Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 23.М04-мм

Реализация системы хранения телеметрии для опорной сети

СМИРНОВ Александр Андреевич

Отчёт по 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 [8, 9]. 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 [13] 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 [13].

На микросервисной архитектуре, 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 множество разных запросов, которые поступают от базовых станций и других компонентов сети. В терминах опорной сети такие запросы называются процедурами. При выполнении процедур может создаваться различная метаинформация, например, уникальный идентификатор мобильного устройства. Понятие трейса можно удобно отобразить на понятие процедуры, что позволяет применить уже разработанные стандарты для генерации системой телеметрии.

Предметная область телеметрии последние несколько лет активно развивается. На рынке представлены [11] различные решения, которые облегчают работу с такими данными. Однако эти решения рассчитаны

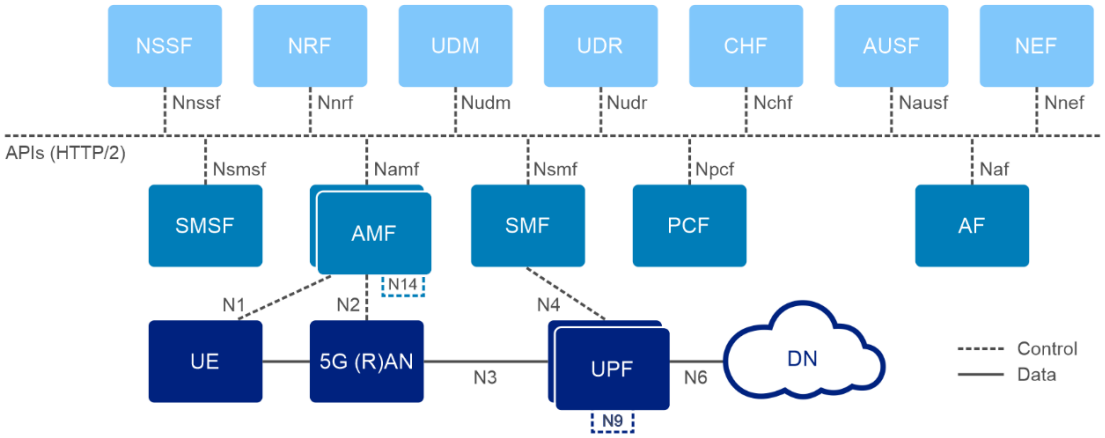

Рис. 1: Архитектура опорной сети
Источник: https://infohub.delltechnologies.com/p/the-5g-core-network-demystified

на широкий круг пользователей, из-за чего вынуждены абстрагироваться от какой-либо предметной области. Таким образом, было принято решение разработать собственный инструмент для сбора, хранения и обработки телеметрии, ориентированный на специфику предметной области телекоммуникационных данных.

1. Постановка задачи

Целью работы является разработка системы для сбора, хранения и обработки телеметрии опорной сети. 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. [13, 5]*

Определение 2. *Выходные данные системы, содержащие информацию о её внутреннем состоянии, называют [13] 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 [15] 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 [15] 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 [15] трейса. Спан представляет информацию об некоторой операции, которая была выполнена при обработке

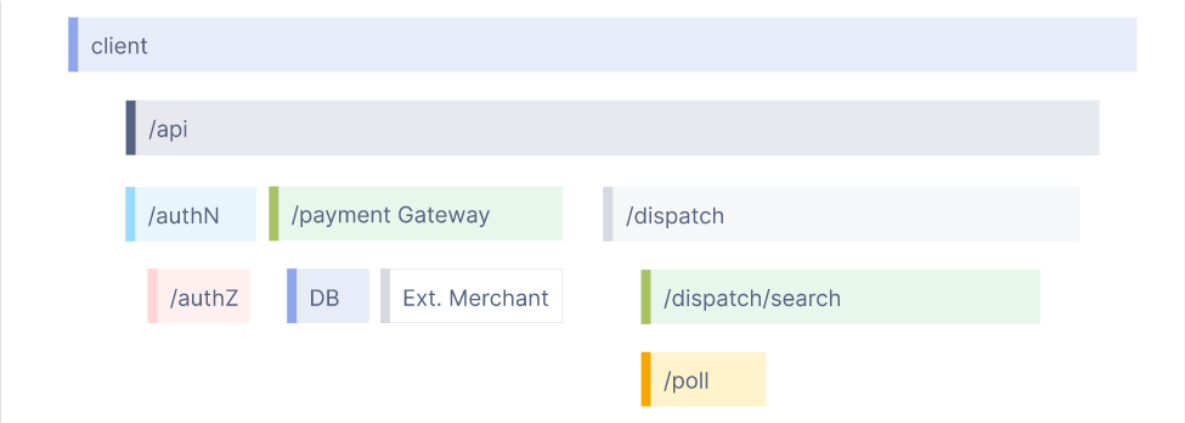

Рис. 2: Пример трейса
Источник: https://opentelemetry.io/docs/concepts/observability-primer

системой запроса (например, запрос системы к базе данных в рамках выполнения REST-запроса). Спан содержит время начала операции и её продолжительность, а также различные метаданные, характеризующие операцию. 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 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 [16] и OpenCensus [12], которые впоследствие объединились, в результате чего был создал фреймворк OpenTelemetry.

OpenTelemetry [13] — 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 [14]. 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 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 [7] — 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 [4]. Состоит из четырёх отдельных продуктов.

- Grafana — веб приложение, представляющее собой инструмент

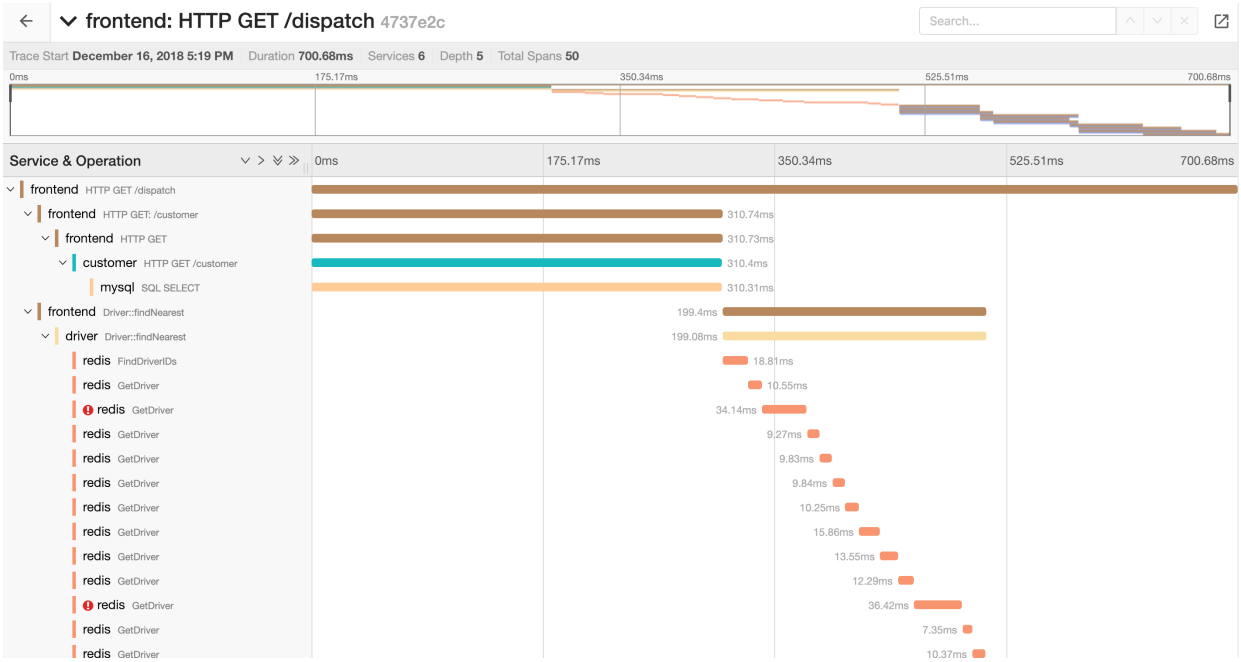

Рис. 3: Пользовательский интерфейс Jaeger-a
Источник: https://www.jaegertracing.io/docs/1.18/frontend-ui

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 [20] — 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 [2], 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 сервис довольно низкий, потому что он используется небольшим количеством людей (в основном, командами поддержки или

разработчиками).

Рис. 4: Обобщённая диаграмма компонентов

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

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 будет использовать механизм, который позволит переходить от одного вида телеметрии к другому не теряя контекста.

2. Проанализиолвать 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

   (a) Удалить приватную информацию.

   (b) Удалить неинтересные данные.

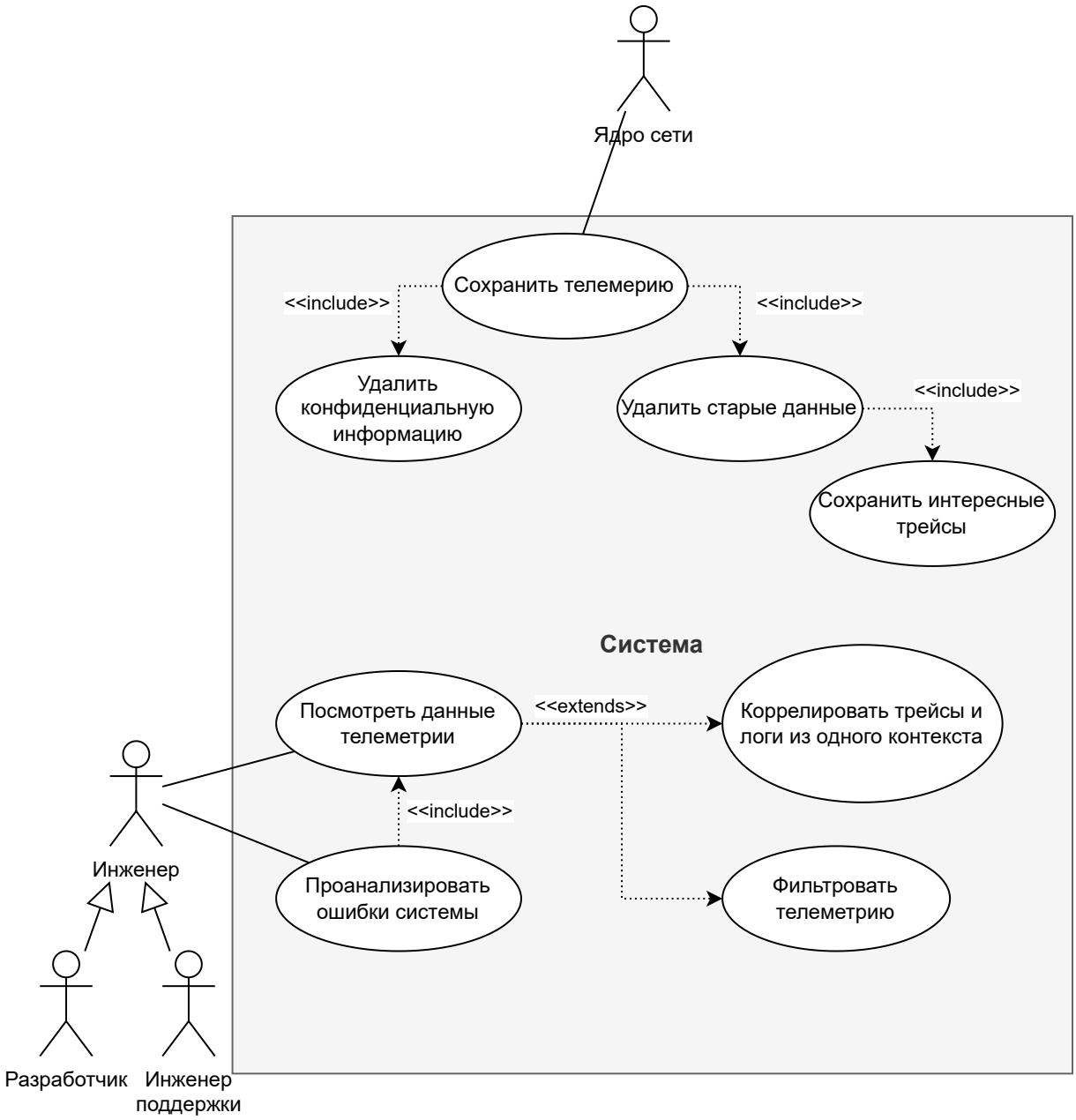

Рис. 5: Диаграмма случаев использования

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 телеметрию от различных сетевых функций и сохраняет её в хранилище.
    (b) При получении данных телеметрии система удаляет всю приватную информацию, содержащуюся в них (например, ключи шифрования).
2. Хранение телеметрии
    (a) Система хранит данные в базе данных (БД).
    (b) Система удаляет устаревшие данные для экономии места в БД.
    (c) Система выявляет “интересные” данные и продлевает срок их хранения в БД.
3. Обработка и выдача телеметрии
    (a) Система реализует REST API интерфейс, с помощью которого клиенты могут запрашивать хранящиеся данные.
    (b) Система умеет выдавать разные виды телеметрии, относящиеся к одному контексту (например, выводить все логи, относящиеся к конкретному трейсу).
    (c) Система умеет находить всю телеметрию с заданным атрибутом (например, все данные по конкретному мобильному устройству).

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

(e) Система поддерживает обработку хранящихся данных посредством пользовательских скриптов.

(f) Система предоставляет web-интерфейс для отображения телеметрии.

4. Обзор: выбор технологий

4.1. 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**. Поскольку не планируется закупать сторонние технологии, 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. **Быстрые вторичные индексы**. 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. **Сжатие данных**. Некоторые атрибуты, по которым планируется выполнять поиск, присутствуют не во всех записях, поэтому в таких столбцах неизбежно появляться пустые значения. 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.

На первый взгляд, под описанные критерии могут подойти некоторые популярные базы данных, такие как Cassandra, Scylla, Clickhouse. Также при сравнении было решено рассмотреть time-series базы данных, а в качестве представителя этого семейства БД была взята InfluxDB. Рассмотрим каждую БД подробнее.

Cassandra/Scylla. Cassandra [1] — 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, не такие функциональные, как в привычных реляционных БД. Также поддерживается возможность работы с временными рядами, хотя эта функциональность не заявлена разработчиками как ключевая. Поддерживается и сжатие данных, однако данные хранятся на диске 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 [18] — 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 [19], за счёт чего в несколько раз превосходит Cassandra по скорости.

InfluxDB. InfluxDB [6] — 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.

Clickhouse. Clickhouse [2] — 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 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. Также стоит отметить, что Clickhouse использует SQL-подобный язык запросов, который по семантике близок к языкам традиционных реляционных БД. Кроме того, Clickhouse

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

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	Возможно
Scylla	Да	Да	Да	По строкам	Возможно
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

Итоговая диаграмма компонент 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.

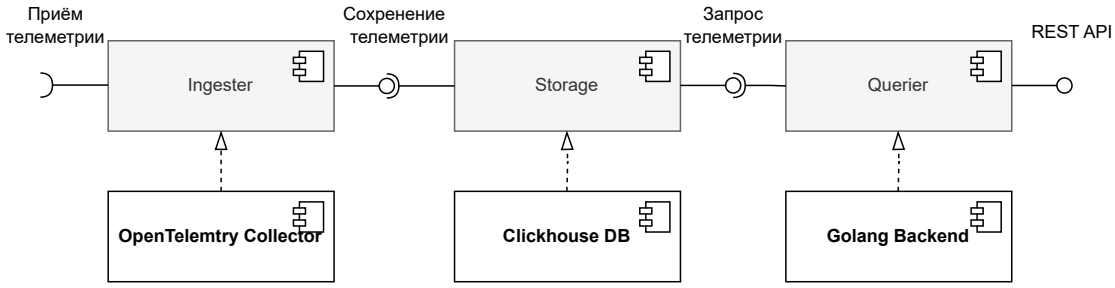

Рис. 6: Диаграмма компонентов реализуемой системы

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 конфирурацию потоков данных. Это также важно при разработке прототипа.

Storage. С учётом результатов сравнения различных баз данных в секции 4.1, в качестве хранилища данных было принято решение использовать базу данных Clickhouse: она полностью удовлетворяет выдвинутым критериям выбора. Кроме того, процесс её развёртывания и эксплуатации 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.

6. Особенности реализации

Рассмотрим подробнее некоторые особенности компонент 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 7. 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 [17] с помощью компонента **Syslog Receiver**, а пайплайн трейсов – по протоколу OTLP [10] 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 (например, если ком). 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.

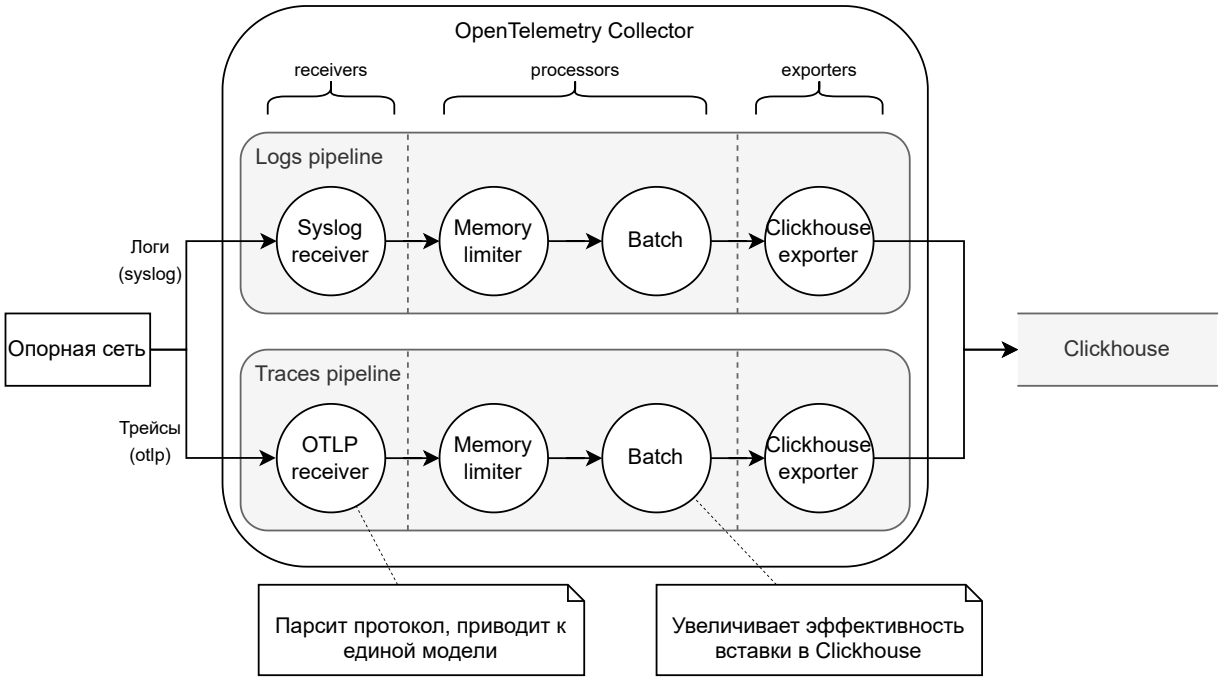

Рис. 7: Конфигурация потоков обработки телеметрии

6.2. Схема Clickhouse

На этапе прототипирования использовалась схема, которая по умолчанию предлагается разработчиками OpenTelemetry Collector. Однако 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 8. 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()`.

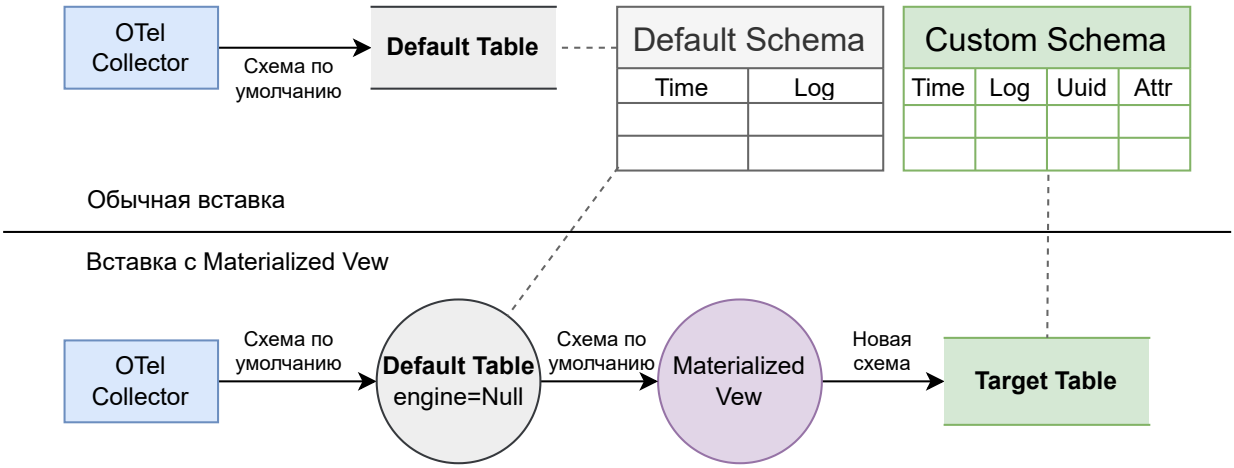

Рис. 8: Варианты вставки в Clickhouse

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 9. 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.

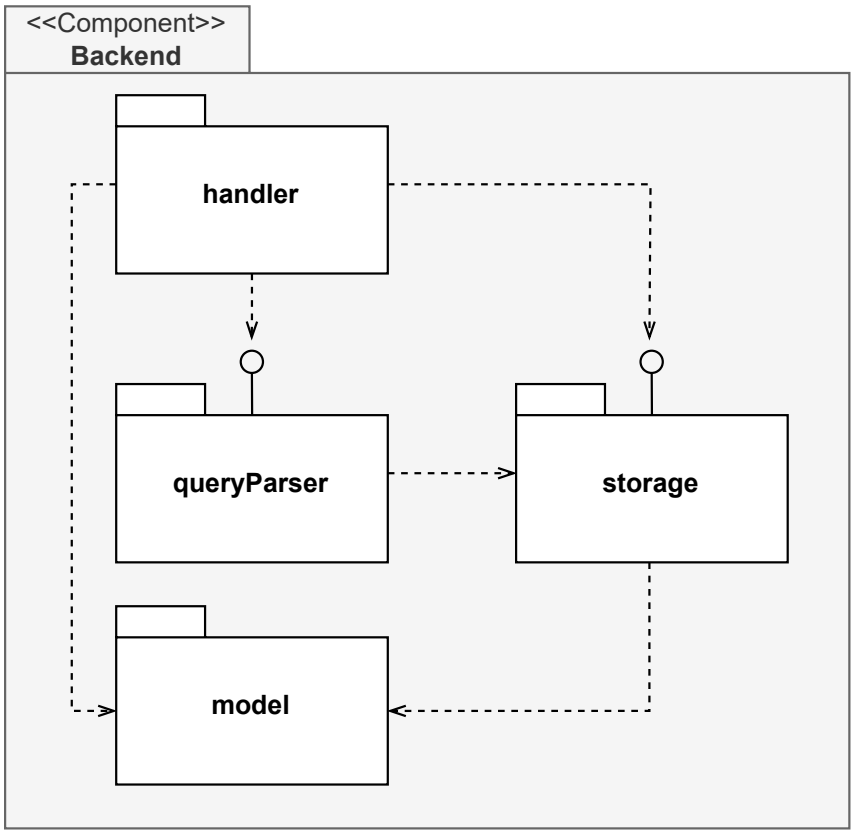

Рис. 9: Диаграмма пакетов backend-приложения

В качестве web-фреймворка, который обрабатывает HTTP-запросы, используется библиотека `gorilla/mux` [21]. Эта библиотека легковесная и полностью совместима с интерфейсом стандартной библиотеки `net/http`, при этом обладает большей функциональностью и гибкостью. 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` [3]. При получении ответа от БД пакет заполняет структуру модели и отдаёт пакету `handler`.

6.4. Развертывание

На данном этапе в целях упрощения разработки и отладки система разворачивается на трёх Docker-контейнерах из следующих образов:

- `opentelemetry-collector-contrib` — образ коллектора из официального репозитория;
- `clickhouse-server` — образ сервера Clickhouse из официального репозитория;
- `backend` — образ backend-части приложения

Контейнеры запускаются на одной машине. Для этого используется инструмент `docker compose` и подготовленный для него файл `docker-compose.yaml`. Также средствами Docker создаётся отдельная изолированная сеть, внутри которой происходит взаимодействие между контейнерами.

Также нужно отметить, что исходный код проекта закрыт.

Заключение

В ходе работы за третий семестр была реализована система хранения телеметрии. В рамках этой задачи было реализовано backend-приложение, подготовлена конфигурация для OpenTelemetry Collector и схема для СУБД Clickhouse.

В следующем семестре планируется произвести апробацию совместно с опорной сетью.

Список литературы

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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