Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 23.М04-мм

Реализация системы хранения телеметрии для опорной сети

СМИРНОВ Александр Андреевич

Отчёт по преддипломной производственной практике
в форме «Решение»

Научный руководитель:
доцент кафедры системного программирования, к. ф.-м. н., Луцив Д. В.

Консультант:
Эксперт по разработке ПО,
ООО «Ядро Центр Технологий Мобильной Связи», Иргер А. М.

Санкт-Петербург
2025

Оглавление

Введение

Современные высоконагруженные системы зачастую создаются на основе микросервисной архитектуры, становясь распределёнными [10, 11]. Такая архитектура позволяет добиться хорошей масштабируемости и отказоустойчивости, позволяет удобно внедрять новую функциональность и следовать практикам CI/CD. Подобные системы, как правило, обрабатывают большое количество запросов, при этом в процессе выполнения каждого запроса задействуются несколько узлов системы. Эта особенность усложняет отладку: при обработке запроса каждый узел генерирует собственный набор логов, что затрудняет отслеживание пути этого запроса и сбор информации, касающейся его выполнения.

Для решения этой проблемы и облегчения отладки инженерами было введено [15] понятие трейса — сущности, которая представляет путь одного запроса в распределённой системе через её узлы от начала и до конца. Такие трейсы система может генерировать в дополнение к логам. В совокупности логи, трейсы и другая информация, которая позволяет анализировать поведение системы, называются телеметрией [15].

На базе микросервисной архитектуры, в том числе, построена система опорной сети. Система изображена на рисунке 1. Она представляет собой распределённый набор сервисов, которые общаются между собой посредством REST API. Система постоянно обрабатывает запросы, которые поступают от базовых станций и других компонентов сети. В терминах опорной сети такие запросы называются процедурами. Понятие трейса можно удобно отобразить на понятие процедуры, что позволяет применить уже разработанные стандарты для генерации системой телеметрии. такой подход значительно упрощает отладку системы и анализ проблем во время эксплуатации.

Предметная область телеметрии последние несколько лет активно развивается. На рынке представлены [13] различные решения, которые позволяют работать с такими данными. Однако эти решения рассчитаны на широкий круг систем и пользователей, из-за чего вынуждены

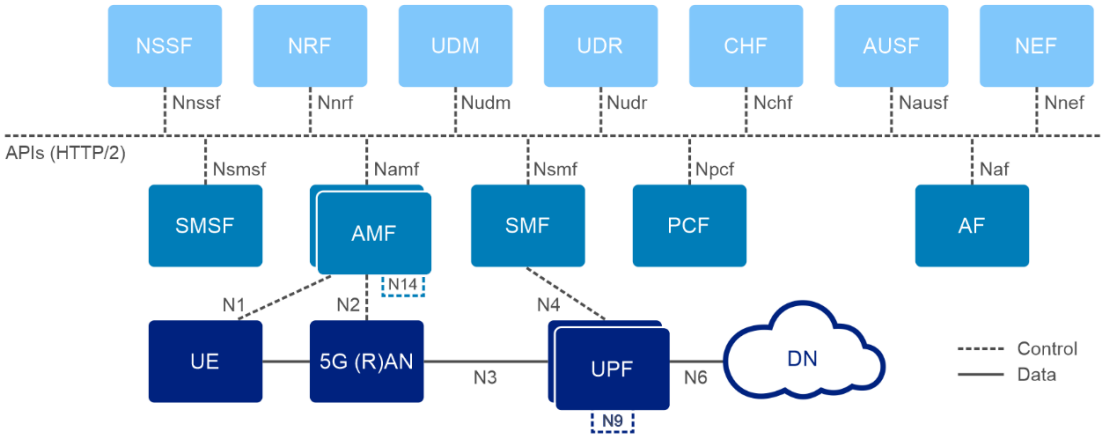

Рис. 1: Архитектура опорной сети
Источник: https://infohub.delltechnologies.com/p/the-5g-core-network-demystified

абстрагироваться от какой-либо предметной области. Таким образом, было принято решение разработать собственный инструмент для сбора, хранения и анализа телеметрии, чтобы иметь возможность реализовать необходимые сценарии работы, а также интегрировать его с другими системами компании.

1 Постановка задачи

Целью работы является разработка системы для сбора, хранения и анализа телеметрии. Для её выполнения были поставлены следующие задачи:

1. выполнить обзор предметной области телеметрии;
2. выявить и сформулировать требования к системе;
3. выбрать технологии и разработать архитектуру системы;
4. реализовать систему;
5. произвести апробацию работы системы совместно с опорной сетью.

2 Обзор предметной области

2.1 Observability: основные понятия

Введём ряд определений, необходимых для понимания предметной области.

Определение 1. ***Observablity*** *(наблюдаемость) — в терминологии распределённых систем это возможность понимать внутренее состояние системы на основании её выходных данных. [15, 6]*

Определение 2. *Выходные данные системы, содержащие информацию о её внутреннем состоянии, называют [15]* ***телеметрией****.*

Телеметрия существенно упрощает работу как при отладке системы, так и при поиске и устрании проблем в уже работающей системе. Позволяет ответить на вопрос “Почему это происходит в системе?”.

Телеметрия обычно состоит из трёх видов данных: трейсов, метрик и логов. Рассмотрим каждый из них подробнее.

Лог — некоторое сообщение [17] от системы, содержащее временную метку. Это один из первых и наиболее популярных видов телеметрии. Однако в логах, как правило, не содержится информация о контексте, в котором они были сгенерированы, из-за чего их нельзя связать с конкретным запросом к системе. Поэтому если в распределённой системе проблема возникла в одном из компонентов из-за упущенной проблемы в другом, то такую цепочку становится тяжело распутать.

Трейс — это некоторая совокупность [17] информации о запросе, который обрабатывается системой. В отличие от логов, трейсы содержат информацию о том, как именно конкретный запрос выполнялся внутри системы от начала и до конца (например, REST-запрос на получение какого-то ресурса). Трейс содержит информацию о том, какие операции над запросом были выполнены, через какие узлы системы (например, компоненты облачного приложения) он прошёл.

Спан — это составляющая [17] трейса. Спан представляет информацию об некоторой операции, которая была выполнена при обработке

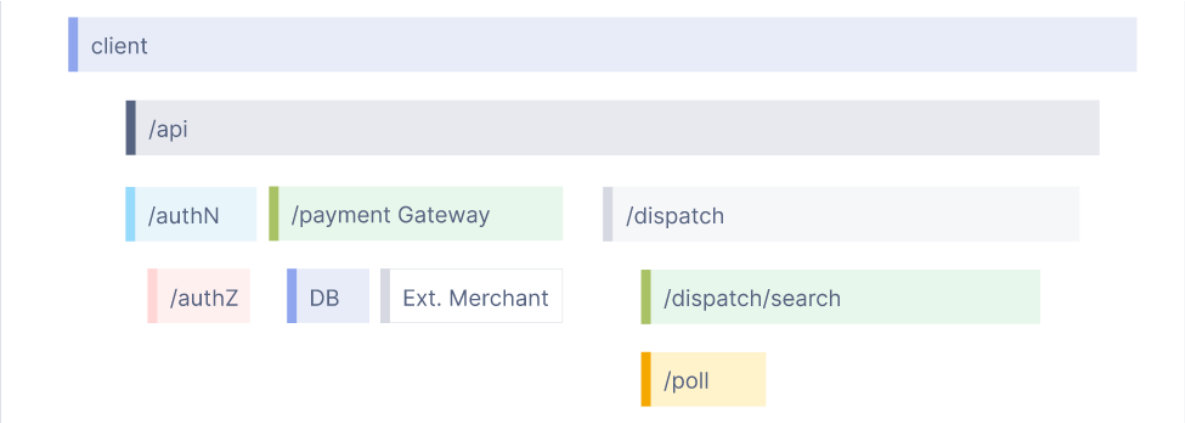

Рис. 2: Пример визуализации трейса
Источник: https://opentelemetry.io/docs/concepts/observability-primer

системой запроса (например, запрос бэкенда к базе данных в рамках выполнения REST-запроса). Спан содержит время начала операции и её продолжительность, а также различные метаданные, описывающие операцию (например, имя узла, идентификатор пользователя, параметры запроса). Также спан содержит информацию о результате выполнения (успех/ошибка).

По своей сути трейс представляет собой совокупность всех спанов, относящихся к одному запросу. Пример трейса изображён на рисунке 2. Минимальный трейс состоит из одного спана, называемного коренным. Коренной спан может быть только один, он представляет весь запрос от начала до конца. Примером коренного спана может служить спан, представляющий вызов некоторого коллбэка `onHttpRequest()`.

В случае опорной сети с помощью трейсов удобно представлять процедуры. Процедура в терминах опорной сети — это запрос к системе, затрагивающий несколько её компонент. Пример такого запроса — процедура регистрации телефона в сети. От базовой станции поступает запрос на регистрацию мобильного устройства, после чего он проходит через несколько узлов сети, выполняет нужные действия и завершается. Благодаря трейсам и содержащейся в спанах метаинформации визуализация такого запроса весьма полезна при отладке, а также поиске проблем в процессе эксплуатации.

2.2 Обзор существующих решений

С ростом популярности телеметрии появилось множество observability-решений от разных компаний, предоставляющих собственные инструменты и форматы данных телеметрии. Из-за желания стандартизировать такой подход были созданы проекты OpenTracing [18] и OpenCensus [14], которые впоследствие объединились, в результате чего был создал фреймворк OpenTelemetry.

OpenTelemetry [15] — это открытый стандарт и фреймворк для обеспечения наблюдаемости в приложениях или системах. Стандарт содержит спецификации для различных типов данных телеметрии (трейсов, метрик и логов), описывает протоколы передачи этих данных, подходы к инструментации кода (иными словами, как писать код, генерирующий телеметрию), различные библиотеки и многое другое.

Одна из важных особенностей стандарта заключается в том, что он позиционируется как vendor-agnostic, то есть он не зависит ни от какой компании. Наоборот, как проприетарные, так и open-source observability-инструменты поддерживают этот стандарт. Это даёт возможность разработчикам приложения единожды инструментировать код, а после чего уже выбирать инструмент под свои нужды.

Рассмотрим одну из проблем, которую решает стандарт. С ростом количества observability-решений от разных вендоров росло и количество спецификаций и протоколов, по которым передаётся телеметрия. Кроме того, для для хранения телеметрии могут использоваться разные хранилища (например, разные базы данных). Поэтому для того, чтобы сохранять телеметрию в нужной базе данных, нужно написать собственный парсер используемого протокола и собственное приложение, сохраняющее данные в базу с нужной схемой. Для решения этой проблемы фреймворк OpenTelemetry предоставляет приложение, называемое OpenTelemetry Collector [16]. Приложение представляет собой прокси, который позволяет принимать телеметрию в разных форматах, изменять её и конвертировать в другие форматы. Конфигурация таких преобразований настраивается с помощью yaml-фала и не требует на-

писания какого-либо кода. Приложение написано на Go, исходный код открыт. Важно отметить, что Collector нигде не хранит телеметрию, а выступает только промежуточным узлом при её обработке.

Хранением, обработкой и визуализацией данных занимаются **бэкенды телеметрии**. Именно в них OpenTelemetry Collector обычно направляет данные. Как правило, бэкенд состоит из хранилища телеметрии (это может быть база данных или объектное хранилище) и веб-интерфейса, на котором телеметрия отображается (например, в виде дерева спанов, графиков). Бэкенды не поставляются фреймворком OpenTelemetry, и каждый вендор реализует их по-своему, предлагая различную функциональность. Однако такие бэкенды часто поддерживают стандарт OpenTelemetry и способны принимать телеметрию в соответствующем стандарту формате. На данный момент существует множество различных как проприетарных, так и open-source бэкендов. Рассмотрим некоторые из них.

2.2.1 Jaeger

Jaeger [8] — один из первых и наиболее известных инструментов для обеспечения наблюдаемости. Инструмент написан на Go, исходный код открыт и доступен под лицезии Apache-2.0. Инструмент собирает спаны и хранит их в базе данных. Также он позволяет визуализировать трейсы и спаны в удобном графическом интерфейсе, который показан на рисунке 3. Однако Jaeger не поддерживает хранение и визуализацию другого типа данных телеметрии —логов. Таким образом, инструмент удобен для быстрого развёртывания и поиска проблем с помощью трейсов, но проигрывает по функциональности более современным инструментам, о которых пойдёт речь далее.

2.2.2 Grafana observability stack

Набор инструментов от компании Grafana Labs [5]. Состоит из четырёх отдельных продуктов.

- Grafana — веб приложение, представляющее собой инструмент

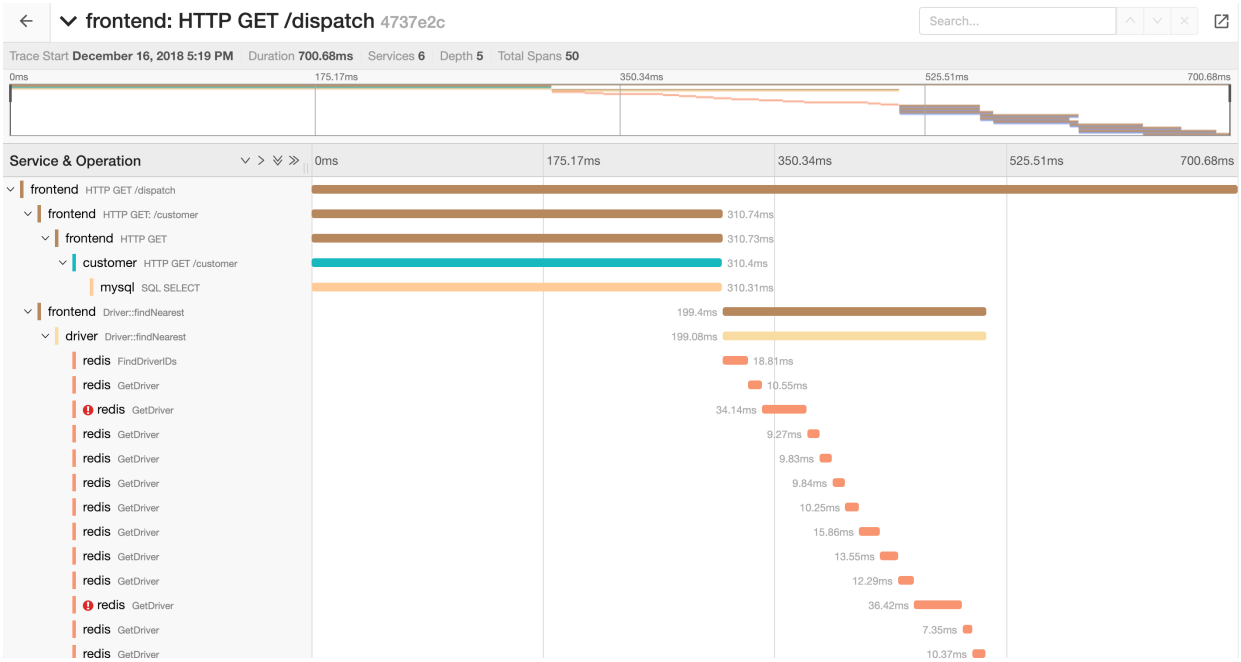

Рис. 3: Пользовательский интерфейс Jaeger
Источник: https://www.jaegertracing.io/docs/1.18/frontend-ui

для визуализации различных данных. Тесно используется с time-series базами данных (например, Prometheus) и позволяет с помощью графиков и диаграмм в режиме он-лайн визуализировать метрики. Позволяет создавать ”приборные панели” (dashboards), на которых можно разместить нужные элементы визуализации. Также поддерживает визуализацию трейсов, аналогично интерфейсу Jaeger, и визуализацию логов.

- Grafana Tempo – бэкенд для обработки трейсов. Инструмент собирает трейсы от различных узлов и хранит их. Для визуализации данных интегрируется с Grafana.

- Grafana Loki – бэкенд для обработки логов. Собирает и хранит логи.

- Grafana Mimir – инструмент, реализующий долговременное хранилище метрик для БД Prometheus.

В связке эти инструменты представляют мощную систему для мониторинга и обеспечения наблюдаемости. Эта система позволяет собирать, хранить и визуализировать все типы даннных телеметрии. Кроме того,

она даёт возможность коррелировать различные виды телеметрии, относящиеся к одному контексту: например, показать все логи, которые были сгенерированы во время выполнения определённого трейса.

Каждый из продуктов Grafana stack распространяется в двух версиях. Первая версия продуктов имеет открытый исходный код, доступный по лицензии AGPLv3. Вторая версия распространяется под Enterprise лицензией, исходный код закрыт.

Стоит отметить, что инструмент визуализации Grafana — вполне самостоятельная и мощная платформа. Она не ограничивает пользователей использованием Grafana Stack, позволяет визуализировать данные из различных источников. Так, например, она может визуализировать данные напрямую из различных БД, а также сторонних систем сбора трейсов и логов.

2.2.3 SigNoz

SigNoz [22] — тоже система обеспечения наблюдаемости с открытым исходным кодом, написана на Go. Большая часть кода распространяется под лицензией Apache-2.0, однако некоторые компоненты доступны только с Enterprise лицензией. В отличие от Grafana Stack, система обрабатывает все типы данных одним инструментом, за счёт чего она более компактна и проста в развертывании. Систему можно развернуть как локально, так и воспользоваться облачным сервисом, который предоставляет компания-разработчик. Система поддерживает визуализацию трейсов, метрик и логов, создание “приборных панелей” с различной информацией. Данные телеметрии хранятся в базе данных Clickhouse [3], куда их записывает особым образом сконфигурированный OpenTelemetry Collector. Благодаря Clickhouse система позволят выполнять различные сложные запросы к хранящимся данным: например, фильтрация спанов от конкретного сервиса, группировка по атрибутам и агрегирование.

2.2.4 Выводы

Существующие инструменты, в особенности Grafana Stack и Signoz, предлагают богатую функциональность в области хранения и обработки телеметрии. Однако полная функциональность этих двух инструментов доступна только по Enterprize лицензии. Кроме того, все инструменты ориентированы на работу с широким кругом разнообразных облачных приложений и, соответственно, не имеют привязки к области телекоммуникационных систем. Таким образом, было принято решение реализовать собственный инструмент.

В процессе исследования было отмечено, что существующие решения объединяет схожая архитектура. В обобщённом виде она приведена на рисунке 4. На ней можно выделить три сервиса:

- Storage — сервис, который хранит полученную телеметрию. Как правило, роль storage играет база данных, однако у некоторых инструментов (например, Grafana Tempo), в качестве хранилища используется объектное хранилище с собственным подходом к индексации данных. Из-за большого количества телеметрии сервис испытывает большую нагрузку на запись, а сам объём хранимых данных быстро увеличивается.

- Ingester — сервис, который принимает потоки телеметрии от множества узлов, и сохраняет эти данные в Storage. Ingester может делать различные операции с потоком данных: менять или удалять какие-то поля, отбрасывать ненужное. При этом сам Ingester не имеет состояния и не хранит никаких данных. Как и Storage, сервис испытывает нагрузку из-за большого потока входных данных.

- Querier — сервис, который принимает запросы на получение данных от клиента, обрабатывает его, получая нужные данные телеметрии из Storage, и отправляет ответ. Как правило, RPC на этом сервисе довольно низкий, потому что он используется небольшим

количеством людей (в основном, командами поддержки или разработчиками).

Сервисы Storage и Querier в совокупности играют роль бэкенда телеметрии. А в качестве Ingester-а, как правило, выступает ранее упомянутый OpenTelemetry Collector.

Рис. 4: Обобщённая диаграмма компонентов

3 Требования к системе

3.1 Случаи использования

Перед началом работы над системой была организована коммуникация с менеджерами продукта и лидерами команд, в результате чего удалось выявить и сформулировать случаи использования системы. Диаграмма случаев использования изображена на рисунке 5, а сами случаи можно разбить на две группы:

Актор — человек (инженер)

1. Изучить данные телеметрии. Пользователь обращается к системе, для того чтобы запросить и изучить различные данные телеметрии. В частности, взаимодействие с системой происходит через графический интерфейс WEB-приложения.

   (a) Изучить телеметрию только с интересующими параметрами. Данные телеметрии имеют множество параметров, таких как хост/узел, на котором она была сгенерирована, и другая метаинформация из предметной области опорной сети. Пользователь может использовать механизм фильтрации, который позволит запрашивать только нужную телеметрию.

   (b) Коррелировать трейсы и логи из одного контекста. Пользователя может интересовать телеметрия, связанная одним контекстом. Например, при анализе трейса полезно посмотреть логи, которые были сгенерированы системой в процессе выполнения этого трейса. Пользователь может использовать механизм, который позволит переходить от одного вида телеметрии к другому не теряя контекста.

2. Проанализировать ошибки системы. Этот случай использования является одним из вариантом случая "изучить данные телеметрии". Тем не менее, он был выделен в отдельную сущность, потому что представляет ключевой случай использования системы.

Актор — система опорной сети

1. Хранить телеметрию. Опорная сеть отправляет телеметрию в систему, для того чтобы её сохранить. Также, в частности, необходимо удалять старые данные по истечении некоторого времени, чтобы они не занимали место на диске.

   Также важно отметить, что в рамках данной работы требуется хранить только логи и трейсы. Метрики хранятся и обрабатываются в другой системе.

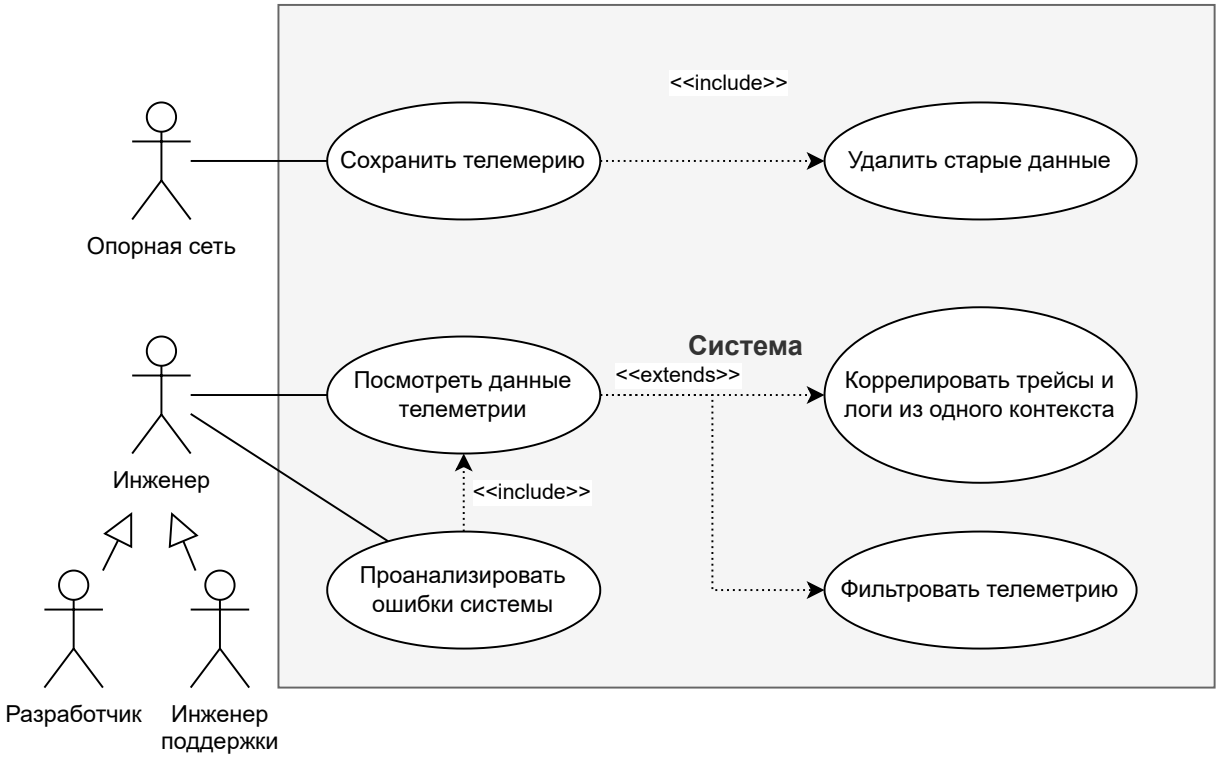

Рис. 5: Диаграмма случаев использования

Благодаря анализу случаев использования были сформулированы функциональные и нефункциональные требования к системе.

3.2 Функциональные требования

Для удобства функциональные требования разбиты на три группы. В первой группе находятся требования, касающиеся сбора телеметрии, во второй – касающиеся хранения телеметрии, и в третей — касающиеся её обработки.

1. Сбор телеметрии

   (a) Система по сети принимает логи и трейсы от различных сервисов и загружает её в хранилище.

2. Хранение телеметрии

   (a) Система хранит данные в базе данных (БД).

   (b) Система удаляет устаревшие данные для экономии места в БД.

3. Обработка и выдача телеметрии

   (a) Система реализует REST API интерфейс, с помощью которого клиенты могут запрашивать хранящиеся данные.

   (b) Система предоставляет пользовательский интерфейс для отображения телеметрии.

   (c) Система умеет выдавать разные виды телеметрии, относящиеся к одному контексту (например, выводить все логи, относящиеся к конкретному трейсу).

   (d) Система умеет находить всю телеметрию с заданным атрибутом (например, все данные по конкретному мобильному устройству).

   (e) Система умеет отвечать на запросы, связанные с агрегацией данных по времени (например, среднее время выполнения процедур).

3.3 Изменения в требованиях

Уже после начала разработки системы появилось дополнительное требование, касающееся хранения и запроса телеметрии. Если изначально подразумевалось, что телеметрия будет храниться в одном узле базы данных, то позднее возникла необходимость собирать и хранить телеметрию в нескольких узлах. Таким образом, дополнительные требования можно сформулировать так.

- Система собирает и хранит данные в нескольких узлах (шардах) БД. Подразумевается, что в шардах хранятся непересекающиеся множества данных.

- При запросе от пользователя система собирает данные из нескольких узлов. При этом сбор и объединение данных происходит прозрачно для пользователя.

4 Обзор: выбор базы данных

Поскольку ключевая функциональность системы — хранение и обработка большого объёма данных, формирование стека технологий было решено начать с выбора базы данных. Для реализуемой системы в сответствии с требованиями были выделены следующие ключевые критериии выбора.

1. **Открытый исходный код**. По требованию стейкхолдеров, необходима БД с открытым исходным кодом и разрешительной лицензией.

2. **Высокая скорость записи**. Поскольку телеметрия поступает большим непрерывным потоком, база данных должна обладать высокой производительностью записи данных.

3. **Наличие вторичных индексов**. Поскольку от системы требуется выполнять поиск данных с заданными фильтрами, необходимо иметь возможность быстро находить в БД записи с нужными значениями атрибутов, учитывая то, что объём хранящихся данный весьма велик (порядка сотен гигабайт). Кроме того, наличие индексов не должно существенно замедлять запись новых данных.

4. **Поддержка сжатия данных**. Некоторые атрибуты, по которым планируется выполнять поиск, присутствуют не во всех записях, поэтому в таких столбцах неизбежно появляться пустые значения. Кроме того, возможно наличие атрибутов с низким количеством уникальных значений. При этом таких столбцов потенциально может быть несколько десятков. В таких случаях полезно иметь механизм сжатия данных, причём сжатие данных по столбцам предпочтительнее.

5. **Поддержка запросов, ориентированных на время**. Один из ключевых параметров телеметрии — временная метка, представляющая момент, в который произошло событие. База данных должна обладать функциональностью БД временных рядов, то

есть поддерживать быструю выборку данных и агрегирование по времени.

Важно отметить, что на момент выбора базы данных ещё не возникло требований к распределённости хранилища, которые были описаны в секции 3.3

В первую очередь для рассмотрения было решено выбрать СУБД Cassandra и Scylla — эти СУБД нередко используются при построении систем хранения телеметрии. Также интерес вызвала Clickhouse — набирающая популярность OLAP СУБД, которая тоже испольуется при создании хранилищ телеметрии. Кроме того, поскольку работа с временными данными – один из ключевых критериев, было решено рассмотреть time-series базы данных. В качестве кандидата была выбрана InfluxDB, как одна из самых популярных представителей этого семейства.

Cassandra/Scylla. Cassandra [2] — распределённая NoSQL база с открытым исходным кодом. Cassandra использует Wide-column формат для организации данных, то есть каждая запись может иметь разное число столбцов. Это может быть удобно для хранения даннных, среди которых много пустых значений. Cassandra поддерживает вторичные индексы, которые, однако, могут быть малоэффективны на столбцах с большим количеством уникальных значений. Также поддерживается возможность работы с временными рядами, однако могут возникнуть проблемы с производительностью при выполнении агрегирующих запросов. Поддерживается и сжатие данных, однако данные хранятся и сжимаются по строкам. Отдельно стоит отметить, что Cassandra в большей степени ориентирована на распределённость и высокую доступность, что затрудняет её настройку, развертывание и эксплуатацию. Кроме того, из-за особенностей организации данных Cassandra требует особый подход к созданию схемы данных, что может негативно сказаться на времени разработки.

Альтернативой Cassandra является Scylla [20] — её идейный наследник. Scylla наследует все преимущества Cassandra и сопутствующую

им сложность. Данные так же организованы по принципу wide column store, поддерживается такой же язык запросов. Основное преимущество Scylla – производительность. Она написана на современном C++ на базе асинхронного фреймворка Seastar [21], за счёт чего в несколько раз превосходит Cassandra по скорости.

InfluxDB. InfluxDB [7] — на момент написания работы один из самых популярных [1] представителей семейства баз данных временных рядов (time series databases). Такие базы данных ориентированы на работу с записями, каждая из которых имеет временную метку. Поддерживаются быстрая выборка по времени, агрегирующие запросы. Также подобные БД хорошо оптимизированы для быстрой записи больших объёмов данных. Данные в InfluxDB хранятся в колоночном виде, за счёт чего реализуется хорошее сжатие. Однако у InfluxDB, как и у многих других БД этого семейства, есть особенность: вторичные индексы оптимизированы для работы со столбцами, имеющими ограниченное количество уникальных значений. Если такой индекс построить на колонке, которая может иметь неограниченное количество уникальных значений (например, уникальный идентификатор), то при большом объёме данных индекс будет работать медленно. Это, в частности, приведёт к уменьшению скорости записи. Эту проблему разработчики решили в движке InfluxDB v3.0, однако на момент написания работы он доступен только по enterprise лицензии, исходный код закрыт. В случае InfluxDb более ранних версий (InfluxDB v1 и v2) часть функциональности доступна по разрешительной лицензии, а полная функциональность — только в enterprize-версии.

Clickhouse. Clickhouse [3] — колоночная БД с открытым исходным кодом. База данных ориентирована на OLAP сценарии работы, что подразумевает выполнение аналитических запросов на больших объёмах данных, быструю запись данных и редкое удаление. Как и в InfluxDB, данные хранятся в колоночном виде, что позволяет экономить место с

[1] По данным сайта db-engines.com: https://db-engines.com/en/ranking/time+series+dbms

помощью сжатия. Важным преимуществом является то, что Clickhouse обладает вторичными индексами, которые называются Data Skipping индексы. При выполнении запроса они позволяют не тратить ресурсы на чтение блоков данных, в которых заведомо нет нужных значений индексируемого столбца. Также в Clickhouse реализована богатая функциональность для работы с данными временных рядов, в том числе различные агрегирующие функции. Стоит отметить, что Clickhouse использует SQL-подобный язык запросов, который по семантике близок к языкам традиционных реляционных БД. Кроме того, Clickhouse достаточно прост в развёртывании и эксплуатации.

Итоги В таблице 1 приведено сравнение характеристик рассмотренных баз данных.

	Открытый код	Быстрая запись	Вторичные индексы	Сжатие	Временные ряды
Cassandra	Да	Да	Да	По строкам	С ограничениями
InfluxDB	Частично	Да	С ограничениями	По столбцам	Да
InfluxDB v3	Нет	Да	Да	По столбцам	Да
Clickhouse	Да	Да	Да	По столбцам	Да

Таблица 1: Характеристики рассмотренных БД.

5 Архитектура

5.1 Компоненты и технологии

Итоговая диаграмма компонентов системы представлена на рисунке 6. На ней показаны абстрактные компоненты (выделены серым), проиллюстрированные ранее на рисунке 4, и конкретные компоненты, которые реализуют поведение этих абстрактных компонентов в терминах предоставляемых/потребляемых интерфейсов.

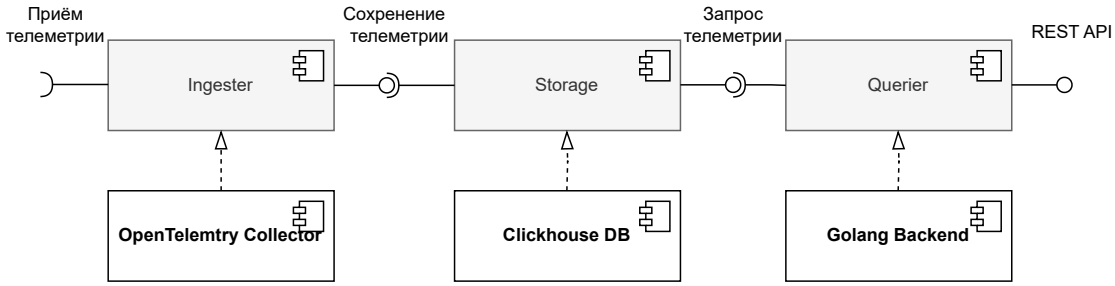

Рис. 6: Диаграмма компонентов реализуемой системы

Рассмотрим каждый из компонентов подробнее.

Ingester. В качестве Ingester-а было принято решение использовать OpenTelemetry Collector и отказаться от идеи реализовать этот компонент самостоятельно. Основная причина такого решения — ограниченное время, которе отведено на разработку. Для текущих требований к производительности достаточно существующего коллектора. Кроме того, поскольку коллектор не хранит данные, при необходимости его можно горизонтально масштабировать. Ещё одним аргументом в пользу коллектора является то, что он более гибкий, поскольку позволяет быстро изменять конфигурацию потоков данных.

Storage. С учётом результатов сравнения различных баз данных в секции 4, в качестве хранилища данных было принято решение использовать базу данных Clickhouse: она полностью удовлетворяет выдвинутым критериям выбора. Кроме того, процесс её развёртывания и экс-

плуатации значительно проще, чем у Scylla и Cassandra, что является несомненным плюсом при разработке прототипа системы.

Querier. Поскольку на Querier не предполагается высокой нагрузки, было принято решение реализовать его на языке Go. Рассматривался также и C++, поскольку он лучше знаком команде разработчиков. Однако от него было решено отказаться, поскольку его использование привело бы к излишним сложностям и замедлению скорости разработки.

На рисунке 7 компоненты системы изображены на диаграмме потоков данных. Опорная сеть генерирует телеметрию и самостоятельно направляет её в OpenTelemetry Collector. Логи отправляются по протоколу syslog, а трейсы по протоколу OTLP. OpenTelemetry Collector принимает телеметрию, парсит протоколы и сохраняет в Clickhouse. По запросу от клиента Backend получает данные телеметрии из Clickhouse и возвращает их в формате json.

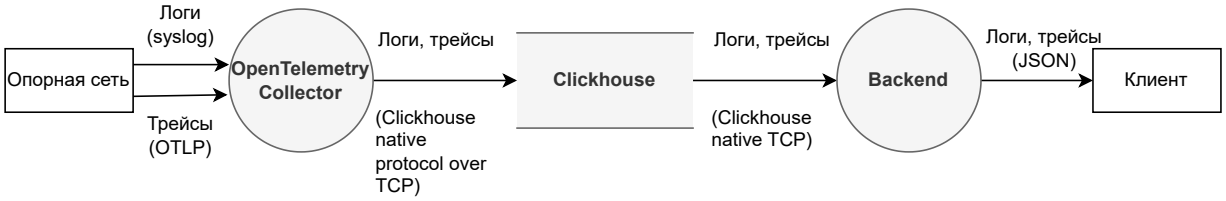

Рис. 7: Диаграмма потоков данных

5.2 Пользовательский интерфейс

Одно из требований к системе — наличие web-интерфейса, который позволит визуализировать телеметрию, облегчит её поиск и анализ. Кроме того, графический интерфейс должен соответствовать корпоративному стилю компании. Поэтому В качестве основной технологии для разработки интерфейса был выбран Angular [1] — фреймворк для создания одностраничных веб-приложений. На выбор повлияли два основных фактора:

- На базе Angular компания разрабатывает собственные корпоративные библиотеки элементов графического интерфейса. Эти библиотеки являются основой при разработке интерфейсов для других продуктов компании, таким образом формируя единый стиль интерфейса. Таким образом, Angular позволит внедрить библиотеки в проект и реализовать интерфейс в корпоративном стиле.

- Из предыдущего пункта вытекает, что в компании работают специалисты, хорошо знакомые с фреймворком. Таким образом, при использовании Angular разработчикам не потребуется дополнительное время на изучение технологии.

Разработка интерфейса велась под руководством автора данной работы. Однако сам автор не погружался в глубокие технические аспекты фреймворка, а принимал ключевые "стратегические" решения, касающиеся UI/UX — какие требуются экраны, что на этих экранах должно быть изображено, какие элементы интерфейса необходимы для того, чтобы пользователям было удобно работать с инструментом. Эти решения переводились в требования и передавались команде фронтенд-разработчиков, которая и реализовывала интерфейс.

Рассмотрим экраны приложения и то, какой функциональностью они обладают.

1. **Экран для просмотра логов.** На этом экране пользователь может в интерактивном режиме просматривать логи, которые хранятся в БД. Поддерживается фильтрация — например, можно отображать логи только от конкретного узла исследуемой системы. При необходимости пользователь может перейти от лога к соответствующему трейсу и изучить его.

2. **Экран для просмотра трейсов.** На этом экране отображается список трейсов. Для каждого трейса отображается полезная информация — например, количество спанов, количество ошибок, длительность. Как и на экране с логами, поддерживается фильтрация.

3. **Экран с древовидным представлением трейса.** Здесь выбранный трейс отрисовывается в виде дерева спанов. Дерево позволяет визуально изучить трейс, наглядно увидеть длительность спанов, их уровень вложенности, наличие ошибок. Для каждого спана можно посмотреть соответствующую ему метаинформацию. Также можно просмотреть все логи, которые соответствуют текущему трейсу. Скриншот экрана представлен на рисунке 8. Вдоль оси абсцисс показано время, вдоль оси ординат — вложенность спанов (отношение родитель-ребёнок, родитель уровнем выше).

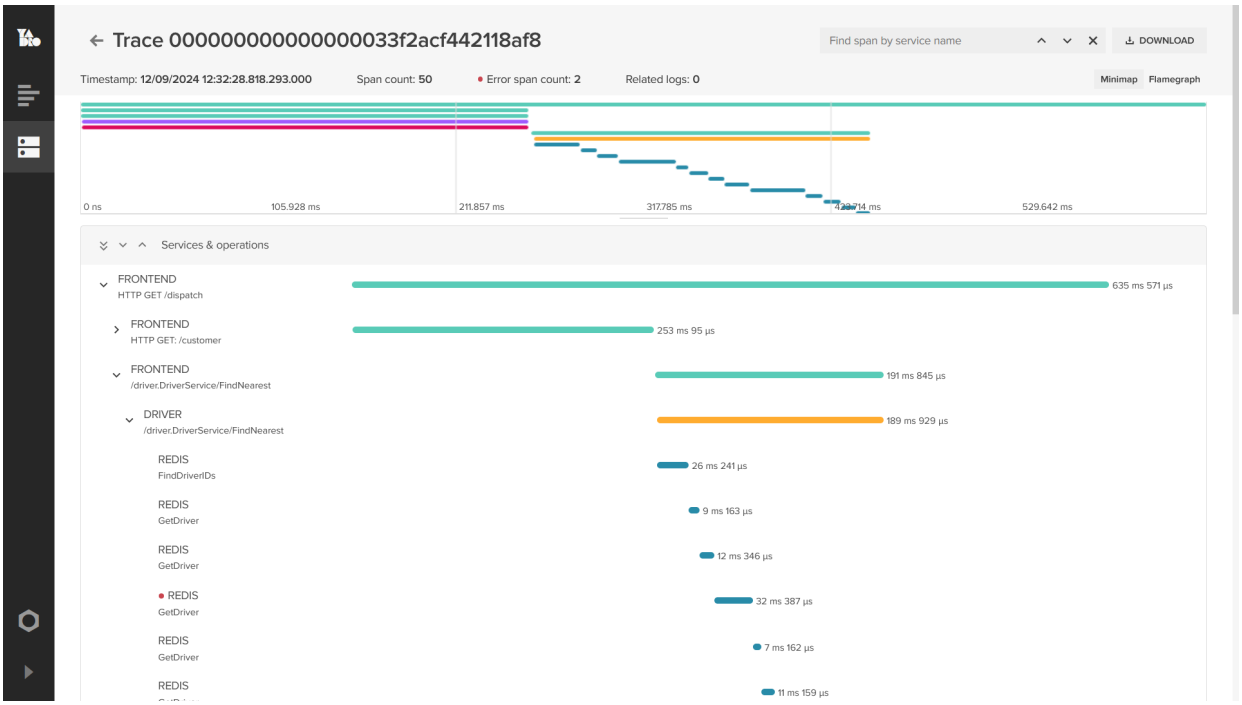

Рис. 8: Скриншот экрана с древовидным представлением трейса

5.3 Пример случая использования

Рассмотрим один из случаев использования системы с точки зрения пользователя. Предположим, инженеру поддержки необходимо проанализировать ошибки в работе опорной сети, возникшие при взаимодействии с конкретным абонентом за последние сутки. Для этого он выполнит следующие действия.

1. Пользователь откроет экран с таблицей трейсов (экран 2). На этом экране задаст фильтр по временному промежутку, фильтр по идентификатору пользователя, а также активирует опцию отображения только трейсов с ошибками.

2. После применения фильтров пользователь увидит в таблице найденные трейсы. При необходимости он сможет выбрать необходимый трейс и изучить его детально в древовидном представлении (экран 3).

3. Если необходимо, пользователь может изучить логи для просматриваемого трейса. В этом случае система переключится от экрана 3 к экрану отображением логов (экран 1) и покажет только необходимые записи.

6 Особенности реализации

Рассмотрим подробнее некоторые особенности компонентов приложения, которые были представлены в предыдущей главе.

6.1 Конфигурация OpenTelemetry Collector

Одна из ключевых особенностей OpenTelemetry Collector — возможность конфигурирования потоков обработки телеметрии с помощью пайплайнов, которые описываются в виде yaml-файла. Это позволяет манипулировать потоками без необходимости написания кода и пересборки самого приложения.

Пайплайн представляет собой цепочку из компонентов-обработчиков трёх типов: **receiver**, **processor** и **exporter**. Компоненты типа receiver принимают данные телеметрии от различных источников по различным протоколам. Компоненты типа processor могут совершать различные манипуляции над данными — например, изменять или отбрасывать. Компоненты типа exporter конвертируют данные в нужный протокол и отправляют в некоторую точку назначения (например, на другой сервер). Стоит отметить, что все компоненты изначально включены в сборку приложения, благодаря чему при изменении конфигурации пересборка не требуется.

Используемая в данном проекте конфигурация изображена на диаграмме 9. На ней изображены цепочки обработки логов и трейсов. Каждая такая цепочка представлена отдельным пайплайном. Рассмотрим пайплайны и компоненты подробнее.

Оба пайплайна начинаются с одного компонента типа receiver. Пайплайн логов принимает данные по протоколу syslog [19] с помощью компонента **Syslog Receiver**, а пайплайн трейсов – по протоколу OTLP [12] с помощью **OTLP Receiver**. Полученные данные приводятся к единой модели логов и трейсов соответственно, и передаются компонентам типа processor.

Набор компонент типа processor у обоих пайплайнов совпадает и состоит из двух обработчиков – **Memory Limiter** и **Batch Procesor**.

Memory Limiter позволяет приостановить приём телеметрии и отбрасывать новые данные в случае, если потребление памяти приложением достигло определённого предела. В противном случае приложение может истратить всю оперативную память системы. Следующий за ним Batch Processor накапливает данные во внутреннем буфере и при его заполнении отправляет содержимое буфера следующему обработчику.

Завершают пайплайн компоненты типа exporter. В данном случае в обоих пайплайнах используется компонент **Clickhouse Exporter**. Компонент непосредственно взаимодействует с экземпляром базы данных Clickhouse и позволяет записывать данные телеметрии сразу в таблицы с определённой схемой. Стоит отметить, что Batch Processor предшествует этому обработчику не случайно — эффективность вставки в Clickhouse возрастает, если записывать данные реже, но большими блоками.

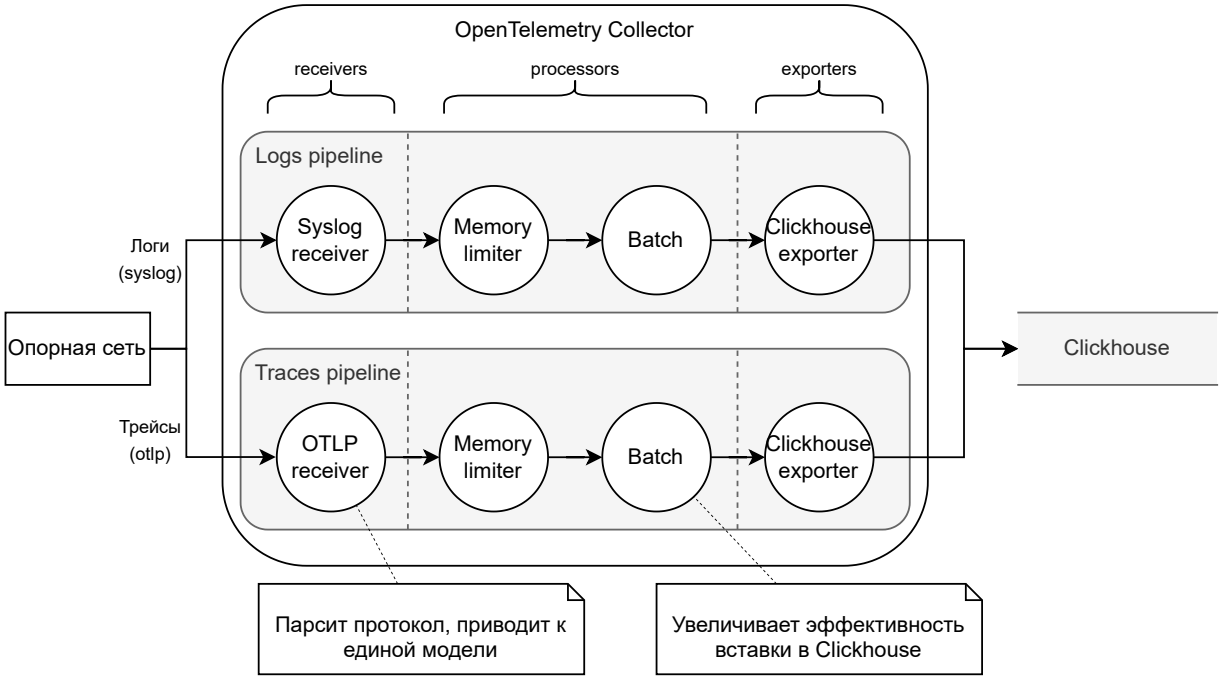

Рис. 9: Конфигурация потоков обработки телеметрии

6.2 Схема Clickhouse

На этапе прототипирования использовалась схема, которая по умолчанию предлагается разработчиками OpenTelemetry Collector. Одна-

ко позднее появилась необходимость определить собственную схему. Проблема заключалась в том, что схема по умолчанию зафиксирована в коде компонента Clickhouse Exporter, который является частью OpenTelemetry Collector. Один из способов решения — изменить код OpenTelemetry коллектора и пересобрать его. Однако позднее было решено отказаться от этого способа в пользу более гибкого варианта.

Итоговый способ подмены схемы использует механики Clickhouse: материализованные представления и null-таблицы. Рассмотрим эти механики подробнее.

Null таблицы — это с точки зрения пользователя обычные таблицы, которые не хранят данные. В них можно записывать данные как и в обычные таблицы с помощью запроса `INSERT`, но на диск данные записываться не будут.

Материализованные представления в Clickhouse несколько отличаются от материализованных представлений в других БД. По-сути, они представляют собой триггер, который срабатывает при вставке данных в таблицу и записывает их в другую таблицу. При этом возможно манипулировать вставляемыми данными — как в обычном SELECT-запросе при задании представления.

Таким образом, комбинация null-таблицы и и материализованного представления позволяет выстроить особую цепочку обработки данных. Диаграмма потоков данных изображена на рисунке 10. Сверху показан исходный вариант — происходит вставка в таблицу со схемой по умолчанию (Default Table). Ниже показан итоговый вариант вставки. Таблица по умолчанию переключается в режим null-таблицы. Для этой null-таблицы создаётся материализованное представление, которое обогащает данные, переводит их в новую схему и записывает в новую таблицу (Target Table). При вставке данных в Default Table срабатывает материализованное представление и записывает данные в TargetTable. После этого исходные данные в Default Table отбрасываются, и лишней записи на диск не происходит.

Полученная цепочка позволяет ослабить зависимость от зафиксированной схемы OpenTelemetry Collector и определить свою, обогащённую

схему. Так, например, в итоговой схеме несколько пар ключ-значение извлекаеются из стоблца типа `Map` и добавляются как отдельные столбцы —значение ключа становится именем столбца. Также добавляется столбец с уникальным идентификатором записи, который генерируется с помощью встроенной функции `generateUUIDv4()`.

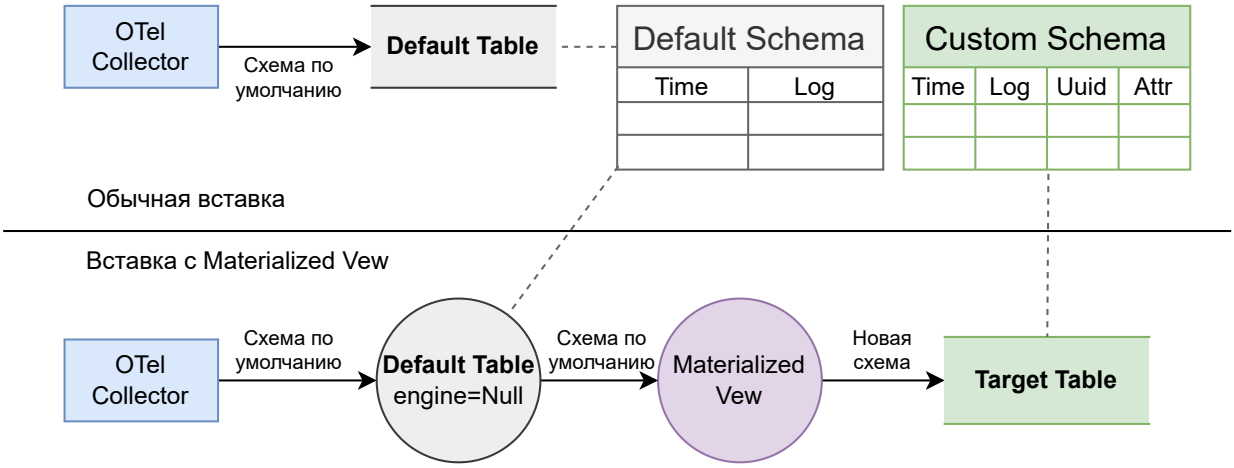

Рис. 10: Варианты вставки в Clickhouse

6.3 Backend

Бэкенд-приложение реализовано на языке Go и представляет собой HTTP-сервер, обрабатывающий REST запросы на получение данных телеметрии. Архитектура приложения представлена на диаграмме пакетов 11. Рассмотрим каждый из пакетов подробнее.

Handler. Пакет принимает REST-запросы, обрабатывает их и отправляет данные клиенту. Для этого он оперирует интерфейсами пакетов queryParser и storage. Параметры входящего REST-запроса handler парсит с помощью пакета queryParser. Полученные параметры передаются в пакет storage, который возвращает нужные данные. Затем hander сериализует данные в формат json и отправляет клиенту.

В качестве web-фреймворка, который обрабатывает HTTP-запросы, используется библиотека `gorilla/mux` [23]. Эта библиотека легковесная и полностью совместима с интерфейсом стандартной библиотеки `net/http`, при этом обладает большей функциональностью и гибкостью.

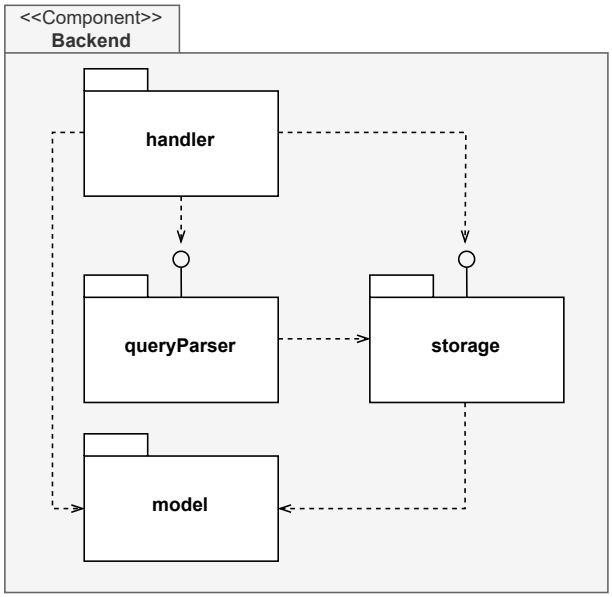

Рис. 11: Диаграмма пакетов backend-приложения

Библиотека позволяет задавать конечные точки REST, а также набор доступных HTTP-методов для каждой точки.

На текущий момент поддерживаются следующие REST ресурсы и коллекции:

- `GET /logs` — возвращает список логов, которые соотвествуют заданным фильтрам; поддерживает пагинацию.
- `GET /logs/dump` — возвращает полный список логов, которые соотвествуют заданным фильтрам; в отличие от предыдущей коллекции, возвращается полный набор логов в текстовом формате, запакованный в gzip-архив.
- `GET /traces` — возвращает список трейсов, которые соотвествуют заданным фильтрам. Также поддерживает пагинацию.
- `GET /traces/{traceId}` — возвращает трейс, соответствующий заданному идентификатору `traceId`
- `GET /services` — возвращает список имён сервисов, данные о которых хранятся в Storage. Список сервисов формируется на основе данных трейсов.

- `GET /services/{serviceName}/operations` — возвращает список операций для сервиса с именем `serviceName`. Список также формируется на основе данных трейсов.

Фильтры задаются в GET-параметрах строки запроса. Примеры фильтров — временной интервал или имя сервиса, который генерирует телеметрию. При помощи фильтров можно коррелировать трейсы с логами — например, запросить все логи, относящиеся к трейсу с заданным traceId. Также для коллекций `/logs` и `/traces` реализована серверная пагинация, которая контролируется параметрами `limit` и `offset`.

Стоит отметить, что ошибки в процессе обработки запросов обрабатываются по-разному. Так, при ошибке парсинга параметров запроса возвращается код состояния HTTP 400 Bad Request, при прочих ошибках — код 500 Internal Server Error.

QueryParser. Пакет реализует логику парсинга параметров запросов. Запросы переводятся во внутреннее представление в виде структуры, которое затем модуль `handler` передаёт в `storage`. Модуль поддерживает парсинг двух наборов фильтров — для логов (коллекции `/logs` и `/logs/dump`) и для трейсов (коллекция `/traces`). Оба набора фильтров содержат как обязательные, так и необязательные параметры. Так, например, фильтр логов обязательно должен содержать временные метки начала и конца некоторого периода. Поэтому QueryParser валидирует запросы с точки зрения семантики набора фильтров и корректности ввода. Стоит отметить, что при этом значения параметров не валидируются с точки зрения корректности синтаксиса SQL.

Storage. Этот пакет отвечает за взаимодействие с базой данных. Пакет использует структуры запросов, подготовленные пакетом queryParser, формирует SQL-запросы и получает данные из БД. Здесь значения параметров запроса валидируются уже с точки зрения синтаксиса SQL. Для взаимодействия с базой данных используется официальная библиотека-клиент `clickhouse-go` [4]. При получении ответа от БД пакет заполняет структуру модели и отдаёт пакету `handler`.

6.4 Распределённые запросы

В рамках дополнительных требований, описанных в пункте 3.3, уже после начала разработки возникла необходимость запрашивать данные из нескольких узлов Clickhouse.

Для этих целей было принято решение использовать табличную функцию `remote()`, которая реализована в Clickhouse. Функция принимает на вход список адресов узлов `hosts`, из которых необходимо получить данные. Например получении запроса вида `SELECT ... FROM remote(hosts)` Clickhouse отправляет запрос на все узлы из списка `hosts`, получает данные со всех узлов и возвращает объединённый результат. Таким образом, сбор и объединение данных производит тот узел БД, который обрабатывает исходный запрос. Назовём его *исходным узлом*. Для клиента БД, в роли которого выступает Backend, в этом случае получение данных происходит прозрачно — бэкенд делает запрос на исходный узел и получает ответ с уже подготовленными данными. Список хостов, из которых необходимо получить данные, конфигурируются при запуске бэкенда. Выполнение запроса проиллюстрировано на диаграмме последовательностей на рисунке 12).

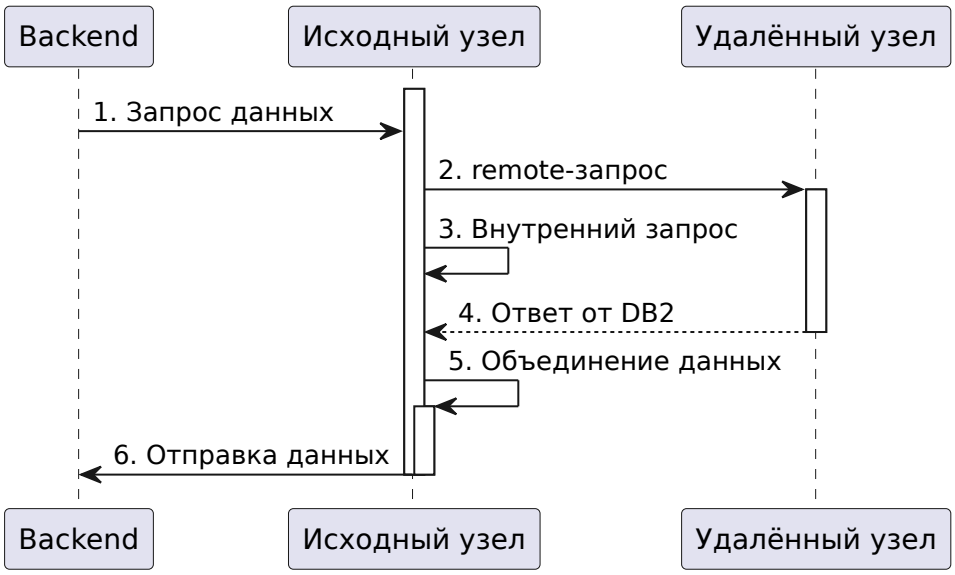

Рис. 12: Выполнение распределённого запроса

Рассмотрим некоторые особенности такого подхода. Во-первых, стоит отметить, что при таком подходе исходный узел БД самостоятельно

объединяет собранные с нескольких узлов данные (шаг 5 на рисунке 12). В частности, при объединении выполяется агрегация (если запрос содержит агрегирующие функции), сортировка, отбрасываются лишние данные (если заданы ограничения на размер выборки). Кроме того, как правило, в список адресов хостов добавляется и сам исходный узел – для того, чтобы в объединённый набор данных попали данные и с самого исходного узла. Запрос собственных данных на исходном узле изображён на третьем шаге рисунка 12. Список адресов узлов хранится на бэкенде и поддерживается актуальном состоянии, основываясь на статусе доступности. Каждому из узлов бэкенд периодически направляет healthcheck-запрос, и если узел не отвечает, то он помечается как неактивный и не участвует в remote-запровах. Впоследствии, если узел снова становится доступен, он вновь помечается как активный.

Рассматривались и другие способы реализации сбора данных из нескольких узлов.

- Объединение всех узлов в кластер Clickhouse. В конфигурационном файле БД создаётся кластер и задаётся список адресов узлов, из которых он состоит. Затем на исходном узле создаётся особая таблица типа DistributedTable. Эта таблица выступает интерфейсом для кластера – при запросе данных из этой таблицы исходный узел Clickhouse автоматически собирает и объединяет данные со всех узлов кластера, сохраняя прозрачность для клиента. Такой подход аналогичен подходу с функцией `remote()`, однако он менее гибкий – при обращении к таблице опрашивается *каждый* узел кластера, даже если нужны не все. Это влечёт существенные задержки в том случае, если узлы кластера находятся на существенном удалении друг от друга. При для изменнения списка узлов в кластере требуется перезагрузка исходного узла. В случае с функцией remote() такого ограничения нет, и при каждом запросе можно указывать произвольный список узлов. Поэтому для обеспечения гибкости было решено оказаться от создания кластера и использовать функцию `remote()`.

- Сбор данных на бэкенде. В этом случае бэкенд самостоятельно опрашивает необходимые узлы, получает данные и объединяет их. Однако при таком подходе возникает необходимость реализовывать процесс агрегации самостоятельно. Это потребовало бы дополнительных ресурсов на разработку и тестирование. Таким образом, было принято решение отказаться от этого варианта в пользу уже готовой remote(), которая реализует нужную функциональность.

7 Тестирование и внедрение

Перед поставкой заказчику были подготовлены автоматизированные тесты, а также проведено ручное тестирование.

7.1 Интеграционное и модульное тестирование

Для тестирования backend-части модульные тесты были подготовлены с помощью пакета `testing` из стандартной библиотеки Go. С помощью модульных тестов, нарпример, проверяется правильность парсинга запросов и валидации параметров. Для frontend-части модульные тесты были подготовлены frontend-разработчиками. Для этого был использован фреймворк Jasmine.

Также возникла необходимость протестировать взаимодействие backend-части и Clickhouse. В первую очередь, было важно убедиться в корректности написанных SQL-запросов и проверить, что возвращаемые backend-приложением данные семантически корректны. В частности, подобные тесты были необходимы для проверки корректности фильтрации и агрегации данных. Также было принято решение подготовить тесты для проверки работы распределённых запросов. Необходимо было убедиться, что данные с нескольких узлов корректно собираются и правильно объединяются.

Таким образом, было принято решение реализовать интеграционные тесты, проверяющие взаимодействие Clickhouse и Backend части. Тестовые сценарии реализованы автором работы и построены следующим образом.

1. На тестовом стенде запускаются два узла Clickhouse и backend
2. В каждый из двух узлов Clickhouse загружаются тестовые данные для теущего сценария тестирования.
3. Отправляется запрос на одну или несколько конечных точек REST backend-части в соответствие со сценарием.

4. Backend-часть обрабатывает запрос, получает данные из узлов Clickhouse и отправлет ответ.

5. Полученный ответ проверяется или сравнивается с ожидаемым.

6. После завершения сценария данные в Clickhouse очищаются для проведения следующего сценария.

Ключевые шаги подобных тестов — отправка REST-запроса и проверка полученного ответа. Для автоматизации этих действий было принято решение использовать фреймворк Karate [9]. Это фреймворк для автоматизации тестирования API с открытым исходным кодом, написанный на Java. Фреймворк предлагает простой предметно-ориентированный язык для написания тестов, который позволяет отправлять запросы и проверять ответы. Фреймворк был выбран по двум основным причинам.

- Простота написания тестов. Тесты пишутся в декларативном человеко-читаемом формате, благодаря чему код удобно пишется и хорошо читается. Также в языке поддерживается множество функций для работы с json, что позволяет проверять различные условия на полученных ответах.

- На момент написания работы Karate активно использовался командой тестирования. Во-первых, это обстоятельство позволило быстро интегрировать фреймворк в тестовую среду, поскольку были доступны все библиотеки и зависимости. Во-вторых, инженеры по тестированию уже обладали опытом работы с фреймворком, что позволило разработчикам быстро освоить инструмент и начать работу. Кроме того, обсуждалась возможность передачи исходного кода тестов команде тестирования для последующей поддержки.

Таким образом, с помощью фреймворка Karate реализованы тесты конечных точек rest. Подготовленные автоматические тесты выполняются в CI/CD пайплайнах при каждом слиянии кода в главной ветке.

7.2 Ручное тестирование и апробация

Для проведения ручного функционального тестирования система была развёрнута на тестовом стенде у инженеров по тестированию, в чём непосредственно принимал участие автор работы. Затем инженеры начали использовать систему по её прямому назначению при тестировании сервисов опорной сети. При таком ad-hoc подходе во время тестирования были задействованы все узлы системы: при выполнении тестовых сценариев опорная сеть генерировала телеметрию, которую принимал OpenTelemetry Collector и сохранял в Clickhouse. также инженеры по тестированию использовали пользовательский интрефейс системы, чтобы проанализировать трейсы и логи, которые сгенерировала опорная сеть при выполнении сценария. Через некоторое время после начала эксплуатации системы у инженеров по тестированию была запрошена обратная связь. Обратная связь была преимущественно положительной, за исключением небольших пожеланий, которые вскоре были исправлены. Инженеры-пользователи отметили удобство пользовательского интерфейса, возможность фильтрации и поиска нужных данных. Позднее был создан канал коммуникации для оперативного обмена обратной связью и запросов на добавление новой функциональности.

Кроме функционального тестирования, было проведено тестирование производительности системы. Во-первых, проводилось нагрузочное тестирование — чтобы убедиться, что система справляется с обработкой и сохранением входящего потока телеметрии. Во-вторых, проводилось объёмное тестирование — чтобы убедиться, что система функционирует при большом объёме хранящихся в ней данных. Тесты подтвердили, что при ожидаемых нагрузках система работоспособна и соответствует нефункциональным требованиям к производительности.

7.3 Внедрение

По результатам тестирования система была признана готовой к продуктовой эксплуатации. Система была передана заказчику и в настоящий момент развёрнута на стенде заказчика.

Заключение

В ходе работы за четвертый семестр была реализована функциональность распределённого сбора данных. Также были подготовлены тесты и произведена апробация. В итоге система передана в эксплуатацию и развернута у заказчика.

Список литературы

[1] Angular framework. — URL: https://angular.dev/.

[2] Cassandra: Open Source NoSQL Database. — URL: https://cassandra.apache.org/.

[3] ClickHouse: Fast Open-Source OLAP DBMS. — URL: https://clickhouse.com/.

[4] Golang SQL database client for Clickhouse. — URL: https://github.com/ClickHouse/clickhouse-go.

[5] Grafana. — URL: https://grafana.com/docs/grafana/latest/introduction/.

[6] Grafana — What is observability? — URL: https://grafana.com/docs/grafana-cloud/introduction/what-is-observability/.

[7] InfluxDB Time Series Data Platform. — URL: https://www.influxdata.com/.

[8] Jaeger: open source, distributed tracing platform. — URL: https://www.jaegertracing.io/.

[9] Karate API Testing. — URL: https://www.karatelabs.io/.

[10] Microservices Adoption in 2020. — URL: https://www.oreilly.com/radar/microservices-adoption-in-2020/.

[11] Microservices architecture. — URL: https://www.atlassian.com/microservices/microservices-architecture.

[12] OTLP Specification. — URL: https://opentelemetry.io/docs/specs/otlp/.

[13] Observability tools. — URL: https://signoz.io/blog/observability-tools/.

[14] OpenCensus. — URL: https://opencensus.io/.

[15] OpenTelemetry. — URL: https://opentelemetry.io/docs/what-is-opentelemetry/.

[16] OpenTelemetry Collector. — URL: https://opentelemetry.io/docs/collector/.

[17] OpenTelemetry basic concepts. — URL: https://opentelemetry.io/docs/concepts/observability-primer/.

[18] The Opentracing project. — URL: https://opentracing.io/.

[19] RFC 5424 - The Syslog Protocol. — URL: https://datatracker.ietf.org/doc/html/rfc5424.

[20] ScyllaDB: Fast and Scalable NoSQL. — URL: https://www.scylladb.com/.

[21] Seastar: open-source c++ asynchronous framework. — URL: https://seastar.io/.

[22] SigNoz: Open-Source Observability. — URL: https://signoz.io/docs/.

[23] gorilla/mux http router. — URL: https://github.com/gorilla/mux.