUB, Request-Reply и т.д. Недостатками данной библиотеки является небольшое сообщество и то, что основа библиотеки написана на языке программирования C, а для Go существуют либо обёртки, либо малоизвестные любительские реализации. Из-за этого ZeroMQ представляется недостаточно надежным и удобным решением.

Оптимальным решением здесь выступит gRPC – RPC-фреймворк, разработанный Google. Он работает поверх одного соединения HTTP/2.0, а также добавляет богатую функциональность поверх него, но с небольшими накладными расходами. Он разрабатывается одной из крупнейших технологических компаний, постоянно обновляется и обладает большим сообществом, имеет нативную реализацию на языке программирования Go. Забегая вперед, стоит отметить, что он также имеет встроенный механизм сериализации и десериализации через protobuf, что будет подробнее описано ниже.

Далее – выбор протокола сериализации: можно отметить, что здесь не требуется человеко-читаемость сообщений, так что популярные форматы

JSON и XML можно не рассматривать, так как у них достаточно большие накладные расходы.

Протоколы сериализации делятся по критерию наличия “схемы”, то есть фиксированного строго типизированного формата данных. Его наличие позволяет ощутимо ускорить сериализацию и десериализацию. Также есть и “schema-less” форматы, например msgpack, который работает гораздо эффективнее JSON [11]. К счастью, куки имеет фиксированный формат данных, что дает возможность воспользоваться преимуществами протоколов со схемой.

Здесь можно вернуться к фреймворку gRPC: он нативно предоставляет формат сериализации со схемой, называемый protobuf. Он является самым одним из самых эффективных среди таких форматов, а gRPC предоставляет инструменты для удобного использования его в разработке. Схема задается в виде спецификации в proto-файле, из которого затем генерируется сервер, клиент и необходимые типы и структуры.

## 2.5 Внутренняя очередь

Также будет реализована внутренняя очередь, чтобы обеспечить буфер для данных. Здесь не понадобятся никакие библиотеки, ибо достаточно будет воспользоваться одним из примитивов синхронизации, присутствующих в Go – **каналами**.

Внутренняя очередь будет реализована как буферизированный канал с длиной, которую желательно подбирать в зависимости от характеристик машины, на которой запускается сервис. Нужно учесть, что при стабильной работе сервиса, то есть без перегрузок, количество элементов в очереди будет около нуля, а расти она будет при перегрузках (в таком случае нужно будет увеличивать количество экземпляров приложения), либо при потере

соединения. Тем не менее, каналы достаточно легковесны, и не потребляют большого количества памяти, поэтому выбор длины не критичен.

Рассмотрим сразу крайний случай, когда очередь максимально заполняется данными – например, при временных неполадках в сети. В таком случае будет выгоднее удалять самые старые элементы, так как они с меньшим шансом приведут к успешному выкупу рекламы, так как их актуальность падает со временем (см. Главу 1).

Как и говорилось ранее, в модуле передачи данных персистентности не будет, а внутренняя очередь будет содержаться исключительно в оперативной памяти. Это сильно упростит разработку, развертывание и масштабирование, сэкономит время обработки данных и уменьшит задержку.

## **2.6 Прочие инструменты**

Для разработки также понадобятся некоторые другие библиотеки, которые не влияют на производительность конечной системы, и выбраны на основе личных предпочтений и опыта разработчика:

1. Библиотека `zerolog` для логирования с минимальным количеством выделений памяти. В данном случае подойдет почти любая другая библиотека, так как важно лишь не размещать вызов логгера в критических участках кода, которые это сильно замедлит;
2. Библиотека  `viper` для гибкой загрузки конфигурации, поддерживающая такие форматы как `YAML`, `JSON`, `TOML`, а также загрузку из флагов и переменных среды;
3. Библиотека  `ozzo-validation` для декларативного задания правил валидации конфигурации;
4. Инструмент `rrprof` для снятия метрик с приложений на `Go`;
5. Библиотека для формирования метрик для `Prometheus`, так как он используется для снятия метрик с приложений в компании;

6. Grafana для визуализации метрик при нагрузочном тестировании;
7. Docker и docker-compose для удобного развертывания приложения и его зависимостей, в том числе и для локальной разработки (например, чтобы локально развернуть MongoDB);
8. Драйвер MongoDB для Go.



## 3 Практическая реализация

### 3.1 Общая схема

Реализация системы имеет достаточно простую структуру, которая уже была описана ранее. В качестве HTTP-сервера и отправителя сообщений выступает компонент `producer`, а потребителя, принимающего сообщения и отправляющего данные в базу – `consumer`.

Эти два компонента написаны как отдельные приложения на языке Go. Связаны они протоколом `gRPC`, спецификация которого написана в отдельном модуле, чтобы `producer` и `consumer` зависели от нее, а не друг от друга. Таким образом, граф зависимостей очень прост и выглядит следующим образом, проиллюстрированным на Рисунке 7.

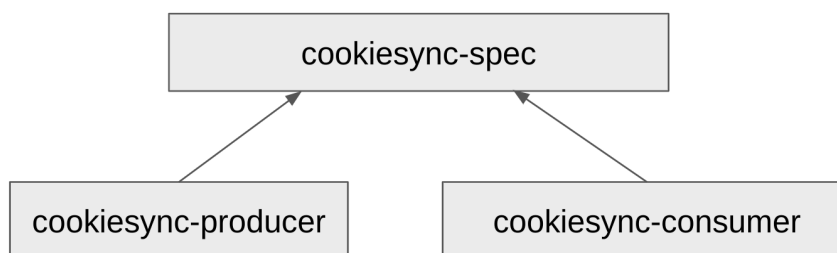


Рисунок 7 – Схема зависимостей компонентов

Также был рассмотрен важный аспект: по какой модели будут взаимодействовать `producer` и `consumer`. Здесь было два варианта:

- PUSH-модель: `producer` подключается к `consumer` и активно пытается отправлять на него данные, вне зависимости от того, может ли `consumer` их обработать.
- PULL-модель: `consumer` подключается к `producer` и принимает от него данные тогда, когда готов их обработать. В случае перегрузки `consumer` медленнее принимает данные, либо отключается, а подключается уже тогда, когда снова готов их обработать.

Для данной реализации логичнее было использовать PULL-модель по ряду причин:

1. Из статистики, имеющейся у компании, известно, что количество экземпляров SSP, а значит и consumer, растет быстрее количества экземпляров producer, так как нагрузка на SSP растет быстрее количества запросов на синхронизацию куки. Таким образом, с точки зрения развертывания удобнее задавать адреса producer-ов на consumer-ах, так как их реже придется обновлять.
2. Машина с SSP не должна быть перегружена, а consumer находится как раз на ней. Если бы использовалась PUSH-модель, то producer мог бы перегрузить consumer, а с ним и машину с SSP. При этом на consumer пришлось бы реализовывать отдельный функционал для восстановления из перегруженного состояния, чтобы он возвращался в работоспособное состояние.

Схема архитектуры реализуемой системы представлена на Рисунке 8.

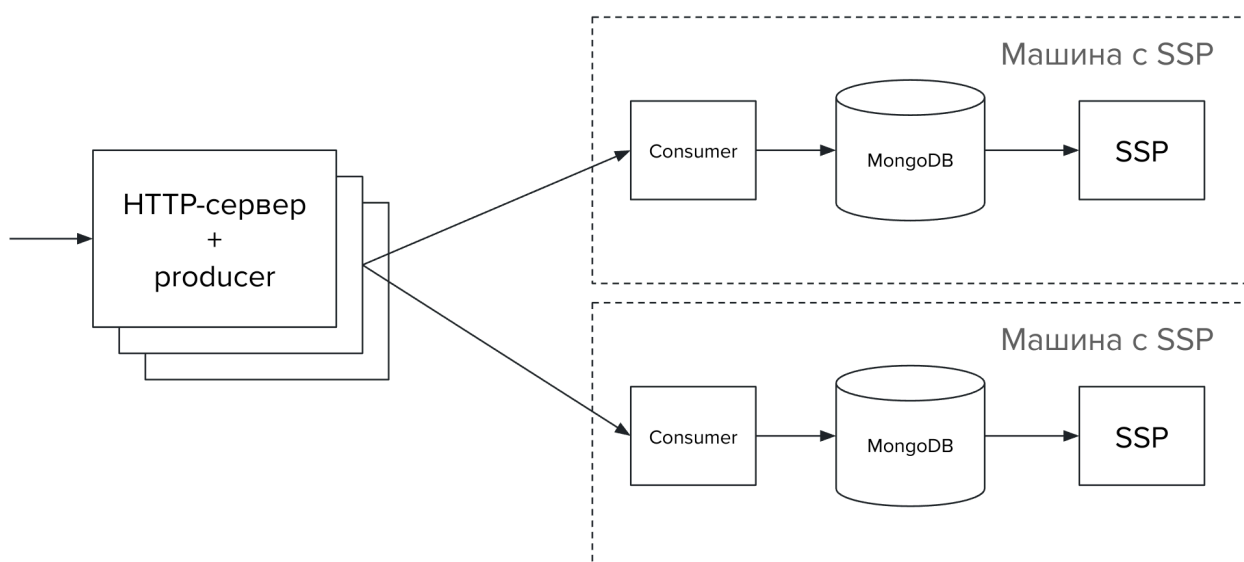


Рисунок 8 – Схема архитектуры реализуемой системы

В качестве конечного хранилища используется БД MongoDB, обеспечивающая высокую скорость записи и чтения с помощью движка WiredTiger. Здесь она имеет следующую структуру: MongoDB делится на базы данных, каждая из которых относится к отдельному DSP (их всего около 20). Далее для структуры каждой базы есть два варианта.

Первый – использовать одну коллекцию для всех куки. Это упростит реализацию записи в базу, но *может* понизить скорость записи в эту коллекцию из-за глубины дерева.

Второй – разбить базу на несколько коллекций (например, 256), где коллекция будет определяться остатком от деления `userId`, содержащегося в куки, на 256, поскольку `userId` также является числом в шестнадцатеричной системе счисления. Это сделано для того, чтобы уменьшить время чтения и записи в MongoDB за счет уменьшения глубины дерева, лежащего в основе коллекции.

Для изначальной реализации был выбран второй вариант, основываясь на предположении, что глубина дерева больше влияет на производительность, чем управление несколькими коллекциями. Выбор окончательной реализации будет сделан на основе нагрузочного тестирования в Главе 4.

### **3.2 Спецификация protobuf, gRPC и генерация кода**

Практическая реализация была начата с написания библиотеки на языке Go, содержащей в себе спецификацию gRPC-интерфейса на языке protobuf, а также сгенерированные из неё клиент и сервер.

Описаны метаданные спецификации, то есть версия синтаксиса proto3, и название генерируемого пакета в Go “spec”. Затем описан формат сообщения (Таблица 2), которое будет передаваться в виде потока данных с producer на consumer. Сообщение Cookie содержит в себе `user_id` – идентификатор пользователя в системе компании, `dsp_id` – идентификатор

DSP, а также `buyer_uid` – идентификатор пользователя в системе DSP. Каждому полю обязательно назначается числовой тег, который помогает сохранять обратную совместимость при изменении протокола.

Таблица 2 – Спецификация сообщения Cookie

```
message Cookie {  
  string user_id = 1;  
  uint32 dsp_id = 2;  
  string buyer_uid = 3;  
}
```

В качестве запроса будет использоваться сообщение `GetCookiesRequest`, содержащее единственное текстовое поле `host` – с его помощью producer может идентифицировать подключившийся consumer, что может быть полезно для различных задач. Например, в реализации producer, описанной ниже, `host` будет использоваться для подключения отключившегося по какой-то причине экземпляра consumer к ранее используемой очереди, которую он не успел обработать.

Наконец, описана реализация сервиса `CookieStreamer` (Таблица 3), имеющего единственный метод `GetCookies`, который принимает на вход одиночное сообщение `GetCookiesRequest`, а возвращает поток из `Cookie`. В gRPC такой механизм называется “server streaming” – также существуют “client streaming” и “bi-directional streaming”, но в предыдущем пункте была выбрана PULL-модель – а для нее подходит как раз server streaming. Stream минимизирует накладные расходы на передачу большого потока данных, что поможет в решении поставленной задачи.

Таблица 3 – Спецификация сервиса CookieStreamer

```
service CookieStreamer {  
  rpc GetCookies(GetCookiesRequest) returns (stream Cookie) {}  
}
```

Для генерации кода на языке программирования Go потребовалась установка трех зависимостей:

1. protoc – компилятор protobuf;
2. protoc-gen-go – плагин компилятора для генерации Go-структур из типов, описанных в protobuf-спецификации, а также методов сериализации и десериализации. Этот плагин генерирует только сами структуры, поэтому его будет недостаточно для реализации сервера;
3. protoc-gen-go-grpc – плагин для генерации клиентских и серверных RPC-интерфейсов в Go, который реализует интерфейс CookieStreamer.

Результатом генерации являются два файла: spec.pb.go, spec\_grpc.pb.go, содержащие сгенерированный код. Далее этот пакет был импортирован и использован для реализации коммуникации между producer и consumer по протоколу gRPC.

### 3.3 Метрики

В разработанных приложениях присутствуют метрики, реализованные в виде счетчиков Prometheus. Для возрастающих счетчиков используется Counter, для произвольных – Gauge. Пример такого счетчика в модуле Handler компонента Producer – Таблица 4. Метрики позволят убедиться, что решение соответствует требованиям, а также вовремя заметить неполадки в работе системы.

Таблица 4 – Пример счетчиков Prometheus в модуле Handler

```
type Handler struct {
    // ...
    metrics struct {
        accepted prometheus.Counter
        published prometheus.Counter
        errors    prometheus.Counter
    }
}

//...
h.metrics.published = prometheus.NewCounter(prometheus.CounterOpts{
    Namespace: "producer",
```

```
Subsystem: "handler",
Name:      "published",
Help:      "Total count of cookies published by handler",
})
reg.MustRegister(h.metrics.published)
```

Счетчики доступны по адресу “/metrics”, откуда их считывает Prometheus. Этот адрес расположен на отдельном HTTP-сервере (который использует сервер из стандартной библиотеки, так как здесь нет требования высокой пропускной способности) вместе с обработчиками для профилирования, которые считываются утилитой pprof. Код, инициализирующий эти обработчики – в таблице 5.

Таблица 5. Инициализация обработчиков для метрик и профилирования

```
// Pprof profiles
mux.HandleFunc("/debug/pprof/", pprof.Index)
mux.HandleFunc("/debug/pprof/cmdline", pprof.Cmdline)
mux.HandleFunc("/debug/pprof/profile", pprof.Profile)
mux.HandleFunc("/debug/pprof/symbol", pprof.Symbol)
mux.HandleFunc("/debug/pprof/trace", pprof.Trace)
mux.Handle("/debug/pprof/block", pprof.Handler("block"))
mux.Handle("/debug/pprof/heap", pprof.Handler("heap"))
mux.Handle("/debug/pprof/goroutine", pprof.Handler("goroutine"))
mux.Handle("/debug/pprof/threadcreate", pprof.Handler("threadcreate"))

// Prometheus metrics
mux.Handle("/metrics", promhttp.HandlerFor(reg, promhttp.HandlerOpts{Registry:
reg}))
```

### 3.4 Реализация компонента producer

Приложение producer реализовано со следующей архитектурой, проиллюстрированной на Рисунке 9:

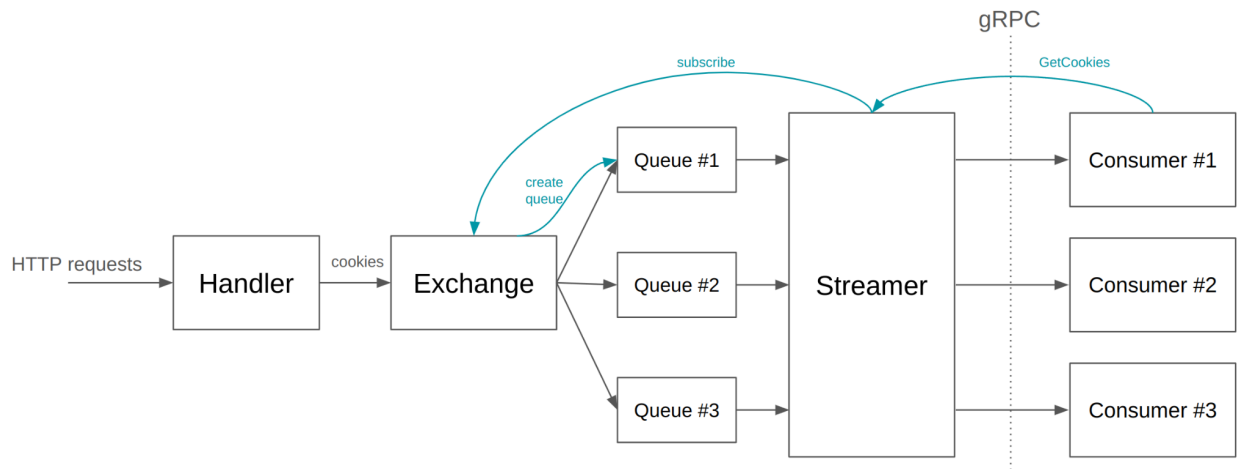


Рисунок 9 – архитектура приложения producer

Модули приложения, слева направо:

1. Handler – обработчик HTTP-запросов, который принимает данные от клиента, преобразует во внутренний формат (go-структуру с тремя полями) и передает их в Exchange;
2. Exchange – упрощенная реализация механизма publisher-subscriber. Принимает данные от Handler и копирует их в отдельные очереди, а также создает эти очереди по запросу от Streamer;
3. Queue – набор очередей, каждая из которых соответствует каждому подключенному consumer;
4. Streamer – gRPC-сервер, реализующий описанную в предыдущем пункте спецификацию. Регистрирует consumer-ы и отправляет им данные.

Синим цветом на рисунке описывается регистрация consumer в producer:

1. Consumer отправляет запрос GetCookies, который принимается модулем Streamer;
2. Streamer обращается к Exchange, сообщая ему имя consumer-a, содержащееся в аргументе host в gRPC-сообщении;
3. Exchange либо создает новую очередь, куда сразу начинает записывать новые данные, либо ищет очередь по имени среди уже существующих – для случаев перепоключения consumer-a. После этого он возвращает эту очередь Streamer-у;
4. Streamer обрабатывает данные из очереди и отправляет их на consumer.

Далее подробнее рассматривается реализация конкретных модулей, представляющих интерес. Реализация загрузки и валидации конфигурации, настройки модулей в функции main рассматриваться не будут, так как довольно тривиальны и типичны для программ на языке Go.

### 3.4.1 Создание структуры Cookie

Структура данных Cookie соответствует структуре из gRPC-спецификации, и содержит те же три поля (Таблица 6).

Таблица 6 – Структура данных Cookie

```
type Cookie struct {  
    DspId    uint32  
    UserId   string  
    BuyerUid string  
}
```

### 3.4.2 Разработка модуля Handler

Здесь и в других модулях рассматриваются только те части кода, где реализуется главная функциональность, и опущены конструкторы и другие второстепенные функции. Модуль Handler использует библиотеку fasthttp для обработки запросов. Сам запрос содержит в себе 3 аргумента: 2 из них



содержатся в параметрах запроса в URI, а один – в HTTP cookie. Листинг обработчика запросов – Таблица 7.

Таблица 7 – обработчик запросов

```
func (h *Handler) cookieSync(ctx *fasthttp.RequestCtx) (err error) {
    args := ctx.QueryArgs()

    dsp, err := fasthttp.ParseUint(args.Peek("d"))
    if err != nil {
        return fmt.Errorf("parse dsp id from query args: %w", err)
    }

    if dsp <= 0 {
        return errors.New("dsp id must be positive")
    }

    user := string(ctx.Request.Header.Cookie("uid"))

    buyerId := string(args.Peek("b"))
    if len(buyerId) == 0 {
        return errors.New("missing buyer id")
    }

    go func() {
        h.exchange.Publish(cookie.Cookie{
            DspId:    uint32(dsp),
            UserId:   user,
            BuyerUid: buyerId,
        })
        h.metrics.published.Add(1)
    }()

    return
}

func (h *Handler) handleRequest(ctx *fasthttp.RequestCtx) {
    h.metrics.accepted.Add(1)

    if err := h.cookieSync(ctx); err != nil {
        h.metrics.errors.Add(1)
        ctx.SetStatusCode(fasthttp.StatusBadRequest)
        h.log.Err(err).Msg("Failed to handle cookie sync request")
        return
    }

    ctx.SetStatusCode(fasthttp.StatusOK)
}
```

Разработчики библиотеки fasthttp рекомендуют минимизировать копирование буферов и переиспользовать объекты с помощью пулов. Например, вызов args.Peek("d") возвращает указатель на массив байт, который можно было бы преобразовать в строку с помощью пакета unsafe из

стандартной библиотеки Go (избегая таким образом копирования), и использовать в рамках метода `cookieSync`, после чего этот массив был бы использован снова – это сильно уменьшает количество выделений памяти.

Однако в данном случае это не представляется возможным, поскольку полученные в запросе данные (`dsp`, `user`, `buyerId`) отправляются в канал на отправку получателям – следовательно, на момент отправки по `gRPC` указатели будут указывать на массивы байт, содержащие уже совсем другие значения, потому что они были переиспользованы сервером `fasthttp`. Получается, что копирования здесь не избежать.

После выполнения перечисленных действий запускается горутина, которая отправляет данные в `Exchange`. Это делается асинхронно, чтобы не задерживать клиент. Горутины имеют крайне малые накладные расходы на создание и работу, поэтому создание даже сотен тысяч в секунду является идиоматической практикой в языке Go.

В целом, в данном модуле в принципе отсутствует потенциал для улучшений, но, как станет понятно в процессе тестирования в 4 главе, сама библиотека `fasthttp` настолько эффективно переиспользует ресурсы, что приложение выдерживает требуемую нагрузку и потребляет адекватный по современным меркам объем ресурсов настолько, что узким местом становится сетевая карта сервера, не позволяющая получить более высокую пропускную способность.

### **3.4.3 Модуль Exchange**

`Exchange` содержит следующие методы:

1. `Publish` – вызывается `Handler`-ом для добавления данных;
2. `Subscribe` – вызывается `Streamer`-ом для подписки на данные;
3. `Unsubscribe` – вызывается `Streamer`-ом при отключении потребителя;

Основная сложность при реализации модуля Exchange состояла в необходимости синхронизации потоков при одновременной публикации (записи данных в очереди), обработки подписок и отписок (то есть изменении списка потребителей), а также завершения работы самого Exchange.

В итоге для синхронизации используется мьютекс – но только при изменении списка подписчиков. В методе Publish (где список подписчиков только читается) мьютекс сознательно игнорируется (таблица 8), так как иначе он создавал бы лишнюю задержку отправки.

Таблица 8 – метод Publish

```
func (e *Exchange) Publish(cookie cookie.Cookie) {  
    for _, s := range e.subscribers {  
        if len(s.queue) == cap(s.queue) {  
            <-s.queue // drop oldest  
            e.metrics.lost.Add(1)  
        }  
  
        s.queue <- &cookie  
    }  
}
```

При этом Publish все равно потокобезопасен, так как при отключении подписчика его очередь (канал) **не** закрывается на запись – поэтому если Publish все-таки успел взять очередь отключившегося подписчика, то он просто запишет туда лишние несколько куки, и никаких критических ошибок не возникнет. При этом память, занятая этим каналом, все равно будет освобождена сборщиком мусора – закрытие каналов в Go нужно лишь для оповещения читающих из него горутин, что данных больше не поступит, и не является обязательным.

Также реализован механизм (таблица 9), позволяющий временно отключившемуся подписчику прочесть “упущенные” данные. После отключения подписчика Exchange пометит очередь как неактивную, но в течении минуты продолжит писать в неё данные – и лишь после этого

полностью удалит её. За это время подписчик должен успеть переподключиться, и тогда он сможет получить эти данные.

Таблица 9 – Методы Subscribe и Unsubscribe

```
func (e *Exchange) Subscribe(name string) <-chan *cookie.Cookie {
    e.mut.Lock()
    defer e.mut.Unlock()

    // Find lost queue if it exists
    for i, sub := range e.subscribers {
        if sub.name == name {
            e.subscribers[i].active = true
            e.log.Info().Str("name", name).Msg("Queue reactivated, attached subscriber")
            return sub.queue
        }
    }

    s := Subscriber{
        name:    name,
        active:  true,
        queue:    make(chan *cookie.Cookie, queueSize),
    }

    e.subscribers = append(e.subscribers, s)

    e.log.Info().Str("name", name).Msg("Subscribed")

    return s.queue
}

func (e *Exchange) Unsubscribe(name string) {
    e.mut.Lock()
    defer e.mut.Unlock()

    e.log.Info().Str("name", name).Msg("Unsubscribed, marked as inactive")

    for i, sub := range e.subscribers {
        if sub.name == name {
            e.subscribers[i].active = false
        }
    }

    go func() {
        time.Sleep(time.Second * 5)

        e.mut.Lock()
        defer e.mut.Unlock()

        for _, sub := range e.subscribers {
            if sub.name == name {
                if sub.active { // reconnected, don't delete queue
                    return
                }
            }
        }
    }
}
```

```

    subs := make([]Subscriber, 0, len(e.subscribers))

    for _, sub := range e.subscribers {
        if sub.name == name {
            continue
        }
        subs = append(subs, sub)
    }

    e.log.Info().Str("name", name).Msg("Deleted inactive queue")
    e.subscribers = subs
}()
}

```

Демонстрация успешного переподключения (“Queue reactivated”), а затем полного отключения с удалением очереди (“Deleted inactive queue”) – Рисунок 10.

```

6:14PM INF Subscribed module=exchange name=cl-hot1-2
6:14PM DBG Handling new subscriber hostname=cl-hot1-2 module=streamer
6:14PM DBG Context is closed, stopping GetCookies stream hostname=cl-hot1-2 module=streamer
6:14PM INF Unsubscribed, marked as inactive module=exchange name=cl-hot1-2
6:14PM DBG Unsubscribed hostname=cl-hot1-2 module=streamer
6:14PM INF Queue reactivated, attached subscriber module=exchange name=cl-hot1-2
6:14PM DBG Handling new subscriber hostname=cl-hot1-2 module=streamer
6:14PM INF Deleted inactive queue module=exchange name=cl-hot1-2
6:14PM DBG Context is closed, stopping GetCookies stream hostname=cl-hot1-2 module=streamer
6:14PM INF Unsubscribed, marked as inactive module=exchange name=cl-hot1-2
6:14PM DBG Unsubscribed hostname=cl-hot1-2 module=streamer
6:14PM INF Deleted inactive queue module=exchange name=cl-hot1-2

```

Рисунок 10 – демонстрация механизма переподключения

### 3.4.4 Разработка модуля Streamer

Модуль Streamer реализует интерфейс gRPC-сервиса из пакета cookiesync-спес. Основной функционал содержится в методе GetCookies, где Streamer обращается к Exchange для регистрации подписчика, получает от него канал, и входит в бесконечный цикл, где есть две ветви, выбор между которыми выполняется с помощью оператора select (таблица 10):

1. `ctx.Done()` означает, что контекст закрыт, а значит пользователь закрыл соединение – в таком случае `Streamer` отписывается от `Exchange` и завершает работу с этим клиентом.
2. `c, ok := <-cookies` означает приход очередного пакета данных, либо закрытие канала. Если `ok` содержит `false` – это означает, что приложение выключается, и `Streamer` завершает обработку.
3. Если же `ok` содержит `true`, то полученные данные отправляются на `consumer`.

Подобная реализация обрабатывает все крайние случаи и гарантирует работу модуля без критических ошибок, а также мягкое завершение работы модуля при выключении приложения.

Таблица 10 – метод `GetCookies` модуля `Streamer`

```
func (s *Streamer) GetCookies(r *spec.GetCookiesRequest, server
spec.CookieStreamer_GetCookiesServer) (err error) {
    if strings.TrimSpace(r.Host) == "" {
        return errors.New("hostname must not be empty")
    }

    ctx := server.Context()

    l := s.log.With().
        Str("module", "streamer").
        Str("hostname", r.Host).
        Logger()

    cookies := s.exchange.Subscribe(r.Host)
    l.Debug().Msg("Handling new subscriber")
    s.metrics.activeConnections.Add(1)

    defer func() {
        s.metrics.activeConnections.Add(-1)
        s.exchange.Unsubscribe(r.Host)
        l.Debug().Msg("Unsubscribed")
    }()

    for {
        select {
        case <-ctx.Done():
            // Client closed connection
            l.Debug().Msg("Context is closed, stopping GetCookies stream")
            return
        case c, ok := <-cookies:
            if !ok {
                l.Debug().Msg("Cookie channel from exchange is closed, stopping
```

```

GetCookies stream")
    return
}

if err = server.Send(&spec.Cookie{
    UserId:    c.UserId,
    DspId:     c.DspId,
    BuyerUid:  c.BuyerUid,
}); err != nil {
    l.Err(err).Msg("Failed to send cookie")
    return
}

s.metrics.sent.Add(1)
}
}
}

```

Изначально предполагалось, что имеет смысл использовать пул объектов для `spec.Cookie` вместо создания на каждый запрос, но его добавление не показало эффективности. Вероятно, компилятор Go, используя escape-анализ, понимает, что создаваемая структура `spec.Cookie` не выходит за пределы метода, и поэтому память для нее выделяется на стеке. В любом случае, наличие пула не отразилось на производительности, поэтому было решено оставить более простую реализацию, чтобы не вносить дополнительную сложность в код.

### 3.4.5 Создание Dockerfile и другие особенности

Для создания Docker-образа (Таблица 11) используется так называемый “builder pattern”, когда приложение собирается в одном контейнере, а запускается уже в другом. Таким образом, финальный контейнер имеет значительно меньший размер по сравнению с тем, в котором приложение и собирается, и запускается – поскольку он не содержит зависимостей, которые нужны только на этапе сборки. Но поскольку в нем не имеется некоторых библиотек, по умолчанию нужных для запуска приложения, то компилировать приложение необходимо с флагами `CGO_ENABLED=0`, `GOOS=linux`.

Таблица 11 – Создание Docker-образа

```
FROM golang:1.22.2 AS builder
WORKDIR /app
COPY go.mod go.sum ./
RUN go mod download
COPY . .
RUN CGO_ENABLED=0 GOOS=linux go build -o app .

FROM alpine:latest
WORKDIR /root/
COPY --from=builder /app/app .
EXPOSE 8080
EXPOSE 7676
CMD ["/app"]
```

Само приложение запускается простой командой без аргументов, поскольку конфигурация была специально сделана таким образом, чтобы подходить в большинстве случаев. Если нужно изменить какие-то параметры, то можно передать файл конфигурации через Docker volumes.

Аналогичным образом написан Dockerfile для компонента consumer, с незначительными отличиями – поэтому далее он упоминаться не будет. Также в проекте имеется файл для docker compose, позволяющий локально запустить MongoDB для разработки и тестирования.

Также в компоненте был реализован механизм “graceful shutdown”, при котором приложению можно передать команду на завершение работы без потери данных. Для этого используются средства из стандартной библиотеки языка программирования Go: пакеты context и signal. При получении сигнала Interrupt приложение через контекст передает сигнал о завершении работы своим модулям, которые производят сопутствующие действия: например, HTTP-сервер и Streamer закрывают соединения с клиентами.



### 3.5 Реализация компонента consumer

Приложение consumer реализовано со следующей архитектурой, проиллюстрированной на Рисунке 11.

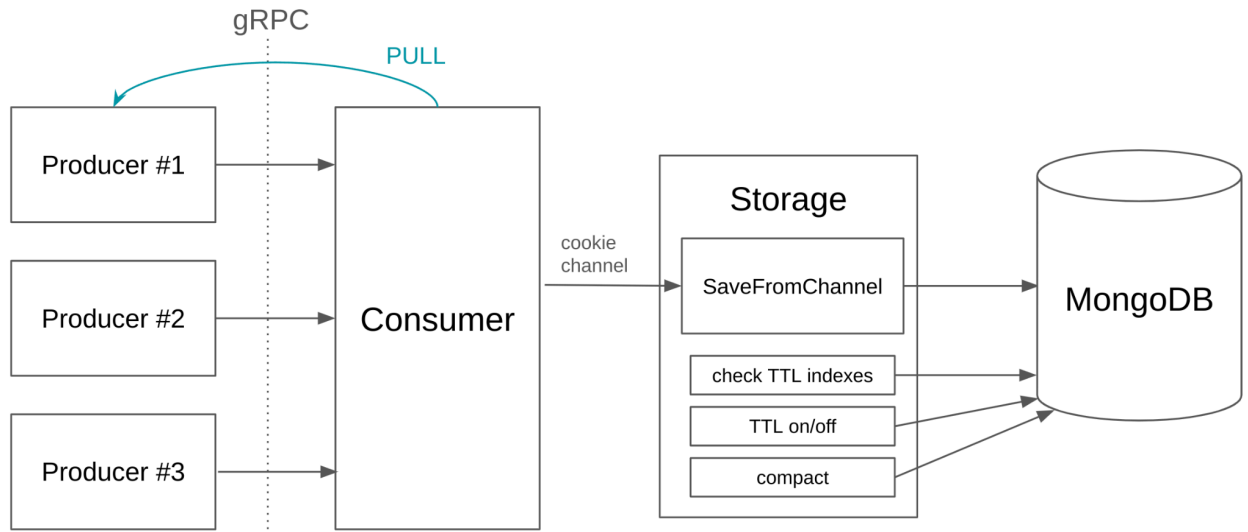


Рисунок 11 – архитектура приложения consumer

Приложение consumer (потребитель) состоит всего из двух модулей, соединенных одним буферизированным каналом:

1. Одноименный Consumer – gRPC-клиент, получающий поток данных с producer, а затем записывающий его в канал;
2. Storage – обертка над драйвером базы данных, читающая куки из канала, затем записывающая их в базу данных.

#### 3.5.1 Разработка модуля Consumer

При запуске Consumer подключается к набору producer, адреса которых указаны в конфигурации, и начинает читать из них данные в методе consume (Таблица 12). При этом, в отличие от компонента producer, при переполнении канала запись блокируется, и новые данные не запрашиваются с сервера, что реализует ранее описанную PULL-модель.

Таблица 12 – метод `consume` модуля `Consumer`

```
func (c *Consumer) consume(ctx context.Context, cl spec.CookieStreamerClient,
p string, ch chan<- *spec.Cookie) {
    l := log.Ctx(ctx).With().Str("producer", p).Logger()

    req := &spec.GetCookiesRequest{Host: c.hostname}

    for {
        select {
        case <-ctx.Done():
            l.Debug().Msg("Context is done, closing consumer")
            return
        default:
        }

        stream, err := cl.GetCookies(ctx, req)
        if err != nil {
            l.Err(err).Msg("Failed to start stream, retrying...")
            time.Sleep(time.Second)
            continue
        }

        l.Info().Msg("Connected to a producer")

    RETRY:
        for {
            cookie, err := stream.Recv()
            if errors.Is(err, io.EOF) {
                l.Err(err).Msg("Producer closed connection (possibly restarting),
retrying...")
                break RETRY
            }

            if err != nil {
                l.Err(err).Msg("Failed to receive, retrying...")
                break RETRY
            }

            select {
            case <-ctx.Done(): // service stopped
                return
            case ch <- cookie:
            }

            c.metrics.received.Inc()
        }
    }
}
```

### 3.5.3 Разработка модуля `Storage`

Модуль `Storage` является оберткой над драйвером базы данных. Он является узким местом всей системы, поскольку запись в базу имеет значительно меньшую пропускную способность, чем отправка по сети.

Как ранее упоминалось в Пункте 3.1, для первой реализации выбрана следующая структура базы данных:

1. Первый уровень – базы данных внутри MongoDB. Они соответствуют номерам DSP, которые содержатся в куки – их количество заранее известно и примерно равняется 20;
2. Второй уровень – коллекции внутри каждой базы данных. Они соответствуют `userId`, но, поскольку область допустимых значений огромна, используются остатки от деления на 256.

Таким образом, сама запись в MongoDB содержит в себе `userId` (`_id`), `buyerId` (`b`) и `timestamp` (`t`), который используется для удаления устаревших данных, потерявших актуальность.

Раз в 10 минут Storage запускает процесс (таблица 13), который выполняет следующие служебные функции:

1. Проверка TTL-индексов в коллекциях – если появились новые коллекции, то в них создается индекс по полю `t`;
2. Включение TTL в ночное время (с 1:00 до 3:00), когда нагрузка является наименьшей;
3. Отключение TTL в остальное время, чтобы не занимать системные ресурсы в период высокой нагрузки;
4. Вызов операции `compact` для оптимизации дерева (с 3:00 до 6:00), что повышает его производительность, так как MongoDB оптимизирует его структуру, избавляясь от удаленных по TTL записей.

Таблица 13 – процесс, выполняющий служебные функции

```
func (s *Storage) handleInternalProcess(ctx context.Context) {  
    if err := s.checkTTLIndexes(ctx); err != nil {  
        log.Ctx(ctx).Err(err).Msg("Failed to checkTTLIndexes")  
    }  
  
    t := time.Now()
```

```

if t.Hour() > 1 && t.Hour() <= 3 {
    if err := s.enableTTL(ctx); err != nil {
        log.Ctx(ctx).Err(err).Msg("Failed to enable ttl")
    }
} else {
    if err := s.disableTTL(ctx); err != nil {
        log.Ctx(ctx).Err(err).Msg("Failed to disable ttl")
    }
}

if t.Hour() > 3 && t.Hour() <= 6 {
    if err := s.compact(ctx); err != nil {
        log.Ctx(ctx).Info().Msg("Failed to compact collections")
    }
}
}
}

```

Изначальная реализация записи в базу представлена в Таблице 14. Здесь куки собираются в пакеты перед отправкой, а затем отправляются в базу раз в 500 миллисекунд. Этот интервал позволяет соответствовать требованиям по времени доставки, и при этом позволяет записывать большое количество куки в базу.

Таблица 14 – Изначальная реализация метода SaveFromChannel

```

func (s *Storage) SaveFromChannel(ctx context.Context, ch chan *spec.Cookie) {
    l := log.Ctx(ctx)

    buf := buffer{}
    start := time.Now()

    for {
        select {
        case <-ctx.Done():
            s.flushBuffer(ctx, buf)
            l.Debug().Msg("Context closed, stopping SaveFromChannel")
            return
        case cookie, ok := <-ch:
            if !ok {
                l.Debug().Msg("Channel closed, stopping SaveFromChannel")
                return
            }

            if !s.mongoEnabled {
                // Dummy storage
                continue
            }

            db := strconv.Itoa(int(cookie.DspId))
            collection := mod(cookie.UserId)

            if buf[db] == nil {
                buf[db] = make(map[string][]any)
            }

```

```

    }

    // Single Storage::SaveFromChannel thread, no synchronization needed
    buf[db][collection] = append(buf[db][collection], bson.D{
        {Key: "_id", Value: cookie.UserId},
        {Key: "b", Value: cookie.BuyerUid},
        {Key: "t", Value: time.Now().Truncate(time.Hour * 24)},
    })

    if time.Since(start) > flushInterval {
        s.flushBuffer(ctx, buf)
        buf = buffer{}
        start = time.Now()
    }
}
}
}

```

## 4 Тестирование и профилирование

### 4.1 Методика тестирования и профилирования

Нагрузочное тестирование данного решения является достаточно нетривиальной задачей, поскольку разработанная система является распределенной, и для полноценного тестирования с большой нагрузкой понадобилось бы использование большого количества машин, что труднодостижимо по финансовым причинам. Поэтому тестирование и профилирование проводились на трех высокопроизводительных машинах, временно предоставленных компанией – а на основе результатов тестирования уже сделан вывод, что система способна справиться с реальной нагрузкой. Подробнее это будет описано ниже.

На каждой из машин были развернуты разные компоненты системы, чтобы минимизировать гонку за ресурсами между компонентами, а также с целью приближения тестовых условий к реальным – то есть использования сети для передачи данных.

Все три машины обладают идентичной конфигурацией, представленной ниже в Таблице 15.

Таблица 15 – Характеристики машин на тестовом стенде

Процессор	Intel Xeon E-2236 (6x3.4 ГГц HT)
Оперативная память	32 ГБ — 4 × 8 ГБ DDR4 ECC
Диск	1000 ГБ SSD NVMe M.2
Материнская плата	X11SCL-F

Сам тестовый стенд имеет архитектуру, представленную на Рисунке 12. На одной машине развернут генератор нагрузки, на двух других – тестируемые компоненты. На локальной машине развернут Prometheus,

собирающий метрики с тестируемых компонентов, а также Grafana, в которой строятся графики по этим метрикам.

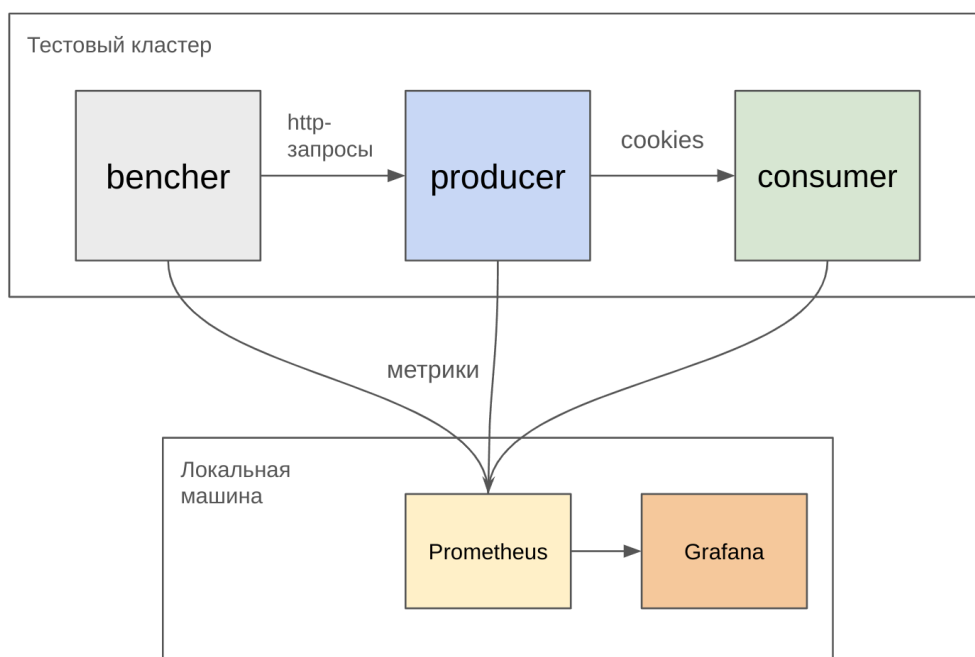


Рисунок 12 – Схема тестового стенда

В качестве генератора нагрузки было испробовано несколько инструментов: `wrk`, `bombardier` и `vegeta`. У каждого из них имелись определенные недостатки, которые не позволяли полноценно нагрузить систему корректными данными. Например, `bombardier` и `vegeta` способны отправлять лишь идентичные запросы, а здесь необходим поток данных, похожий на реальный. В свою очередь, `wrk` способен выполнять скрипт на языке программирования Lua перед каждым запросом, но это сильно влияет на производительность. Наконец, испробовав все три инструмента, удалось создать нагрузку лишь около 500-600 тыс. запросов в секунду (Рисунок 13). Поэтому было решено написать небольшой собственный инструмент, пожертвовав универсальностью (запрос генерируется прямо в коде).

```

root@spb-w3-stathandler:~/bench/bin# ./bombardier -c 512 -d 15s -l -H 'Cookie: uid=12345678901234567890'
'http://cl-hot1-1.moevideo.net:8080?b=10000&d=10'
Bombarding http://cl-hot1-1.moevideo.net:8080?b=10000&d=10 for 15s using 512 connection(s)
[=====] 15s
Done!
Statistics      Avg      Stdev      Max
Reqs/sec    401243.38  25449.63  437109.94
Latency       1.27ms   565.17us  70.48ms
Latency Distribution
  50%      1.25ms
  75%      1.37ms
  90%      1.50ms
  95%      1.64ms
  99%      2.43ms
HTTP codes:
  1xx - 0, 2xx - 6012786, 3xx - 0, 4xx - 0, 5xx - 0
  others - 0
Throughput:   75.31MB/s

```

### Рисунок 13 – Пример работы утилиты bombardier

Собственный инструмент был реализован с помощью всё той же библиотеки `fasthttp`, что позволило достичь высокой производительности клиента. Утилита создает несколько подключений к серверу, пользуясь специальным клиентом `fasthttp.PipelineClient`, который отправляет все запросы через одно TCP-подключение. На это подключение назначается большое количество горутин, которые генерируют и отправляют запросы.

Также в инструмент интегрирован опциональный ограничитель скорости запросов, а также возможность установки длительности бенчмарка либо суммарного количества отправленных запросов (Рисунок 14).



```
root@spb-w3-stathandler:~/bench# ./bench -h
Usage of ./bench:
  -c int
      Number of connections (default 10)
  -d duration
      Duration of benchmark
  -l int
      Limit requests per second
  -r int
      Number of requests (approximate)
  -t int
      Number of concurrent threads per connection (default 12)
  -url string
      URL of targeted server (default "http://cl-hot1-1.moevideo.net:8080")
  -v
      Verbose mode
```

Рисунок 14 – Флаги разработанной утилиты

Максимальная скорость генерации запросов у разработанного инструмента составляет как минимум 940 тысяч запросов в секунду (Рисунок 15), что вполне достаточно для полноценного нагрузочного тестирования системы.

```
root@spb-w3-stathandler:~/bench# ./bench -t 512 -c 512 -d 5s
2024/05/08 13:58:17 Target URL: http://cl-hot1-1.moevideo.net:8080
2024/05/08 13:58:17 Connections: 512
2024/05/08 13:58:17 Threads (goroutines) per connection: 512
2024/05/08 13:58:17 Rate limit: unlimited
2024/05/08 13:58:17 Starting benchmark at 2024-05-08 13:58:17
2024/05/08 13:58:17 Duration: 5s
2024/05/08 13:58:18 Progress: sent = 758186, avg rps = 757368 r/s, elapsed = 1s
2024/05/08 13:58:19 Progress: sent = 1766356, avg rps = 882641 r/s, elapsed = 2s
2024/05/08 13:58:20 Progress: sent = 2704223, avg rps = 901031 r/s, elapsed = 3s
2024/05/08 13:58:21 Progress: sent = 3642084, avg rps = 910175 r/s, elapsed = 4s
2024/05/08 13:58:22 Benchmark finished
2024/05/08 13:58:22 Sent total of 4840394 requests
```

Рисунок 15 – Пример работы разработанной утилиты

Более того, выше этого значения показатели не поднимаются, так как узким местом становится уже гигабитная сетевая карта, что видно (Рисунок 16) по показателям утилиты btop (“upload”).

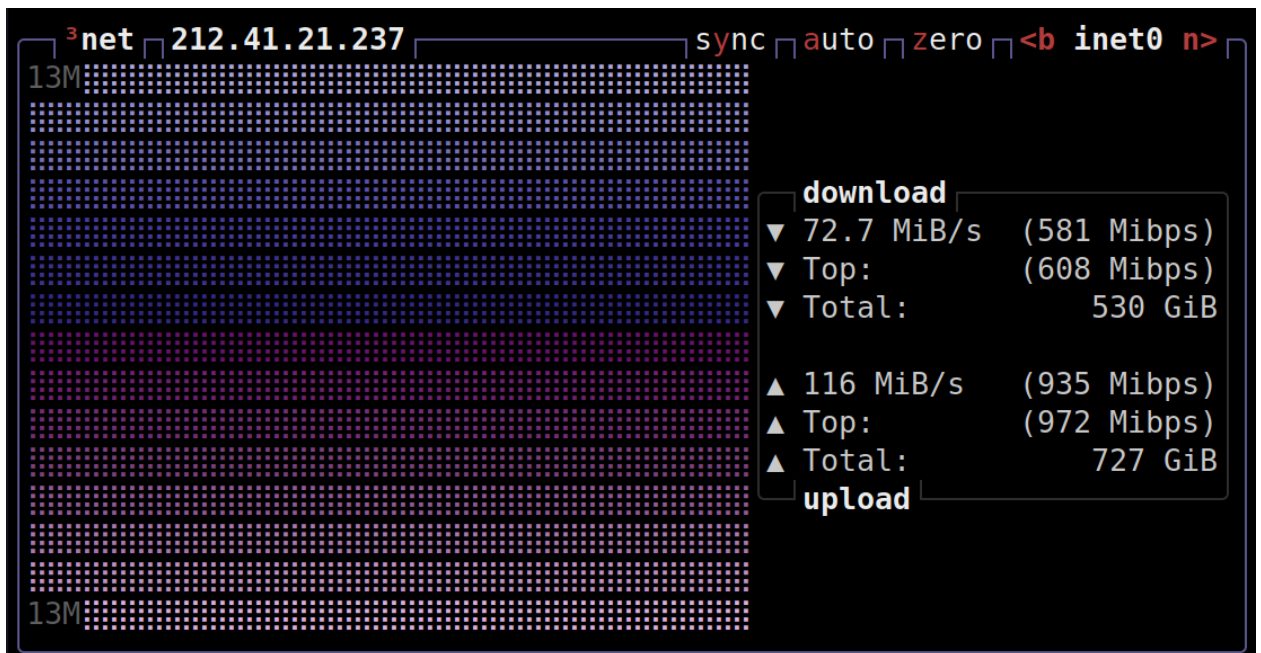


Рисунок 16 – показатели из утилиты btop

## 4.2 Тестирование передачи по сети

Для начала была протестирована пропускная способность канала producer-consumer. Для этого consumer был запущен в режиме без записи в базу, в режиме приема данных. Было выявлено, что вся цепочка имеет пропускную способность около 860 тыс. запросов в секунду, при которой она работает исправно (Рисунок 17, графики совпадают).

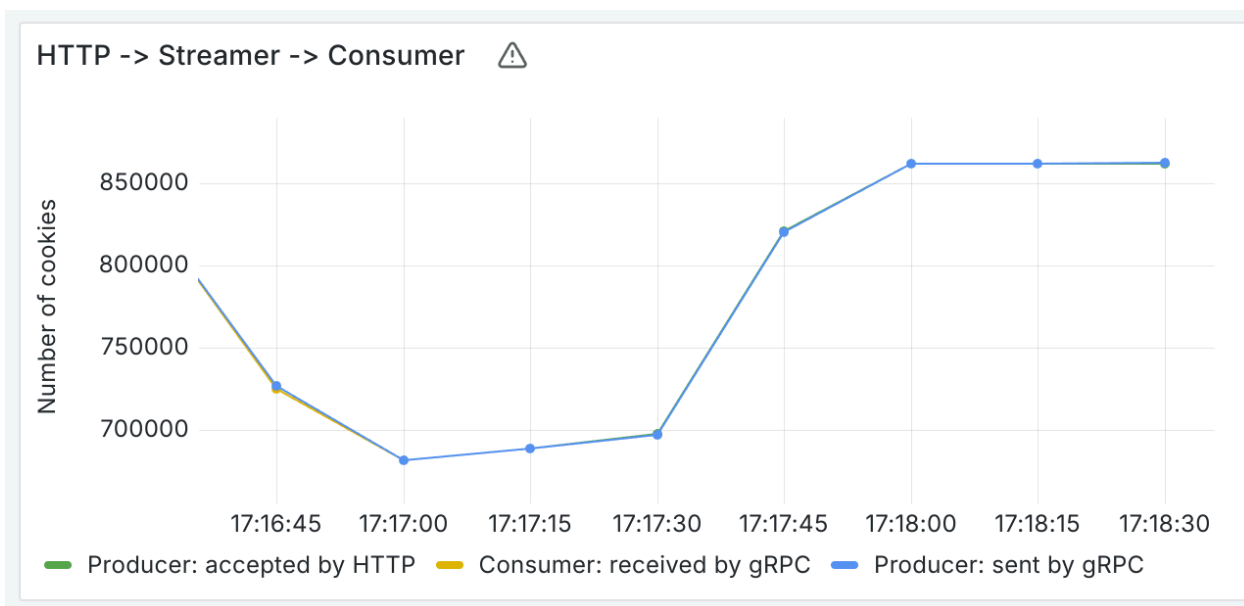


Рисунок 17 – показатели при 860 тыс. запросов в секунду

При этом количество потерянных (удаленных из-за переполнения очереди) куки остается равным нулю (Рисунок 18).

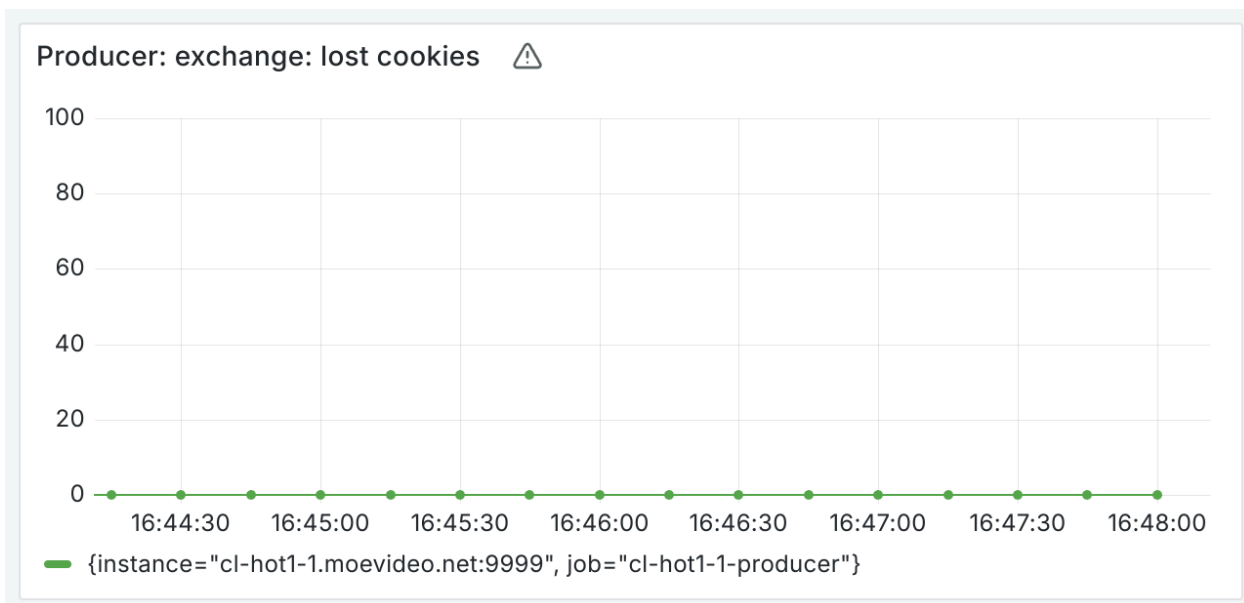


Рисунок 18 – количество потерянных куки

Предел был достигнут при показателе около 870 тыс. запросов в секунду. На графиках (Рисунок 19) появилось расхождение отправленных и полученных данных, а внутренняя очередь стала переполняться.

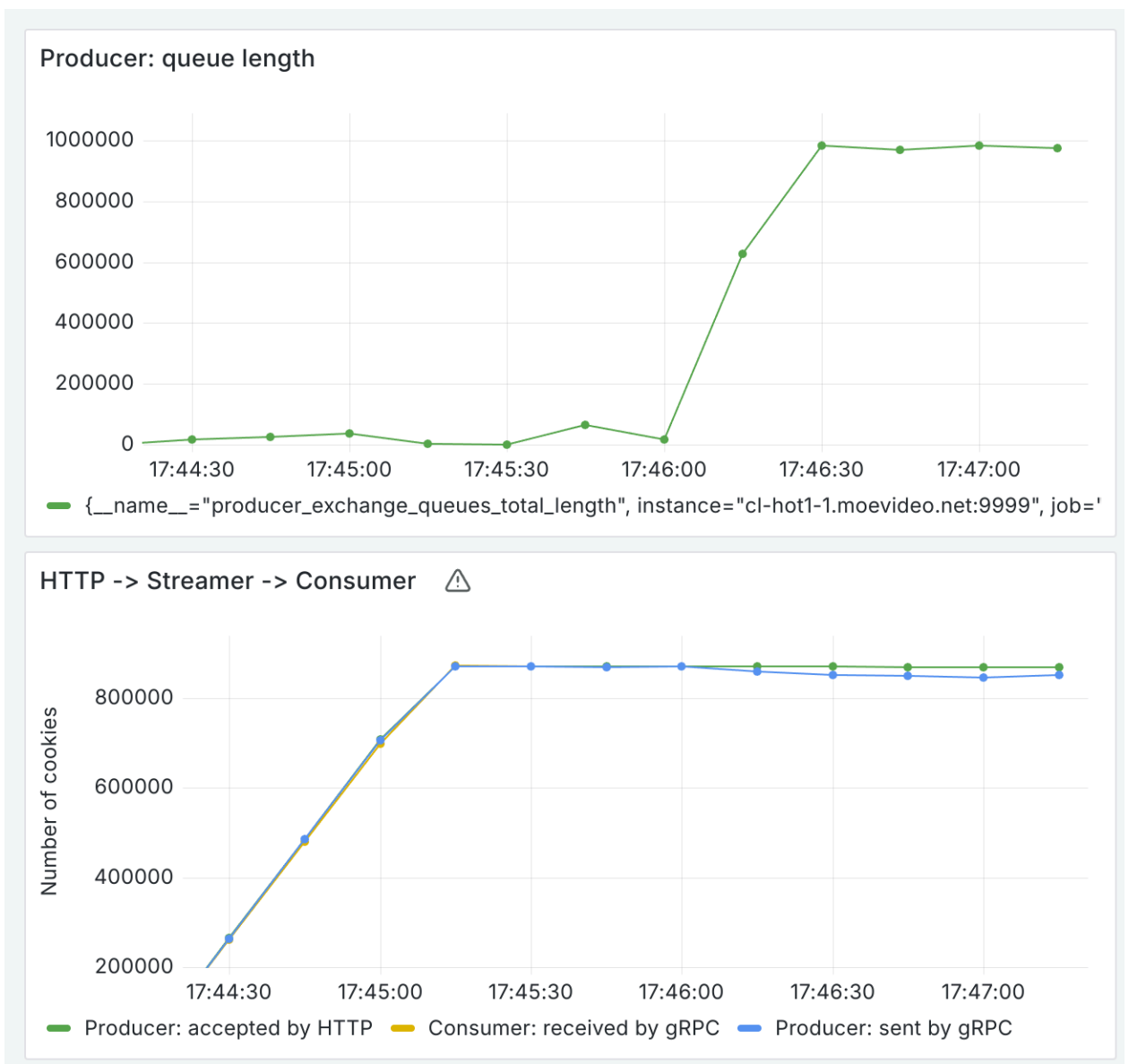


Рисунок 19 – размер очереди на producer и объем данных при 870 тыс. RPS

При этом по показателям утилиты `btcp` можно заключить, что проблема не в самом producer, а в сетевой карте, установленной на сервере, поскольку процессор был загружен не полностью (рисунок 20).

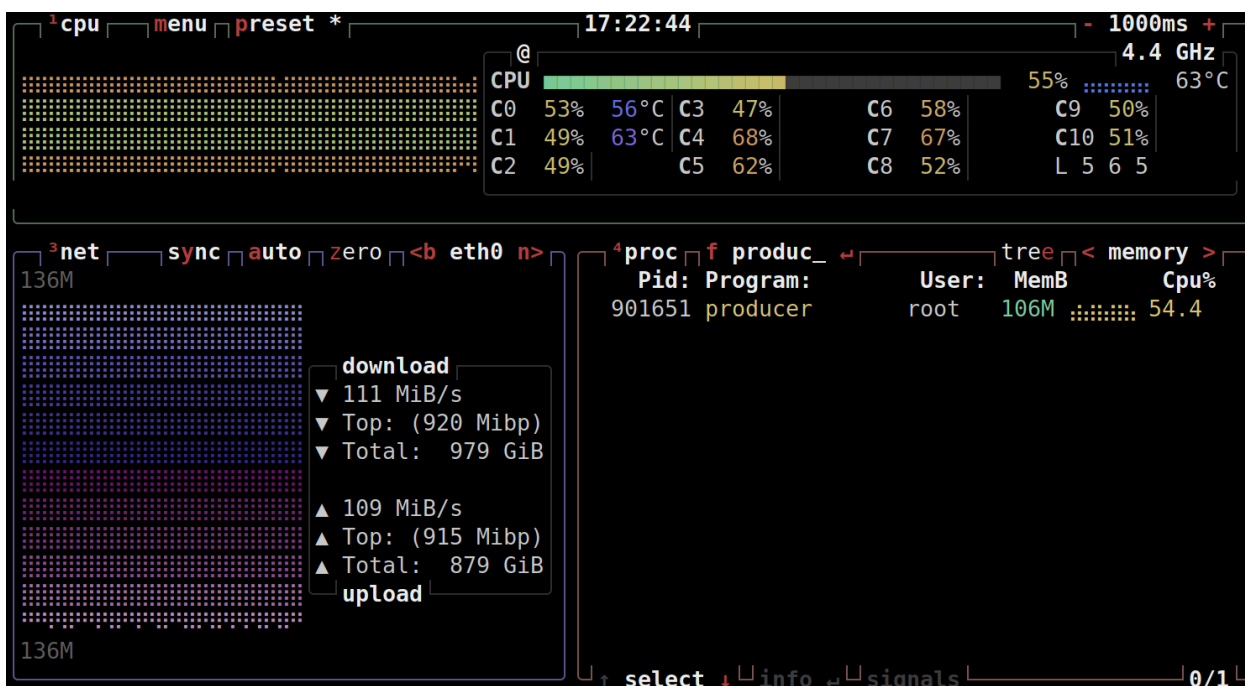


Рисунок 20 – показатели в btop

Здесь, к сожалению, невозможно было получить точные данные по максимальной пропускной способности одного экземпляра producer, поскольку сделать это не позволяли сетевые карты в машинах. Однако сделан вывод, что определенное количество экземпляров будет способно справляться с нагрузкой, и тогда пропускная способность будет соответствовать требуемой. На промышленном кластере нужно развернуть некоторое произвольное количество экземпляров producer (например, 3) и посмотреть, справятся ли они с нагрузкой. Если нет, то увеличивать их количество до необходимого.

Также было проведено профилирование producer через rprof, но узких мест в приложении обнаружено не было – большая часть процессорного времени и оперативной памяти тратилась на приём и отправку данных по сети.

### 4.3 Тестирование записи в базу данных

Далее было проведено тестирование с включенным режимом записи в базу данных. Изначальная проверка была проведена при нагрузке в 10 тыс. запросов в секунду. Было замечено (Рисунок 21), что модуль в целом справляется с записью в базу, но внутренняя очередь периодически заполняется данными, причем иногда достаточно большим их количеством.



Рисунок 21 – показатели при 10 тыс. запросов в секунду

Для исправления этой проблемы прямой вызов `flushBuffer` по истечению интервала был заменен на отправку данных через канал в дополнительную горутину, которая вызывала `flushBuffer` в отдельном потоке, не блокируя первую горутину, которая сразу начинала собирать новый пакет из новых данных. После этого куки перестали задерживаться в очереди.

На 20 тыс. запросов в секунду снова начала возникать проблема, при которой в очереди задерживались куки (рисунок 22). Также было замечено, что некоторые пакеты записываются дольше 1 секунды. Дальнейшее увеличение нагрузки показало, что пропускная способность не соответствует требуемой.



Рисунок 22 – показатели при 20 тыс. запросов в секунду

С помощью инструмента `pprof` было проведено профилирование `consumer` по процессорному времени, в результате чего получена диаграмма затраченного на каждый вызов процессорного времени (Рисунок 23). Было установлено, что узким местом `consumer` является только метод `Insert` записи в MongoDB – на него тратится большая часть времени работы программы.

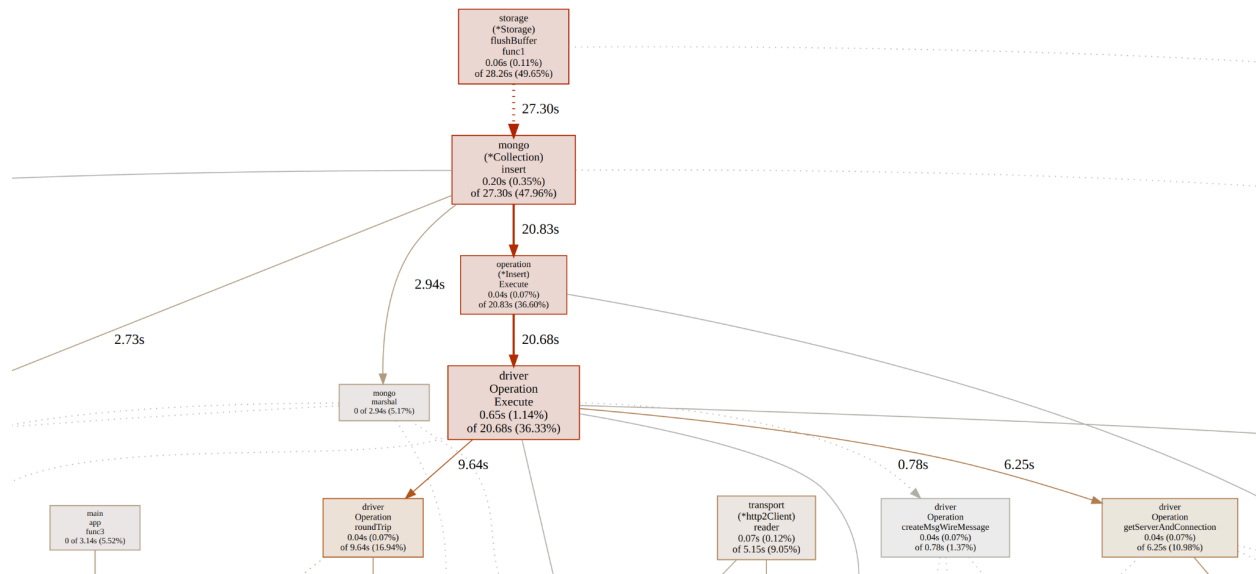


Рисунок 23 – часть диаграммы затраченного процессорного времени

Далее были предприняты попытки изменения структуры программы. Например, одна записывающая горутина была заменена на несколько горутин на каждую базу данных – но это не оказало особого влияния на скорость записи. Также для соединения с MongoDB вместо TCP был использован протокол Unix Domain Sockets, однако, как ни странно, это тоже не оказало никакого заметного влияния на скорость записи.

Также были внесены различные изменения конфигурации MongoDB, например: `journalCommitInterval` (интервал синхронизации журнала), `wiredTigerCacheSizeGB` (размер кэша внутреннего движка базы данных), отключено сжатие сообщения (`networkMessageCompressors`).



К сожалению, никакие из этих изменений не улучшили производительность в достаточной мере. Также из-за постоянных перезапусков тестового стенда было замечено, что в случае пересоздания контейнера с MongoDB первые несколько **минут** работы consumer записывал пакеты размером около 50 куки в течение нескольких секунд каждый. Это происходило из-за того, что создание 20 баз данных с 256 коллекциями занимало значительное время. Перенос инициализации коллекций в начало работы программы лишь увеличил время запуска приложения, и не улучшил производительность во время работы.

В связи с этим было решено отказаться от разбиения MongoDB на коллекции, и оставить лишь разбиение по базам, каждая из которых все так же соответствует DSP, а внутри имеет одну коллекцию – это второй из предложенных в Главе 2 вариантов.

Новое решение сразу же показало кратную эффективность по сравнению с предыдущим (рисунок 24). Нагрузка в 50 тыс. запросов в секунду с легкостью записывается в базу менее чем за 600 мс (показатель `took_total`, равный сумме времени записи и интервала сбора пакета). Значит, средняя нагрузка в течение дня точно будет выдерживаться разработанной системой.



Рисунок 24 – показатели при 50 тыс. запросов в секунду

Также был установлен интервал сбора пакета в 200 мс, чтобы сократить время доставки (Рисунок 25).

```
5:39PM DBG Flushed buffer inserted=10046 took=32.119894ms took_total=232.12028ms
5:39PM DBG Flushed buffer inserted=10019 took=33.62865ms took_total=233.62908ms
5:39PM DBG Flushed buffer inserted=10027 took=31.894994ms took_total=231.895387ms
5:39PM DBG Flushed buffer inserted=10039 took=33.289115ms took_total=233.289471ms
5:39PM DBG Flushed buffer inserted=10052 took=32.152303ms took_total=232.152608ms
5:39PM DBG Flushed buffer inserted=10035 took=31.225661ms took_total=231.226104ms
5:39PM DBG Flushed buffer inserted=10482 took=36.965211ms took_total=236.965928ms
5:39PM DBG Flushed buffer inserted=10066 took=38.414478ms took_total=238.414871ms
5:39PM DBG Flushed buffer inserted=10448 took=37.333429ms took_total=237.333936ms
5:39PM DBG Flushed buffer inserted=10076 took=30.226513ms took_total=230.226874ms
```

Рисунок 25 – Время записи пакетов при 200 мс и 50 тыс. запросов в секунду

Предел стабильной работы был найден на уровне около 120 тыс. запросов в секунду (Рисунок 26). На более интенсивной нагрузке в определенный момент начинались проблемы со скоростью записи. Однако этого показателя уже достаточно для соответствия требованиям.



Рисунок 26 – показатели при 120 тыс. запросов в секунду

Далее была проверена работа при пиковой нагрузке. При 200к процессор сильно перегружается, и для записи требуется уже больше 2 секунд. В очереди могут иногда задерживаться куки, но это приемлемо по условиям задачи. Однако долго такая производительность не сохраняется – запись начинает замедляться через 6-7 минут. Тем не менее, приложение не отключается, и продолжает постепенно обрабатывать новые данные, поступающие в очередь – как и требуется.

С помощью увеличения параметра `wiredTigerCacheSizeGB` (размера кэша движка `WiredTiger`) со зрением по умолчанию (половина доступной оперативной памяти, в данном случае 16 ГБ) на 24 ГБ удалось увеличить время стабильной работы при нагрузке в 200 тыс. запросов в секунду до 10 минут (Рисунок 27).



Рисунок 27 – работа при 200 тыс. запросов в секунду с кэшем в 24 ГБ

После уменьшения нагрузки до 100 тыс. запросов в секунду приложения возвращается в стабильное состояние (Рисунок 28).



Рисунок 28 – возврат в стабильное состояние при 100 тыс. запросов в секунду

#### 4.4 Выводы

По результатам проведенного нагрузочного тестирования было заключено, что разработанное решение соответствует установленным требованиям к средней нагрузке в 40 тыс. запросов в секунду, выдерживая как минимум 50 тыс. запросов в секунду (по последнему бенчмарку – 120 тыс.), а также к пиковой нагрузке в 200 тыс. запросов в секунду в течение 10 минут. Время оптимальной работы приложения при пиковой нагрузке зависит от размера кэша движка WiredTiger, который устанавливается в

конфигурации MongoDB, либо по умолчанию равняется половине доступной оперативной памяти. Таким образом, при необходимости увеличения времени работы достаточно будет развернуть решение на сервере, обладающем большим количеством оперативной памяти.

## ЗАКЛЮЧЕНИЕ

В процессе выполнения выпускной квалификационной работы был произведен обзор предметной области, в рамках которого описана проблема, для которой разрабатывается решение, рассмотрены существующие аналоги (выделены преимущества и недостатки), обоснована необходимость разработки собственного решения.

Далее был спроектирован и реализован первый прототип собственного решения, в соответствии с исходными данными и заявленными требованиями. Проведено нагрузочное тестирование и профилирование этого прототипа на высокопроизводительном серверном оборудовании, а затем произведен анализ результатов тестирования и профилирования.

На основе анализа результатов были внесены изменения, устраняющие найденные недостатки, и увеличивающие производительность системы. Далее снова было проведено нагрузочное тестирование, с помощью которого подтверждено соответствие требованиям по выдерживаемой нагрузке и пропускной способности.

В итоге была разработана система, полностью соответствующая заявленным требованиям, а также имеющая потенциал к дальнейшему расширению. На момент завершения написания пояснительной записки к ВКР реализованное решение было развернуто в промышленном кластере компании, который представляет из себя:

1. 3 машины с компонентом `producer`, которые полностью справляются с реальной нагрузкой;
2. 67 машин, на каждой из которых работает приложение `consumer` и база данных MongoDB.



Все перечисленные компоненты оркестрируются Kubernetes, и в течение дня кластер обрабатывает от 20 до 200 тыс. запросов на синхронизацию куки в секунду. Общий объем каждого экземпляра MongoDB составляет около 180 Гб, а каждую неделю поступает около 100 Гб новых данных, хранящихся в течение двух недель.

Развернутое в кластере решение функционирует без критических ошибок и неполадок, обеспечивая работу рекламной сети.

Реализованная система имеет потенциал к улучшению пропускной способности и времени доставки. К примеру, можно рассмотреть альтернативные базы данных, а также воспользоваться оптимизацией на основе результатов профилирования (Profile-Guided Optimization или PGO), поддержка которой недавно появилась в языке программирования Go.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Описание протокола OpenRTB [Электронный ресурс]. – URL: <https://www.iab.com/wp-content/uploads/2016/03/OpenRTB-API-Specification-Version-2-5-FINAL.pdf> (дата обращения 09.03.2024)
2. Документация Google по сопоставлению куки [Электронный ресурс]. – URL: <https://developers.google.com/authorized-buyers/rtb/cookie-guide?hl=ru> (дата обращения 12.03.2024)
3. Как запускать RabbitMQ в Docker [Электронный ресурс]. – URL: <https://habr.com/ru/companies/slurm/articles/704208/> (дата обращения 16.03.2024)
4. RabbitMQ Performance Measurements, part 2 [Электронный ресурс]. – URL: <https://www.rabbitmq.com/blog/2012/04/25/rabbitmq-performance-measurements-part-2> (дата обращения 21.03.2024)
5. Сравнительный анализ Apache Kafka и RabbitMQ [Электронный ресурс]. – URL: <https://apni.ru/article/5144-sravnitelnij-analiz-apache-kafka-i-rabbitmq> (дата обращения 22.03.2024)
6. Dissecting message queues [Электронный ресурс]. – URL: <https://bravenewgeek.com/dissecting-message-queues/> (дата обращения 22.03.2024)
7. Документация протокола NATS [Электронный ресурс]. – URL: <https://docs.nats.io/reference/reference-protocols/nats-protocol> (дата обращения 22.03.2024)

8. Модели получения в NATS [Электронный ресурс]. – URL: [https://docs.nats.io/running-a-nats-service/nats\\_admin/jetstream\\_admin/consumers](https://docs.nats.io/running-a-nats-service/nats_admin/jetstream_admin/consumers) (дата обращения 22.03.2024)
9. Kernighan B., Donovan A. The Go Programming Language. – Addison-Wesley Professional, 2015. – С. 13
10. GitHub-репозиторий библиотеки fasthttp [Электронный ресурс]. – URL: <https://github.com/valyala/fasthttp> (дата обращения 22.03.2024)
11. Comparing JSON and MessagePack [Электронный ресурс]. – URL: <https://thephp.website/en/issue/messagepack-vs-json-benchmark/> (дата обращения 22.03.2024)