

Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 22.Б15-мм

Утилита для парсинга SCSI логов

Кузнецов Арсений Николаевич

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
к. ф.-м. н., доц. Куликов Е. К.

Консультант:
Старший инженер по разработке ПО, ООО «Ядро Центр Программных Разработок»,
Казаков И. А.

Санкт-Петербург
2024

Оглавление

Введение	3
1. Постановка задачи	5
2. Обзор	6
2.1. Обзор существующих решений	6
2.2. SCSI	7
2.3. Используемые технологии	10
3. Парсинг логов	12
3.1. Абстракции парсеров	13
3.2. Реализации парсеров	15
3.3. Utils	16
3.4. Тестирование	19
3.5. Производительность	19
4. Работа с архивами	21
4.1. Реализация	21
4.2. Тестирование	23
5. Command line interface	24
5.1. Реализация	24
5.2. Тестирование	25
6. Заключение	28
6.1. Результаты	28
6.2. Планы развития	28
Список литературы	29

Введение

В современном мире компании для хранения большого количества информации используют системы хранения данных (СХД). YADRO¹ является одной из компаний, выпускающих решения в этой области, например, TATLIN.UNIFIED и TATLIN.OBJECT. Как и любой программный продукт, эти системы подвержены багам. Если баг обнаружится в финальной версии продукта, то это может привести к многомиллионным убыткам и потере репутации.

В компании YADRO для регрессионного тестирования СХД применяют внутреннюю закрытую разработку под названием MEYER SAN (далее MRSN). MRSN — это программно-аппаратный комплекс, который позволяет вносить неисправности (fault injection) при взаимодействии СХД со SCSI² диском.

Для дальнейшего развития MRSN хотят использовать информацию об ошибках, возникших с дисками у заказчиков. Зачастую единственная информация, которая имеется о таких ошибках — это LINUX логи. Из этих логов наиболее полезными для команды MRSN являются SCSI логи, так как MRSN работает на уровне SCSI.

Анализировать вручную пришедшие логи, размер которых может измеряться гигабайтами, достаточно долго и неприятно. Неудобность для анализа кроется еще в формате логов, которые зачастую приходят в виде архивов со сложной структурой, и в том, что формат самих логов может отличаться в зависимости от версии операционной системы и опций логирующих систем.

Можно представить логи в виде JSON³, с разобранными SCSI командами. Это повысит эффективность работы с ними благодаря машиночитаемости JSON, которая облегчает дальнейшую обработку логов средствами автоматизации. Кроме того, данный формат удобнее и для ручного анализа, так как для работы с JSON существует множество

¹<https://yadro.com/>

²<https://www.snia.org/education/online-dictionary/term/small-computer-system-interface>

³<https://www.json.org/json-en.html>

инструментов, например, утилита `jq`⁴.

Для удобства работы с логами можно создать утилиту, которая сможет принимать логи в различном виде, парсить SCSI логи и выдавать результат в виде JSON.

⁴<https://jqlang.github.io/jq/>

1. Постановка задачи

Целью работы является создание утилиты для анализа LINUX SCSI логов, которая может принимать логи как в текстовых файлах, так и в архивах, отбирая нужные файлы в нем, разбирать их и преобразовывать результат в JSON. Для выполнения цели были поставлены следующие задачи:

1. Спроектировать и реализовать модуль для парсинга основных LINUX SCSI логов (раздел 3).
2. Спроектировать и реализовать модуль для работы с архивами (раздел 4), который даст возможность получать текстовые файлы по архивам, в форматах `zip`, `tar`, и сжатым файлам в формате `gz`⁵. А также фильтровать файлы по предикату.
3. Спроектировать и реализовать CLI⁶ (раздел 5), который позволит на основе файла или архива с логами получать JSON результат с разобранными SCSI логами.

⁵Данные форматы файлов были встречены в имеемых архивах с логами.

⁶CLI — Command Line Interface (Интерфейс командной строки)

2. Обзор

2.1. Обзор существующих решений

Прямых аналогов утилиты нет. Существуют утилиты для анализа бинарных SCSI лог-страниц [5], но работа со SCSI лог-страницами является задачей другого плана, не рассматриваемой командой MRSN.

Среди близких аналогов можно выделить крупные фреймворки и наборы фреймворков для анализа логов. Например:

- fluent bit⁷;
- rsyslog⁸;
- ELK Stack⁹.

Использование этих фреймворков не особо подходит для решения задачи, так как они ориентированы на общую работу с логами, когда логи доступны напрямую с логирующих устройств, а не конкретно архивов со SCSI логами. Ниже приведены преимущества и недостатки в рамках задачи.

Плюсы:

- наличие инструментов для визуализации;
- возможность сохранения в базы данных.

Минусы:

- зачастую достаточно тяжеловесны и сложны в настройке и использовании;
- избыточная для задачи функциональность в виде сбора логов напрямую с различных устройств в реальном времени;

⁷<https://www.fluentbit.io/>

⁸<https://www.rsyslog.com/doc/index.html>

⁹<https://www.elastic.co/elastic-stack>

- правила парсинга для SCSI логов не заложены и их придется писать;
- ограниченные возможности для добавления новой функциональности из-за привязки к готовым фреймворкам.

Аналогами в области работы с архивами можно считать:

- AVFS¹⁰ — виртуальная файловая система с возможностью прямого доступа к файлам внутри архива;
- ugrep¹¹ — утилита, которая позволяет выполнять поиск по данным в архиве.

На основе этих инструментов можно создавать самописные скрипты для парсинга и поиска конкретных строк логов.

Плюсы такого подхода:

- легко и быстро написать.

Минусы:

- трудно масштабировать;
- сложно производить какие-либо трансформации над строчками логов.

2.2. SCSI

SCSI в общем смысле представляет собой набор стандартов описывающих передачу данных между компьютерами и периферийными устройствами. Из этого набора для анализа логов нужны только стандарты описывающие протоколы и команды, а именно стандарты от INCITS¹² SAM-5 [3], SBC-3 [4], SPC-4 [5].

В упрощенном виде взаимодействие инициатора (сервер) с целевым устройством (диск) по SCSI протоколу происходит в следующем виде:

¹⁰<https://avf.sourceforge.net/>

¹¹<https://github.com/Genivia/ugrep>

¹²International Committee for Information Technology Standards — ANSI аккредитованная организация по разработке стандартов

1. Инициатор шлет команду цели. Информация о команде содержит в CDB (Command Descriptor Block).
2. Целевое устройство отправляет код ответа.
3. В зависимости от кода ответа, инициатор может отправить еще одну команду. Например, для получения информации об ошибке (Request sense).

2.2.1. Устройство SCSI лога

Листинг 1: Пример dmesg LINUX SCSI лога.

```
kern :info : [Sat Mar 9 13:38:21 2024] sd 1:0:102:0:  
[sdkg] tag#5 Add. Sense: Unrecovered read error
```

LINUX SCSI логи, полученные через утилиты `dmesg` и `journalctl`, можно поделить на несколько последовательно идущих частей.

Часть созданная утилитой `dmesg` или `journalctl`. В основном содержит информацию об уровне важности лога и о том, откуда и когда он был получен. Информацию о всех возможных вариациях можно получить из документации `dmesg` [7] и `journalctl` [13].

Листинг 2: Пример dmesg части.

```
kern :info : [Sat Mar 9 13:38:21 2024]
```

Часть с информацией о SCSI устройстве. Состоит из LINUX SCSI адреса и имени устройства. Более подробную информацию можно получить в документации описывающей SCSI подсистему для LINUX KERNEL 2.4 [2].

Листинг 3: Пример части с информацией об устройстве.

```
sd 1:0:102:0: [sdkg]
```

Часть с информацией специфичной для конкретного типа SCSI лога. Включает всю информацию, касающуюся непосредственно SCSI протокола и его взаимодействия с LINUX KERNEL, например, отправленные на устройство команды и полученные ответы, захват и освобождение ресурса устройства, очередь команд и информация о диске.

К сожалению, логирование LINUX KERNEL SCSI не имеет документации, поэтому для анализа возможных вариантов приходится опираться на код SCSI midlayer LINUX KERNEL¹³ последней на момент написания данного отчета версии 6.12-RC1¹⁴ и более ранней версии 5.13¹⁵, которая была выбрана по причине того, что в версии 5.14 частично изменили устройство SCSI подсистемы и ее логирование, но так как новые версии LINUX еще не везде используются, необходимость в обработке логов с более ранних версий остается.

Листинг 4: Пример части со SCSI информацией.

```
tag#5 Add. Sense: Unrecovered read error
```

Среди SCSI специфичных логов можно выделить еще одну подгруппу, начинающуюся со слова **tag**. Данная подгруппа состоит из команд отправленных на диск и полученных ответов. Логи с одним и тем же значением **tag** содержат данные по одной и той же отправленной команде. Пример **tag** лога можно увидеть на листинге 4.

¹³<https://github.com/torvalds/linux/tree/master/drivers/scsi>

¹⁴<https://elixir.bootlin.com/linux/v6.12-rc1/source/drivers/scsi>

¹⁵<https://elixir.bootlin.com/linux/v5.13/source/drivers/scsi>

В LINUX KERNEL есть следующие виды `tag SCSI` логов:

- `CDB` — содержит бинарное представление CDB, по возможности также включая текстовое представление команды и дополнительную информацию;
- `Sense` — код возврата (`sense code`);
- `ExtdSense` — дополнительная информация об ошибке;
- `Result` — результат выполнения команды;
- `IOCompletion` — количество обработанных секторов и байт;
- `HostInfo` — информация о занятости и ошибках на хосте;
- `Send` — содержит адрес команды;
- `SetupRW` — информация о команде чтения/записи, номере стартового блока и количестве блоков;
- `Reading` — количество блоков на чтение и их размер;
- `SgDone` — количество переданных байт.

2.3. Используемые технологии

Утилита написана на C++23 и компилируется с использованием g++14 и clang 18. C++ был выбран из-за его активного использования в команде MRSN и скорости, так как логов может быть достаточно много и хотелось бы быстро их обрабатывать.

В качестве системы сборки был использован CMake, а менеджера зависимостей — vcpkg. В проекте также используется автоформатер clang-format и линтер clang-tidy.

2.3.1. Сторонние используемые библиотеки

Google Logging Library (glog) [1]

Простая в использовании библиотека для логирования.

Boost.Json [6]

C++ библиотека для работы с JSON. Простота в использовании и имеет неплохую производительность по сравнению с более популярной библиотекой `nlohmann_json`.

Magic Enum [11]

C++ библиотека для различных операций над `enum`.

libarchive [12]

Мощная C библиотека для работы с архивами. Умеет работать со всеми нужными форматами (`zip`, `tar`, `gz`) и может читать заголовки закодированных файлов, в отличие от `Boost.iostreams`, `zlib`, `libzip`.

CLI11 [9]

C++ библиотека для создания CLI. Предоставляет значительно большую функциональность, чем `Boost.Program_options`, `TCLAP`, `sxhopts`.

GoogleTest [10]

Фреймворк для тестирования приложений на C++.

3. Парсинг логов

Для начала выделим требования для модуля парсера:

1. Парсинг основных LINUX SCSI логов.

Парсер должен уметь разбирать SCSI tag логи перечисленные в 2.2.1.

2. Простота расширения.

В дальнейшем хотелось бы иметь возможность парсить не только SCSI логи и при этом не вносить изменения во множество различных участков кода.

3. Как можно меньше кода в конкретных реализациях.

Количество различных вариантов только SCSI логов больше десяти и желательно, чтобы реализация их парсинга занимала минимальное количество усилий.

4. Гибкость.

При дальнейшем расширении не лишней бы была возможность интегрировать в парсер библиотеки, которые реализуют ту или иную логику парсинга логов.

5. Приемлемая производительность.

Логи должны парситься за относительно небольшое время, чтобы конечному пользователю не пришлось долго ждать результата. В качестве ориентира взято около 100 мс на 1000 LINUX SCSI логов на компьютере, описанном в 3.5.

Парсер логов было решено оформить в виде библиотеки `logs_parser`, состоящей из трех библиотек:

- `parser_abstractions` — абстракции парсеров;
- `utils` — вспомогательная библиотека с различными полезными функциями;

- `parsers` — конкретные реализации парсеров.

Данная библиотека является основным компонентом утилиты, которую решено было назвать UNILOGS.

3.1. Абстракции парсеров

Исходя из требования к простоте расширения, пользуясь тем что строчку лога можно поделить на части, было принято решение сделать парсер как дерево из парсеров, отвечающих за парсинг части строки.

Рассмотрим подробнее. Первым запускается корневой парсер. Он парсит префикс строки, а далее передает постфикс и результат парсинга дочерним парсерам. Дерево парсеров можно увидеть на рисунке 2.

Результаты парсинга, как возвращаются пользователю наверх, так и сохраняются в листовых (крайних) парсерах, чтобы потом их можно было легко разделять и идентифицировать.

Такой подход позволяет не переписывать парсинг префикса для каждого SCSI лога и легко расширять дерево парсеров. Для этого достаточно написать свой парсер постфикса и просто добавить его в список дочерних парсеров парсера префикса.

Теперь рассмотрим введенные абстракции. Все их можно увидеть на UML¹⁶ диаграмме 1.

LogUnit

Представляет собой разобранную часть лога. Также содержит указатель на следующую разобранную часть. Таким образом `LogUnit` является односвязным интрузивным списком¹⁷. Имеет метод для сериализации имеющихся в нем данных в JSON с помощью библиотеки `Boost.Json`.

RootLogUnit

Является головой списка `LogUnit`-ов и содержит общую информацию о

¹⁶Unified Modeling Language (UML) — язык для визуального проектирования.

¹⁷<https://www.data-structures-in-practice.com/intrusive-linked-lists/>

логе (на данный момент только оригинальную строку). К `RootLogUnit` можно относиться как к разобранной строчке лога.

Parser

Интерфейс парсера. Имеет методы:

- `parse` — для начала парсинга строки;
- `getName` — имя парсера, требуется для дерева парсеров;
- `getUnits` — возвращает сохранённые в нем `RootLogUnit`-ы;
- `parseRemaining` — для продолжения парсинга.

Введен для гибкости, чтобы можно было ввести парсер, который не удовлетворяет древовидной структуре.

ParserHelper

Абстрактный класс, который реализует, часть методов `Parser` и отвечает за древовидную структуру парсеров.

Содержит указатель на родительский парсер и указатели на дочерние парсеры. Также добавляет метод `parseUnit`, отвечающий за парсинг префикса строки и в случае успеха возвращающий постфикс строки и `LogUnit` с разобранными данными префикса. Частично реализует методы:

- `parse` — при вызове создает `RootLogUnit` и вызывает `ParseRemaining`.
- `parseRemaining` — вызывает метод `parseUnit`, в случае его успеха, вызывает `parseRemaining` у дочерних парсеров.
- `getUnits` — объединяет результаты `getUnits` дочерних парсеров.

При инстанцировании также инстанцирует все дочерние парсеры.

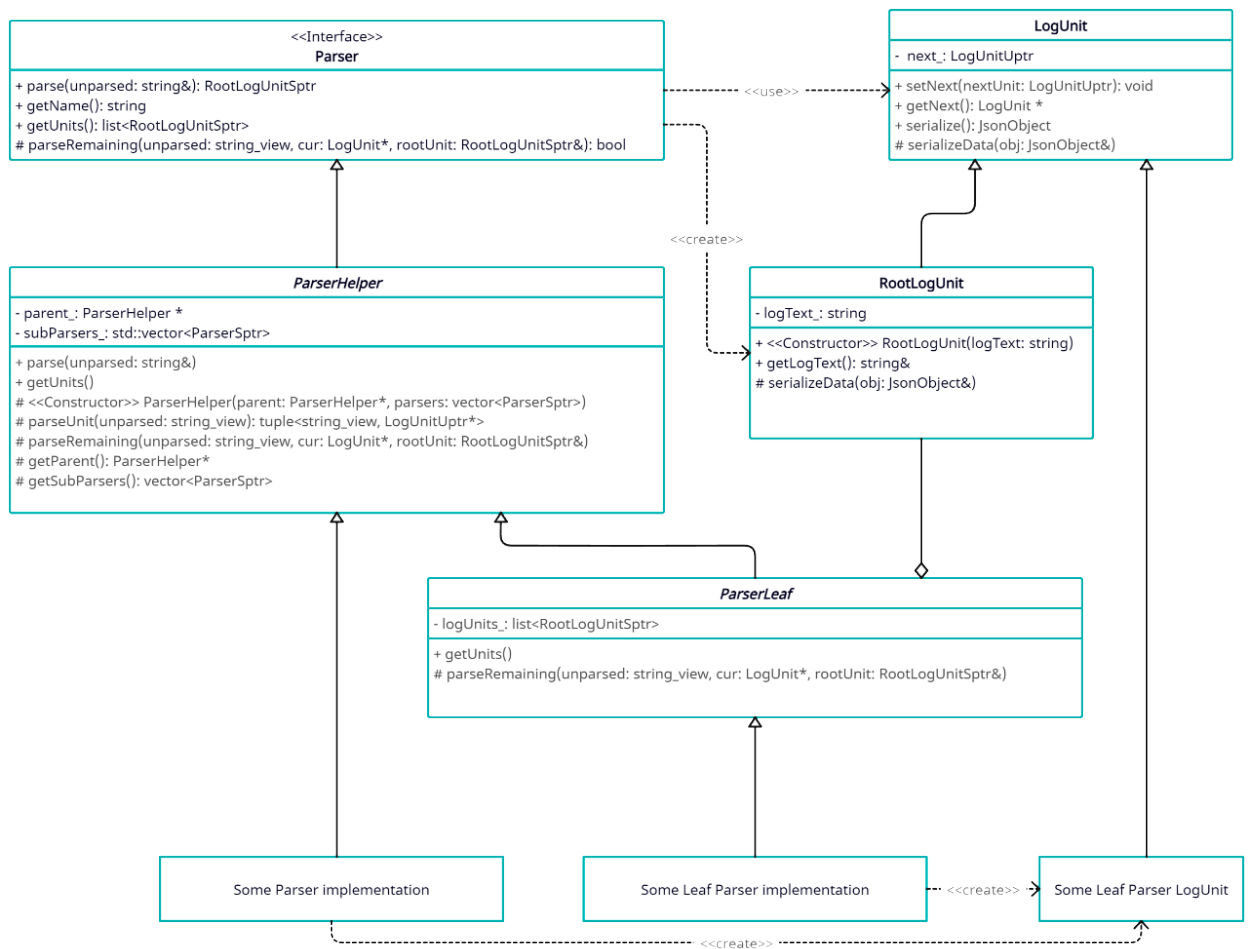


Рис. 1: UML диаграмма с абстракциями парсера.

ParserLeaf

Абстрактный класс, наследующийся от `ParserHelper`. Создан для упрощения реализации листовых (конечных) парсеров. Имеет поле `logUnits_` для сохранения разобранных логов.

Переопределяет методы:

- `parseRemaining` — возвращает результат `parseUnit` и сохраняет полученный `RootLogUnit`.
- `getUnits` — возвращает `logUnits_`.

3.2. Реализации парсеров

Реализации наследуются от `ParserHelper` и `ParserLeaf`.

Среди реализаций, можно выделить парсеры, которые не парсят какую-либо часть строки, а именно `UnsupportedParser` и `MainParser`. Они сделаны для служебных целей.

`UnsupportedParser` — это листовой парсер, который вызывается в том случае, если ни один из дочерних парсеров не подошел, и сохраняет неразобранную часть лога в `UnsupportedLogUnit`. Таким образом от переданной на вход строчки жадно парситься как можно большая часть, а неразобранная часть сохраняется в `UnsupportedLogUnit`.

`MainParser` — парсер, который имеет в качестве дочерних парсеров все основные парсеры, а именно `Dmesg`, `Journalctl`, `UnsupportedParser`. При стандартной эксплуатации UNILOGS парсинг запускается через `MainParser`.

На рисунке 2 можно увидеть итоговое дерево парсеров, в которое, для упрощения, не включены `MainParser` и `UnsupportedParser`. А на рисунке 3 можно увидеть пример итогового пути лога через парсеры. Листинг 5 содержит пример с сериализованным результатом работы парсера.

Реализаций парсеров создавались на основе функций логирования SCSI midlayer в LINUX KERNEL. Более подробно в 2.2.1. Парсеры разбирают информацию в логах и также восстанавливают информацию, которая не была явна указана в логе, например, opcode¹⁸ по строковому представлению команды. Для разбора строки парсеры используют регулярные выражения в модифицированной C++ грамматике ECMAScript [8].

3.3. Utils

Библиотека для удобства разработки реализаций парсеров. Содержит функции, макросы и классы для работы с `std::string_view`, регулярными выражениями, сериализацией в JSON.

¹⁸числовой код операции

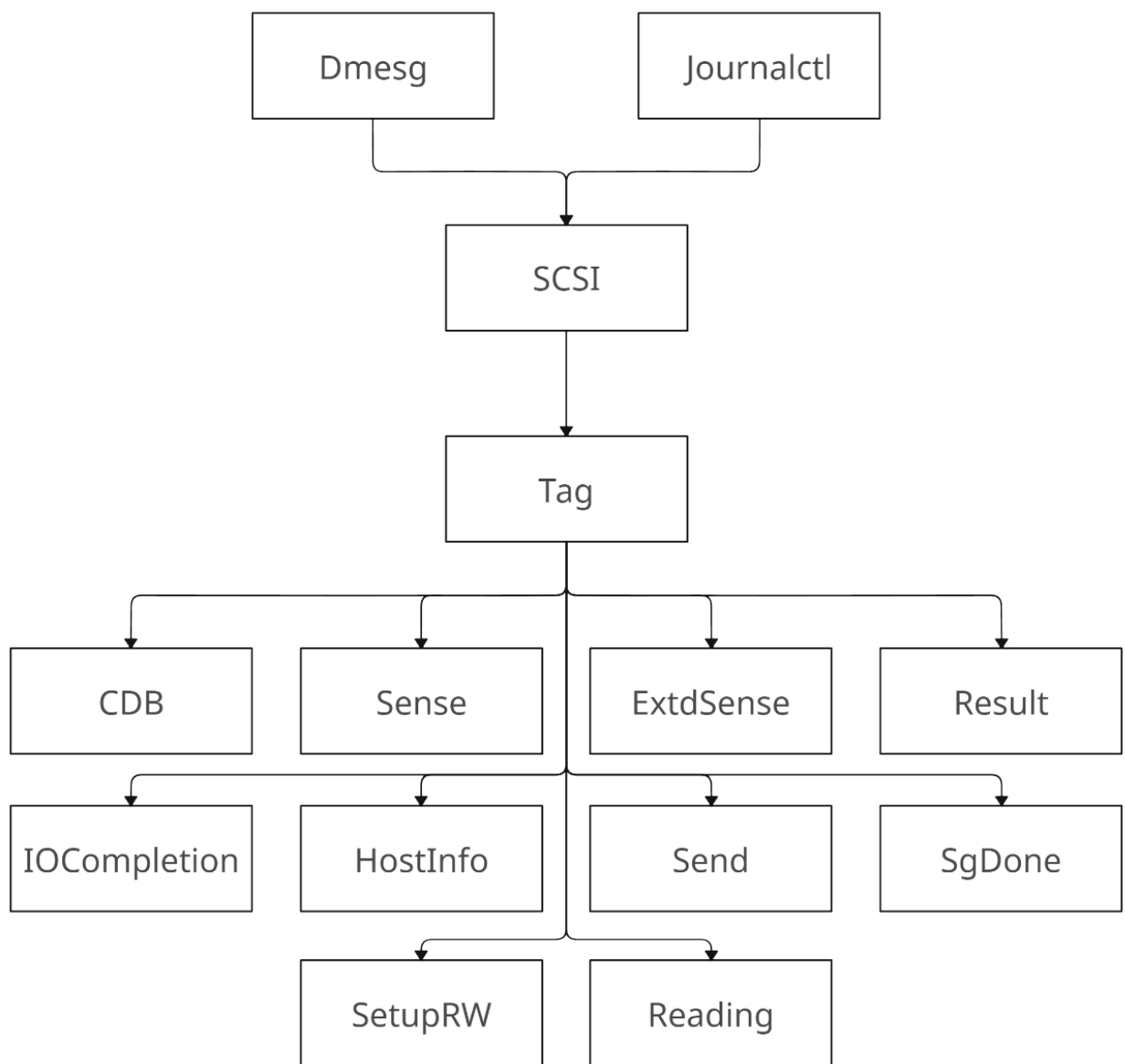


Рис. 2: Упрощенное дерево парсеров UNILOGS.

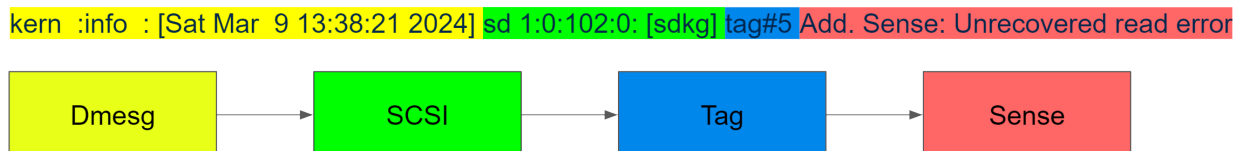


Рис. 3: Разбор dmesg SCSI лога.

Листинг 5: Пример сериализованного результата работы парсера. Оригинальную строчку лога можно увидеть в поле log line.

```
{
  "log line": "kern :info : [Sat Mar  9 13:38:21 2024]
sd 1:0:102:0: [sdkg] tag#5 Add. Sense: Unrecovered read error",
  "next": {
    "next": {
      "name": "dmesg",
      "facility": "Kern",
      "level": "Info",
      "timestamp": "Sat Mar 09 13:38:21 2024",
      "next": {
        "name": "SCSI",
        "address": {
          "host": 1,
          "bus": 0,
          "target": 102,
          "lun": 0
        },
        "device name": "sdkg",
        "next": {
          "tag": 5,
          "next": {
            "name": "Extended Sense",
            "error": "Unrecovered read error",
            "format": null,
            "ASC": "0x11",
            "ASCQ": "0x00",
            "vendor specific ASC": false,
            "vendor specific ASCQ": false
          }
        }
      }
    }
  }
}
```

3.4. Тестирование

Для парсера были написаны тесты проверяющие базовую функциональность:

- создание `RootLogUnit` при парсинге;
- правильность структуры `LogUnit`;
- правильность парсинга нескольких строк подряд;
- правильность работы `UnsupportedParser`.

Также для каждого парсера были созданы по два вида тестов:

- Тесты на проверку соответствующих этому парсеру различных вариантов строки.

В этих тестах нужный парсер запускается как главный парсер, и ему передается кусок строки лога. Результат сериализуется и сравнивается с ожидаемым JSON значением.

- Тесты на проверку запуска парсера в дереве парсеров.

Создается `MainParser` и ему передается строка, для которой должен был запуститься этот парсер. Результат сериализуется и сравнивается с JSON значением в соответствующем тесту файле.

Тестовое покрытие было оценено с помощью утилиты `gcov`¹⁹ и составило 96.3%.

3.5. Производительность

Тестирование производилось с помощью утилиты `hyperfine`²⁰ на компьютере с процессором 12th Gen Intel(R) Core(TM) i7-12700H 2.30 GHz и с 32 ГБ оперативной памяти под управлением операционной системы Windows 10 Pro 22H2 через WSL 2 Ubuntu 24.04. UNILogs

¹⁹<https://github.com/gcovr/gcovr>

²⁰<https://github.com/sharkdp/hyperfine>

компилировался в **Release** конфигурации с параметром компилятора **-O2**. В качестве источника выступал **txt** файл с 2123 строчками SCSI логов. Было сделано 10 прогревочных (warmup) и 100 замеряемых запусков, $CI = 0.95$.

среднее время: 29.2 мс $\pm$ 0.8 мс

минимальное время: 27.8 мс

максимальное время: 31.2 мс

Из этого можно сделать вывод, что утилита показывает удовлетворительные результаты по скорости.

4. Работа с архивами

Компонента для работы с архивами должна по архивам с форматами `zip`, `tar` и файлам `gz` обеспечивать доступ к текстовым файлам с логами, которые в дальнейшем можно было бы передать парсеру. Также нужна фильтрация по предикату, чтобы можно было запускать парсер не на всех возможных файлах, а только на нужных.

Из этих требований появляются следующие требования к инструментарию.

- возможность работать с форматами `zip`, `tar`, `gz`;
- возможность получать имя архива из метаданных.

Библиотеки с C++ API, которая бы удовлетворяла этим требованиям, найдено не было. Поэтому выбор пал на C библиотеку `libarchive` (2.3.1).

4.1. Реализация

Назовем компоненту для работы с архивами `FilesFilter`. Чтобы иметь возможность доступа к текстовым файлам в архивах со сложной структурой, она должна уметь рекурсивно проходиться по файлам в архиве и возвращать распакованные текстовые файлы. Для удобства пользователей также было добавлено требование на запуск над папкой.

Для работы с архивами был создан отдельный класс `Archive`.

4.1.1. Класс `Archive`

`Archive` абстрагирует работу с `libarchive`, распаковывает и возвращает файлы удовлетворяющие переданному предикату при вызове метода `next`.

К сожалению, `libarchive` не содержит функций, которые осуществляют рекурсивный доступ к файлам. Поэтому если при распаковке архива встретился файл, который также является архивом, то он распако-

вываается в оперативную память и на его основе конструируется другой класс `Archive`.

Распакованный архив хоть и имеет большой размер, но имеемые реальные данные показали, что в среднем размер вложенных архивов не превышает 400 МБ, что является приемлемым показателем.

`Archive` может быть сконструирован на основе пути к архиву или на основе массива в памяти. Оба конструктора также имеют параметр `namePredicate`, отвечающий за предикат, которому должны удовлетворять пути конечных текстовых файлов.

4.1.2. Класс `FilesFilter`

Так как размер входных данных может измеряться гигабайтами, то `FilesFilter` должен генерировать файлы лениво, чтобы минимизировать нагрузку на оперативную память. Поэтому `FilesFilter` был сделан ленивым итератором, удовлетворяющим концепту `std::input_iterator`²¹ для совместимости с `std::ranges`²². Эта совместимость позволяет использовать множество полезных алгоритмов из 20 и 23 стандарта C++.

`FilesFilter` конструируется на основе структуры `FilesFilterParams`. Она содержит предикат для путей текстовых файлов и путь к анализируемому файлу. Введение `FilesFilterParams` позволяет в будущем безболезненно расширить параметры `FilesFilter`.

Результатом работы модуля является класс `File` — абстракций над `std::istream`²³ с дополнительными метаданными. Так как из `istream` можно получать строки, это позволяет запустить парсер над `File`.

Далее рассмотрим случай если `FilesFilter` передана папка, так как другие случаи более тривиальны.

При инкрементировании итератора запускается цикл с рекурсивным проходом по папкам с помощью `std::filesystem::recursive_directory_iterator`²⁴. Цикл оста-

²¹https://en.cppreference.com/w/cpp/iterator/input_iterator

²²<https://en.cppreference.com/w/cpp/ranges>

²³https://en.cppreference.com/w/cpp/io/basic_istream

²⁴https://en.cppreference.com/w/cpp/filesystem/recursive_directory_iterator

навливается, если находится нужный элемент или больше нечего рассматривать.

Действия на итерации зависят от того, лежит ли `Archive` в поле `archive_`.

- Если нет, тогда просматривается следующий файл в текущей рассматриваемой папке, если он текстовый, то он проверяется предикатом и в случае успеха возвращается как `File`. Если файл — архив, то он помещается в поле `archive_`.
- Если да, тогда вызывается метод `next` у `archive_` и его результат возвращается как `File`. Если метод `next` ничего не вернул, то поле `archive_` отчищается.

4.2. Тестирование

Для `FilesFilter` были написаны тесты проверяющие работу с текстовыми файлами, папками и файлами `zip`, `tar.gz`, `gz` на:

- корректность чтения имен файлов;
- корректность чтения данных самих файлов;

Тестовое покрытие, полученное через `gscov`, составляет 98,9%.

5. Command line interface

CLI должно дать возможность по источнику логов и фильтру выдавать разобранные JSON SCSI логи. А также предоставить опцию `—help` с описанием взаимодействия.

5.1. Реализация

Для реализации CLI была выбрана библиотека CLI11 (2.3.1), она содержит множество приятной функциональности, например, поддержку `std::optional` для аргументов, и уже реализует `—help` на основе описания аргументов.

UNILOGS CLI имеет две подкоманды (subcommand) `source` и `parse`, которые отвечают за взаимодействие с компонентами `FilesFilter` и `logs_parser` соответственно. Использование UNILOGS можно увидеть на листинге 6.

Для взаимодействия с подкомандами был реализован абстрактный класс `SubCommand`. Он содержит ссылку на класс из CLI11 — `CLI::App`, определяющий аргументы приложения.

На основе `SubCommand` были созданы классы `ParseCommand` и `SourceCommand`, реализующие логику подкоманд `source` и `parse` соответственно. Эти классы являются функторами. При конструировании они принимают ссылку на `CLI::App` и добавляют соответствующие подкомандам параметры командной строки. При вызове функторы парсят аргументы и вызывают соответствующую логику.

5.1.1. source

`source` является подкомандой, которая определяет источник логов. В случае, если `source` не был передан, в качестве источника выступает стандартный поток ввода.

При передаче `source` в качестве источника логов может выступать строка, файл, папка или архив. Также для фильтрации файлов можно

Листинг 6: Результат вывода `unilogs --help`

```
UniLogs! Helps you analyze your SCSI logs!
Usage: ./unilogs [OPTIONS] SUBCOMMANDS

Options:
  -h, --help          Print this help message and exit

Subcommands:
  source              Specify the source for UniLogs,
                     standart input by default
  parse              Parse logs
```

передать параметр `filter` с `glob` шаблоном, на его основе будет сгенерирован предикат для имен файлов.

`SourceCommand` при запуске вызывает `FilesFilter` и возвращает лениво сгенерированные `File`, которые дальше передаются в `ParseCommand`.

Использование подкоманды `source` можно увидеть на листинге 7.

5.1.2. `parse`

`parse` запускает парсинг файлов на основе данных полученных из `source` и выдает JSON результат. По умолчанию результат выводиться в стандартный поток вывода, также возможен вывод в файл. Есть опция для форматирования JSON в человекочитаемый (`pretty print`) вид. Использование `parse` на листинге 8.

5.2. Тестирование

CLI тестировалось вручную. Основным материалом для тестирования выступали логи полученные через `journalctl` по SCSI запросам, отправленным с помощью `sg_utils`²⁵ на виртуальный диск, созданный

²⁵https://sg.danny.cz/sg/sg3_utils.html

Листинг 7: Результат вывода `unilogs source --help`

Specify the source for UniLogs

Usage: `./unilogs source [OPTIONS]`

Options:

<code>-h, --help</code>	Print this help message and exit
<code>-a, --archive TEXT:FILE</code>	Path to archive with logs
<code>-d, --dir TEXT:DIR</code>	Path to directory containing files with logs
<code>-f, --file TEXT:FILE</code>	Path to file with logs
<code>-s, --string TEXT</code>	Use given string as a source
<code>--filter TEXT</code>	Glob filter for files name in dir or archive

Листинг 8: Результат вывода `unilogs parse --help`

Parse logs

Usage: `./unilogs parse [OPTIONS]`

Options:

<code>-h, --help</code>	Print this help message and exit
<code>-o, --output TEXT</code>	File to write the result
<code>-p, --pretty</code>	Pretty print the serialized result

через `targetcli`²⁶. Эти логи передавались утилите как текстовый файл, архив с логами внутри, а также напрямую из `journalctl` через `LINUX pipe`²⁷. Также утилита была запущена над несколькими реальными архивами с логами, полученными с TATLIN.UNIFIED.

²⁶<https://manpages.ubuntu.com/manpages/bionic/man8/targetcli.8.html>

²⁷<https://manpages.ubuntu.com/manpages/lunar/ru/man7/pipe.7.html>

6. Заключение

6.1. Результаты

Была разработана утилита UNILOGS для анализа SCSI логов, а именно:

- Был спроектирован и реализован модуль для парсинга логов. Данный модуль может парсить SCSI tag логи и является легко расширяемым.
- Был спроектирован и реализован модуль **FilesFilter** для работы с папками и архивами.
- Был спроектирован и реализован CLI.

На момент написания работы UNILOGS не используется активно, так как основная работа по анализу логов с целью развития MRSN запланирована на 2025 год.

Код UNILOGS находится под NDA и является собственностью компании YADRO, поэтому не может быть предоставлен.

6.2. Планы развития

В дальнейшем планируется:

- объединение связанных по смыслу логов;
- сохранение в базы данных;
- построение диаграмм по разобранным логам;
- GUI.

Список литературы

- [1] Deitsch Sergiu. — Документация Google Logging Library. — URL: <https://google.github.io/glog/stable/> (дата обращения: 3 октября 2024 г.).
- [2] Gilbert Douglas. — The Linux 2.4 SCSI subsystem HOWTO. — URL: <https://tldp.org/HOWTO/SCSI-2.4-HOWTO/names.html> (дата обращения: 30 сентября 2024 г.).
- [3] INCITS. — SCSI Architecture Model — 5 (SAM-5), 2016. — INCITS 515-2016.
- [4] INCITS. — SCSI Block Commands — 3 (SBC-3), 2014. — INCITS 514-2014.
- [5] INCITS. — SCSI Primary Commands — 4 (SPC-4), 2015. — INCITS 513-2015.
- [6] Vinnie Falco Krystian Stasiowski Dmitry Arkhipov. — Документация Boost.JSON. — URL: <https://www.boost.io/doc/libs/release/libs/json/doc/html/index.html> (дата обращения: 3 октября 2024 г.).
- [7] Zak Karel. — Справочная страница утилиты dmesg. — URL: <https://manpages.ubuntu.com/manpages/bionic/man1/dmesg.1.html> (дата обращения: 30 сентября 2024 г.).
- [8] Грамматика модифицированных регулярных выражений ECMAScript в C++. — URL: <https://en.cppreference.com/w/cpp/regex/ecmascript> (дата обращения: 2 октября 2024 г.).
- [9] Документация CLI11. — URL: <https://cliutils.github.io/CLI11/book/> (дата обращения: 3 октября 2024 г.).
- [10] Документация Google Test. — URL: <https://google.github.io/googletest/> (дата обращения: 3 октября 2024 г.).

- [11] Документация Magic Enum. — URL: https://github.com/Neargye/magic_enum/blob/master/README.md (дата обращения: 3 октября 2024 г.).
- [12] Документация libarchive. — URL: <https://libarchive.org/> (дата обращения: 3 октября 2024 г.).
- [13] Справочная страница утилиты journalctl. — URL: <https://manpages.ubuntu.com/manpages/focal/man1/journalctl.1.html> (дата обращения: 30 сентября 2024 г.).