|  |  |  |
| --- | --- | --- |
| Министерство образования и науки Российской Федерации | | |
| федеральное государственное бюджетное образовательное учреждение высшего образования | | |
| «Иркутский государственный университет» | | |
| (ФГБОУ ВО «ИГУ») | | |
| Факультет бизнес-коммуникаций и информатики | | |
|  | | |
|  | |
| **ОТЧЕТ ПО ПРОИЗВОДСТВЕННОЙ ПРАКТИКЕ**  **Б2.В.04(П) Технологическая (проектно-технологическая) практика**  **по направлению 09.03.03 «Прикладная информатика»**  **(прикладной бакалавриат)**  **профиль «Прикладная информатика (Разработка программного обеспечения)»** | |
|  | |
| Разработка фреймворка для построения конвейера обработки данных | |
|  |  |
|  | Студент 4 курса очной формы обучения, группа 14421-ДБ |
|  | \_\_\_\_\_\_\_\_\_\_ В. С. Сизых |
|  |  |
|  | Руководитель: к.ф.-м.н., доцент  \_\_\_\_\_\_\_\_\_\_ А.Г. Балахчи |
|  | |
|  | Работа защищена: |
|  | «\_\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_ 2024 г. |
|  | С оценкой \_\_\_\_\_\_\_\_\_\_\_\_\_ |
|  |  |
|  | |

Оглавление

[Введение 3](#_Toc162479322)

[1 обзор предметной области 5](#_Toc162479323)

[1.1 Требования к проекту 5](#_Toc162479324)

[1.2 Обзор существующих решений 6](#_Toc162479325)

[1.3 Выбор языка программирования 11](#_Toc162479326)

[2 Обзор инструментария 12](#_Toc162479327)

[2.1 Автоматизация сборки программного обеспечения 13](#_Toc162479328)

[2.2 Сериализация данных 14](#_Toc162479329)

[2.3 Представление данных внутри конвейера 18](#_Toc162479330)

[2.4 Брокер сообщений 21](#_Toc162479331)

[2.5 Соединение стадий конвейера между собой 22](#_Toc162479332)

[2.6 Оптимизация процесса управления потоками 24](#_Toc162479333)

[2.7 Обработка и манипуляция большим количеством многомерных данных 24](#_Toc162479334)

[2.8 Отслеживание утечек памяти в проекте 25](#_Toc162479335)

[3 Разработка примитивных модулей для будущего фреймворка 26](#_Toc162479336)

[3.1 Генератор данных 26](#_Toc162479337)

[3.2 Примитивная реализация однопоточного конвейера 27](#_Toc162479338)

[3.3 Набросок конфигурационного файла 29](#_Toc162479339)

[Выводы по главе 31](#_Toc162479340)

[4 Разработка фреймворка для построения конвейера обработки данных 31](#_Toc162479341)

[4.1 Разработка примитивного конвейера на с++ 31](#_Toc162479342)

[4.2 Осуществление получения данных с генератора 32](#_Toc162479343)

[4.3 Построение многопоточного конвейера 34](#_Toc162479344)

[4.4 Мониторинг загруженности конвейера 36](#_Toc162479345)

[4.5 Написание CMake скрипта 39](#_Toc162479346)

[4.6 Архитектура сущностей фреймворка 41](#_Toc162479347)

[Заключение 44](#_Toc162479348)

[Список использованных источников 45](#_Toc162479349)

[Приложения 49](#_Toc162479350)

[приложение б 51](#_Toc162479351)

# Введение

Конвейер обработки данных [1] - программа, которая принимает на вход поток данных и затем производит над ними ряд преобразований, после чего преобразованные данные сохраняются в хранилище. Ряд преобразований над данными производится за счет отдельных функциональных единиц, которые содержат определенную функцию, описывающую способ преобразования данных. Такие функциональные блоки соединяются между собой при помощи коннекторов, которые позволяют передавать данные от одного блока к другому.

Фреймворк - программа, позволяющая сконфигурировать свой конвейер, в том числе создать свои функциональные блоки, распределить ресурсы внутри конвейера, и задать направление данных внутри конвейера при помощи коннекторов.

Институт солнечно-земной физики (ИСЗФ) - организация, в которой проводится большое количество наблюдений и исследований. На солнечных телескопах, чаще всего, данные представляют собой двумерные изображения, полученные в разных спектральных диапазонах при помощи цифровых камер (КМОП, ПЗС и пр.), и несущие информацию о разных физических процессах, происходящих на Солнце. В зависимости от целей наблюдений могут как варьироваться некоторые этапы обработки данных, так и добавляться новые этапы. Существующее решение на предприятии заказчика позволяет автоматизировать этот процесс, но имеет ряд недостатков. На момент начала выполнения выпускной квалификационной работы, обработка данных осуществляется при помощи написанного в ИСЗФ программного обеспечения, которое позволяет многопоточно обрабатывать поступающие данные. Однако, последовательность обработки данных представляет собою фиксированную последовательность вызова функций. Изменение последовательности обработки требует перекомпиляции существующего решения. Помимо этого, так как обработка данных происходит в режиме реального времени, необходимо предусмотреть следующие возможности программы:

1. необходимо отслеживать количество ресурсов, потребляемых конвейером, например, такие как количество потребляемой памяти, или общая загруженность стадий конвейера. Такой мониторинг позволит реализовать возможность масштабирования количества воркеров (рабочих потоков) конвейера в случае, если конвейер не справляется с нагрузкой. Если после масштабирования конвейера, все равно не удается снизить его загруженность, то фреймворк выдаст предупреждение для пользователя;
2. необходимо предоставить возможность изменения последовательности обработки данных без необходимости в перекомпиляции фреймворка. Такой способ позволит быстро адаптировать конфигурацию конвейера, то есть внести изменения в последовательность обработки данных, а также применить их, загрузив новую или измененную конфигурацию. Конфигурация конвейера представляет собою json-файл, в котором описаны стадии (этапы), содержащиеся в конвейере и их вызываемые функции. Также там содержится описание связей между стадиями, что позволяет фреймворку выстроить последовательность обработки данных.

Текущее решение обладает рядом недостатков, описанных выше, поэтому технологии по созданию фреймворка для построения конвейера обработки многомерных данных обеспечивают **новизну** решения.

**Цель** данной работы заключается в разработке архитектуры и написании фреймворка для построения конвейера обработки многомерных данных.

**Задачами** данной работы являются:

1. анализ существующих библиотек, позволяющих реализовать данное решение, а также подробное их изучение;
2. разработка архитектуры фреймворка с учетом подобранных библиотек и требований заказчика;
3. кодирование фреймворка при помощи гибкой (agile) методологии разработки ПО и проведение тестов на оборудовании заказчика.

Отсутствие на предприятии заказчика необходимого программного продукта, который бы позволил использовать удобные и понятные файлы конфигурации для построения конвейера обработки многомерных данных обуславливает **актуальность** темы дипломной работы.

**Объектом исследования** данной работы являются технологии, используемые при проектировании архитектуры приложений, а также при написании фреймворков, позволяющих построить конвейер обработки многомерных данных.

**Предметом исследования** являются проектирование архитектуры и разработка фреймворка для построения конвейера обработки многомерных данных.

**Практическая значимость** исследования заключается в реализации эффективного подхода по построению и модификации конвейера для обработки потока многомерных данных в режиме реального времени.

# обзор предметной области

## Требования к проекту

Фреймворк разрабатывается для предоставления возможности его дальнейшего использования на предприятии заказчика. ЭВМ заказчика работает на базе операционной системы Linux, поэтому она будет единственной целевой платформой проекта. Данные, которые будут поступать на вход конвейеру, снимаются с цифровых камер. Цифровые камеры отправляют данные, используя библиотеку-брокер ZMQ. Вследствие этого, реализация приема данных при помощи ZMQ обязательна для фреймворка. В качестве библиотеки для сериализации и десериализации данных, заказчик использует Cap’n Proto, поэтому ее интеграция в фреймворк обязательна. Если будут найдены более быстрые решения, то их реализация может также быть интегрирована в проект. Исходя из этого, перечень требований, следующий:

* целевая платформа Linux;
* реализация приема данных при помощи ZMQ;
* реализация трансформации данных при помощи библиотеки Cap’n Proto.

## Обзор существующих решений

Реализация последовательности обработки не является новой задачей. Существуют различные решения, которые позволяют задать порядок вычислений в виде графа или в виде простой последовательности вызовов. Большинство найденных проектов [2] написаны при помощи таких языков, как: java, scala, и python. В текущем разделе будут рассмотрены некоторые из таких проектов. Основной целью при их рассмотрении является изучение применяемых подходов и практик, которые использовались при создании подобных фреймворков. Также важно обратить внимание на архитектуру проектов, так как она определяет структуру и возможность расширяемости программы. Данная информация позволит сократить количество ошибок, в том числе при проектировании, которые могут возникнуть в ходе реализации проекта.

Apache Flink

Apache Flink представляет собою фреймворк и механизм для построения распределенной системы вычислений, где важно обеспечивать высокую надежность обрабатываемых данных, а также отказоустойчивость системы. Фреймворк обладает надежной архитектурой и набором полезных функций. Одной из таких функций является, отслеживание состояния над потоком данных, что позволяет в режиме реального времени производить мониторинг загруженности менеджера задач (Task manager). В качестве отслеживаемых параметров используются: количество обработанных записей, время выполнения операций, а также общая загруженность менеджера. Все эти метрики фреймворк позволяет отслеживать посредством консольного приложения или браузера, используя поднятие локального сервиса. Также фреймворк периодически сохраняет свое состояние, используя для этого базу данных, что позволяет восстановить конвейер в случае сбоя, и не нарушить консистентность данных [3].

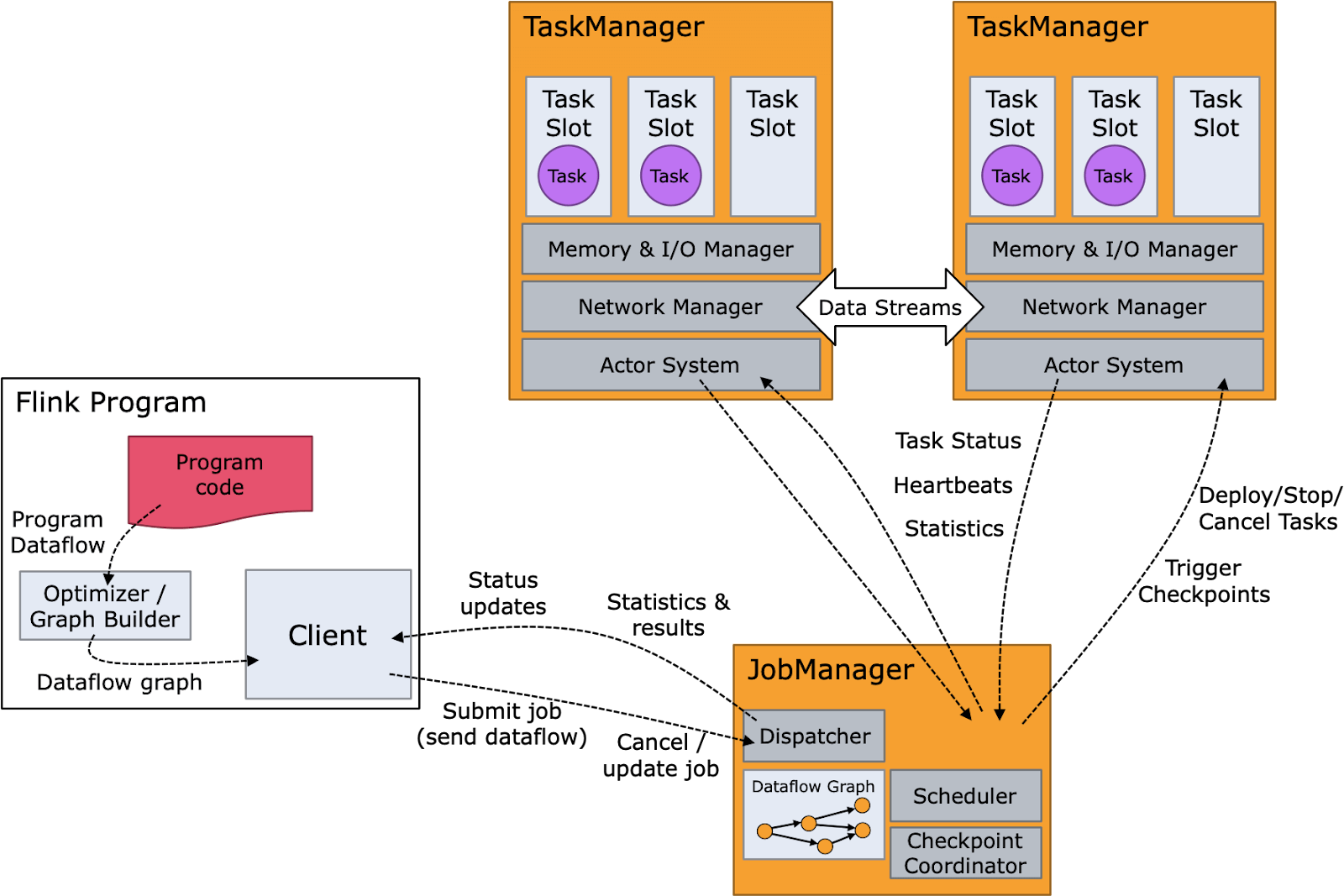


Рисунок 1 – Схема работы менеджеров в фреймворке Apache Flink

При построении конвейера, есть возможность программировать потоки данных, как в параллельном режиме, так и в конвейерном. При конвейерной организации потока, данные обрабатываются последовательно, на одном узле (вычислительной единице), проходя цепочку операторов в определенном порядке. Такой подход обеспечивает эффективное использование ресурсов и ускоряет обработку данных, так как нет задержек при передаче данных между узлами. В случае параллельной организации, данные могут обрабатываться на разных узлах, при этом каждый оператор становится независимым от остальных, то есть выполняется на выделенном ему узле. Такой подход позволяет равномерно распределить нагрузку между узлами системы, и обрабатывать бОльшие объемы данных.

Apache Flink является фреймворком с открытым исходным кодом [4]. Для запуска Flink необходимы компоненты java, так как задачи запускаются на виртуальных java-машинах. Для пробного запуска фреймворка достаточно создать кластер, который будет работать в фоне (процесс-демон), а затем указать ему файл формата jar, представляющий собой код программы на языке программирования java. Код для своих заданий (task) можно писать на языках программирования java, scala и python при помощи SDK от разработчиков Flink. Фреймворк обладает хорошей документацией [5], где помимо описания инструкций при работе с программой, также есть материал о его внутреннем устройстве.

Luigi

Luigi - фреймворк для построения последовательностей выполнения задач. Разработан в компании spotify и является open-source проектом [6]. Luigi написан при помощи языка программирования python, и главными его преимуществами являются простота использования и возможность создавать зависимости между выполняемыми задачами.

Установка фреймворка происходит очень удобным способом: при помощи пакетного менеджера pip. Для использования необходимы знания ООП и синтаксиса python. Задачи, которые будут запускаться на кластере, описываются при помощи классов. Для этого создается класс задачи и наследуется от базового класса, предоставляемого фреймворком. Для того, чтобы описать действия и атрибуты задачи, используется перегрузка методов базового класса. Например, в методе run() записывается код, который выполняется при запуске задачи. Также, фреймворк предоставляет гарантию однократного выполнения задач, когда это необходимо. Такая гарантия может использоваться, когда задача очень ресурсоемкая, а ее результат может пригодиться позже для другой задачи. Для того, чтобы задача выполнялась только один раз, необходимо указать место, где будет сохранен результат ее выполнения. Для этого необходимо перегрузить метод output(), и вернуть из него объект-наследник класса Target - это может быть локальный файл, или соединение с базой данных.

Еще одним важным преимуществом является возможность создания зависимых задач [7]. Такие задачи отличаются от обычных тем, что они смогут начать свое выполнение только тогда, когда все задачи, от которых они зависят, закончили свое выполнение. Фреймворк также имеет планировщик, который планирует запуск задач, а также визуализирует их зависимости и статус выполнения при помощи графа зависимостей.

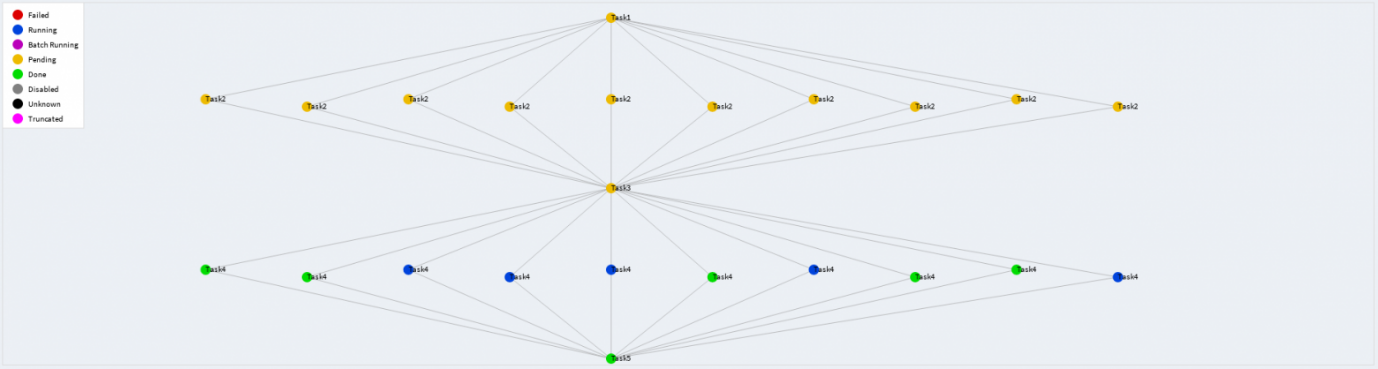


Рисунок 2 – Пример графа зависимостей, количество воркеров равно 4

Как и Flink, Luigi имеет возможность конвейерной или параллельной обработки данных. Выбор осуществляется при помощи установки значения для аргумента workers. Если количество воркеров больше одного, то данные обрабатываются параллельным способом, иначе конвейерным.

Фреймворк Luigi не предназначен для обработки больших объемов данных, но он имеет функционал, которого нет у похожих проектов, и который возможно стоит реализовать в настоящем проекте.

Apache kafka

Kafka - распределенная система, предназначенная для обработки потоков данных в режиме реального времени. Она не представляет собою фреймворк, позволяющий построить конвейер обработки данных, однако она является брокером сообщений, то есть посредником между сервисами, направляющими данные, и сервисами, которые их запрашивают. Kafka легко масштабируется, устойчива к отказам, а также имеет высокую пропускную способность.

Для достижения данных показателей, в Kafka была детально продумана архитектура для механизма по приему, хранению и отправке сообщений. Для соединения издателей (producer) и подписчиков (consumer), Kafka создает канал. Один канал может содержать несколько разделов, на которые подписываются подписчики. При этом, подписчики могут следить за несколькими разделами. Это позволяет более гибко направлять поток сообщений от издателя к подписчикам [8]. Для избежания потери сообщений при больших объемах передаваемых данных, Kafka дублирует разделы при помощи создания их репликаций. Подчиненные разделы (реплики) сохраняют себе всю информацию, которую имеет лидер. Тогда, в случае сбоя одного из разделов, данные будут использоваться с репликационного раздела, и система сможет продолжить работу.

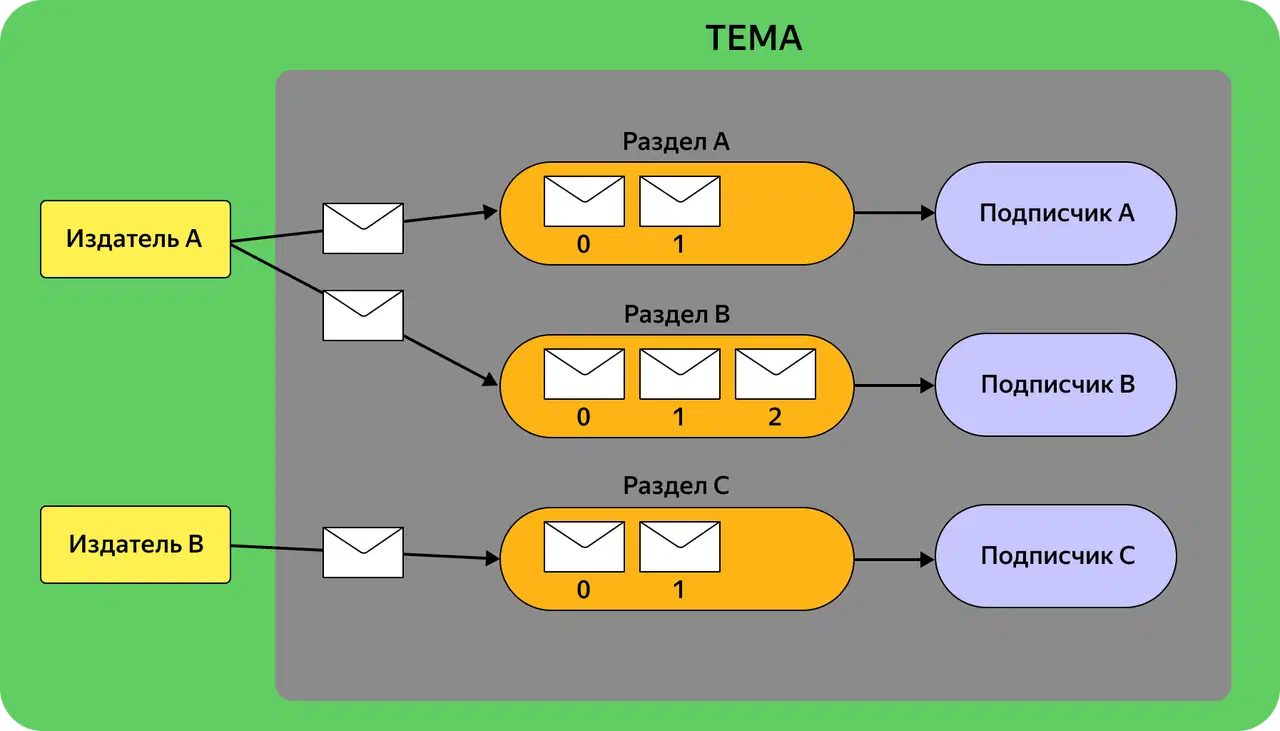


Рисунок 3 – Схема распределения сообщений в Kafka

Для обмена сообщениями может использоваться две разных модели запросов: pull или push [9]. В первом случае, подписчики сами запрашивают данные у брокера, что позволяет им эффективно контролировать собственную загрузку. Помимо этого, такая модель позволяет группировать сообщения в батчи (batch), благодаря чему достигается более высокая пропускная способность. Недостатком pull модели является разбалансированность нагрузки, так как все подписчики обращаются к брокеру хаотично.

В случае push модели, запрос инициируется брокером. Брокер делает запросы к своим подписчикам, чтобы выслать им новую порцию данных. Здесь есть риск, что подписчики могут перегружаться, поэтому требуются дополнительные механизмы для предотвращения этого.

В Kafka используется модель pull. Но главной отличительной особенностью Kafka от других брокеров является то, что Kafka не удаляет сообщения по мере их обработки, они продолжают храниться на брокере, что позволяет обрабатывать эти сообщения разными подписчиками в разных контекстах.

Для того, чтобы данные не обрабатывать повторно, а также сохранить их правильный порядок обработки, для каждого набора данных создается запись, представляющая некий отпечаток. Такая запись содержит ключ, значение, таймстамп (timestamp), а также опциональный набор метаданных.

## Выбор языка программирования

При выборе языка программирования для реализации проекта необходимо учитывать особенности языка программирования, требования проекта и внешние факторы. Среди основных рассматриваемых языков программирования были: Java, Python, Rust и C++.

Python является одним из самых популярных языков программирования, благодаря своей простоте, гибкости и удобству, но несмотря на это, он имеет низкую производительность. Все дело в том, что он является интерпретируемым языком, поддерживает динамическую типизацию [10], а также интерпретатор осуществляет глобальную блокировку (GIL), которая не позволяет эффективно использовать выполнение кода в нескольких потоках [11]. Все это не допустимо для проекта с высокими требованиями к производительности и скорости исполнения кода.

При рассмотрении языка Java были учтены следующие его преимущества: кроссплатформенность, неплохая скорость выполнения кода, встроенные механизмы безопасности, а также обширная стандартная библиотека, предоставляющая большой функционал. Недостатком языка оказалось потребление довольно большого количества ресурсов при исполнении кода, что может быть вызвано необходимостью Java запускать и поддерживать работу jvm-машины. Так, согласно бенчмарку [12], для выполнения аналогичного кода с C++, Java требует в среднем в 3-4 раза больше времени, и в 8-10 раз больше памяти. Так как для проекта критично наличие свободных ресурсов, таких как процессорное время и объем свободной памяти, то здесь предпочтение отдается компилируемым языкам.

Окончательный выбор языка программирования для реализации проекта был сделан в пользу с++. Так как язык существует уже достаточно давно, вокруг него сформировалось плотное сообщество, множество форумов. Помимо этого, язык имеет множество различных библиотек, реализующих большое количество алгоритмов. Примером одной из самых крупных является - Boost - она предоставляет очень широкий функционал, а сама разрабатывается людьми, которые входят в комитет стандартизации языка с++. Так как проект не имеет legacy-кода, то в качестве стандарта был взят последний из доступных на момент написания работы - с++23.

**Вывод по главе**

В ходе написания данной главы были сформулированы требования к проекту, которые будут учитываться при его реализации. Также были рассмотрены существующие решения, выявлены их особенности реализации, в том числе архитектурные. Проанализированы их преимущества и недостатки. Помимо этого, был выбран язык программирования для реализации проекта с тщательным анализом его уместности использования в рамках данного проекта.

# Обзор инструментария

Для с++ написано множество инструментов и библиотек, позволяющих расширить базовый функционал языка. Они играют важную роль в процессе разработки программного обеспечения, предлагая разработчикам возможность создавать сложные приложения эффективно и удобно. Помимо этого, библиотеки сильно ускоряют процесс разработки, а главное делают код более безопасным с точки зрения утечек памяти и появления исключений, так как разработчики библиотек тратят немало времени на проектирование и тестирование их кода. В этой главе рассматриваются инструменты с их подробным описанием, которые будут использоваться при разработке проекта. Наиболее важными критериями при выборе качественных библиотек являлись: хорошая и понятная документация, наличие обзорных статей и положительных отзывов, активная поддержка библиотеки разработчиками и сообществом, а также удобство использования.

## Автоматизация сборки программного обеспечения

Для сборки проекта принято использовать специализированное ПО, которому при помощи скриптов разработчик проекта описывает правила и цели сборки. Программа перед сборкой проверяет наличие необходимого компилятора, а также библиотек-зависимостей, которые указывает разработчик проекта. Помимо этого, программа способна запускать тесты после этапа сборки для проверки корректности установленного проекта.

В качестве программы для автоматизированной сборки проекта был выбран CMake, так как является самой популярной программой среди подобных для с++.

CMake - кроссплатформенное программное обеспечение, позволяющее автоматизировать процесс сборки программ, написанных, как правило, на языке программирования С или С++. CMake не собирает программы, а лишь генерирует необходимые для сборки исходного кода файлы согласно инструкциям, описанным в файле сценария CMakeLists.txt [13]. Этот файл содержит правила и цели сборки. То есть CMake представляет собою front-end часть, которая используется при сборке исходных кодов и создает файлы, нужные для последующего этапа сборки, при этом можно сгенерировать файлы, которые обеспечат готовую структуру проекта для среды разработки/компилятора, где в дальнейшем будет производится развитие или сборка проекта (примерами могут быть: CodeBlocks, Eclipse, или MS VC++). CMake также может использоваться для сборки проекта, для чего он использует make или Ninja в качестве back-end [14].

Самый тривиальный файл CMake может содержать лишь 3 строчки [15]. Первой строчкой почти всегда указывается минимальная версия CMake, которая необходима для сборки проекта. Это необходимо, чтобы избежать проблем при сборке и вывод непонятных ошибок. Далее обычно следует строка, содержащая очень краткую информацию о проекте: его название, версию и описание. Далее в файле идут команды для CMake, указывающие ему исходные файлы, библиотеки, цели сборки и тому подобное.

Для того, чтобы получить из исходных файлов проекта файлы решения, необходимо создать папку build. Обычно она создается в корневой папке собираемого проекта. Далее необходимо перейти в эту папку и вызвать программу “cmake”, указав в качестве аргумента относительный или абсолютный путь до директории с файлом CMakeLists.txt. Этот файл обычно находится на один уровень выше, если папка build была создана в корневой папке проекта, и если это действительно так, то CMake можно вызвать из консоли следующей командой: “cmake ..”. При сборке проекта, особенно на чистой системе, случаются ситуации, когда CMake не может найти нужного пакета или библиотеки. Тогда, необходимо скачать исходники библиотеки, и сгенерировать файлы, необходимые для дальнейшей сборки. Затем, можно, либо собрать проект в среде разработки, либо собрать при помощи CMake, используя команду “cmake –build .”. Также можно указать опцию “-DCMAKE\_INSTALL\_PREFIX=usr\installPath”, которая определит путь, куда будут установлены файлы собранного проекта или библиотеки (указывать необходимо в момент конфигурации файлов для сборки, а не в момент сборки) [16].

Помимо самого процесса сборки, CMake позволяет автоматически запускать тесты для собранного кода, сгенерировать документацию, провести статический анализ, а также измерить покрытие кода [17].

## Сериализация данных

Json

Для проекта необходима библиотека, которая будет брать на себя работу с сериализацией и десериализацией данных из файлов с форматом json, так как именно этот тип файлов будет отвечать за хранение информации о конфигурации будущего конвейера. Был выбран именно этот формат для хранения информации о конфигурации по нескольким причинам:

1. формат удобочитаем для человека. Пользователями фреймворка будет меняться конфигурация конвейера, и важно, чтобы человеку было легко воспринимать информацию из файла конфигурации, а также модифицировать ее, при этом затрачивая оптимальное количество временных ресурсов;
2. занимает мало памяти. Так как формат состоит из текста, он хорошо сжимается, и при его отправке нагрузка на сеть будет крайне незначительной. Также он занимает мало оперативной памяти при парсинге файла непосредственно в программе-фреймворке.

Так как файлы с конфигурацией в данном проекте не будут хранить информацию, которая требует криптографической защиты и не будут хранить бинарные данные, то формат json идеально подходит здесь.

В результате была выбрана библиотека nlohmann::json [18], так как она удовлетворяет всем критериям [19] по выбору библиотек, которые были описаны выше. Помимо них, она является header-only, что упрощает ее импорт в проект. Также поддерживает STL-подобный синтаксис, что очень удобно, так как код из этой библиотеки совместим с функциями из стандартной библиотеки языка с++.

Cap’n proto

Данные, которые конвейер принимает на вход, могут поступать из разных источников. Ниже приведен перечень возможных способов:

* локально. Этот способ подразумевает получение данных, отправленных из другого процесса или сенсора, подключенного к рабочей станции, на которой запущен конвейер;
* удаленно. В данном случае данные поступают из сети. Источником данных в данном случае может быть удаленный сервер или база данных. Также допускается такой случай, когда конвейер делится на части и каждая из них выполняется на разных рабочих станциях. Тогда данные необходимо доставлять из одной части конвейера в другую.

Во всех вышеперечисленных случаях для эффективной передачи данных их нужно преобразовывать в форму, которая позволит безошибочно отправить данные из одного места в другое. При этом для минимизации задержек необходимо делать это быстро и безопасно.

В таблице 1 приведено сравнение популярных библиотек на основе бенчмарка [20]:

Таблица 1

Показатели различных библиотек для (де) сериализации данных.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Название протокола | Размер получившегося сообщения (байт) | Время сериализации (наносекунд) | Время десериализации (наносекунд) | Общее время (наносекунд) |
| FastBinaryEncoding | 234 | 66 | 82 | 148 |
| Cap’n proto | 208 | 558 | 359 | 917 |
| FlatBuffers | 280 | 830 | 290 | 1120 |
| ProtoBuf | 120 | 628 | 759 | 1387 |

Из наиболее быстрых библиотек оказались FBE (Fast Binary Encoding) [21] и Cap’n proto [22]. По данным таблицы, на сериализацию и дальнейшую десериализацию одинакового представления данных они занимали 148 и 917 наносекунд соответственно. Однако при дальнейшем поиске информации об использовании FBE была обнаружена только документация и при попытке установить библиотеку в качестве зависимости для проекта возникли сложности. Библиотека успешно собралась, но в момент сборки возникали ошибки времени редактирования связей. По причине отсутствия полной информации по сборке в документации и отсутствия сообщества вокруг библиотеки, проблему устранить не удалось. Поэтому выбор был сделан в сторону библиотеки Cap’n proto. Она в отличие от своего конкурента, имеет богатую документацию, довольно широкое сообщество пользователей, а также в сети можно найти статьи с примерами ее использования, что упрощает усвоение инструмента и дальнейшую реализацию проекта. На официальном сайте можно ознакомиться с протоколом, который реализован внутри библиотеки, а также с алгоритмами защиты от внешнего вмешательства при преобразовании данных, которые применяют авторы [23].

Cap’n proto хранит данные в памяти согласно тому же макету организации данных, что и у компилятора, также при организации данных используется выравнивание, согласно которому данные в памяти располагаются особым образом, что позволяет процессору быстрее их обрабатывать. Для описания вида и типов данных, участвующих в процессе сериализации используется файл описаний (“schema files”), который в дальнейшем при помощи поставляемой утилиты компилируется в файл исходного кода на нужном языке программирования и встраивается в проект. При описании формата данных используется синтаксис, похожий на С. Но отличие в том, что сначала указывается имя поля, а затем его тип, так как считается, что это удобнее при беглом анализе кода. Также используются аннотации, при помощи которых обеспечивается обратная совместимость при изменении схемы описания данных. При описании формата сериализуемых данных присутствует много знакомых с языка с++ вещей, например, структуры (struct), группы (в языке с++ — это пространства имен), объединения (union) и перечисления (enum). Помимо преобразования данных, библиотека предоставляет функционал для написания кода с использованием RPC (Remote Procedure Call), что может быть полезно при дальнейшем развитии и масштабировании проекта.

Преимущества:

1. хорошая документация;
2. широкое сообщество, поддерживающее проект;
3. аннотации, при помощи которых обеспечивается обратная совместимость при изменении структуры данных;
4. наличие алгоритмов, отвечающих за безопасность данных;
5. функционал для написания кода, использующего RPC.

Недостатки:

1. не является самой быстрой библиотекой для преобразования данных;
2. из–за встроенных в библиотеку алгоритмов выравнивания данных приходится делать лишние копирования, которые могут негативно повлиять на производительность программы.

## Представление данных внутри конвейера

Для представления данных внутри конвейера были выбраны 2 структуры данных, первая предоставляет функционал для работы с тензорами, а вторая позволяет хранить и обрабатывать цифровые изображения и видео.

Изначально при поиске библиотек для данной задачи, было найдено три, которые на первый взгляд удовлетворяли условиям отбора. После чего библиотеки были установлены, и в результате был написан бенчмарк, в рамках которого, измерялось время, необходимое на помещение полученных из сети данных в структуру данных, которую предоставляет тестируемая библиотека. Также отслеживалось и потребление памяти. Библиотеки OpenCV и xTensor обладают богатой документацией и множеством примеров, которые можно найти как в репозитории авторов, так и в сети Интернет. Они также оказались и удобными в использовании, в добавок библиотека xTensor реализовывает возможности, появившиеся в последних стандартах с++. А OpenCV внутри себя использует множество идиом, одна из которых copy-on-write, аналог семантики перемещения, появившийся задолго до ее появления. Библиотека boost::ublas::numeric::tensor же, в отличие от перечисленных выше, написана под старый стандарт языка, и для нее не удалось найти статей и примеров использования в сети Интернет. Также при тестировании в бенчмарке, она показала плохой результат, который объясняется тем, что в библиотеке нет подходящих способов инициализации создаваемого объекта. Для эффективного создания объекта, с минимум копирований памяти, объекту обычно передается умный указатель, как это реализовано в xTensor. Либо передается указатель на общие данные, как в OpenCV. В последнем случае копирование данных происходит тогда, когда происходит изменение данных, при условии, что у них есть несколько владельцев. Библиотека boost::ublas::numeric::tensor требует копирования сразу же при создании объекта, что не является подходящим и эффективным вариантом. Это демонстрирует результат бенчмарка на рисунке 4.

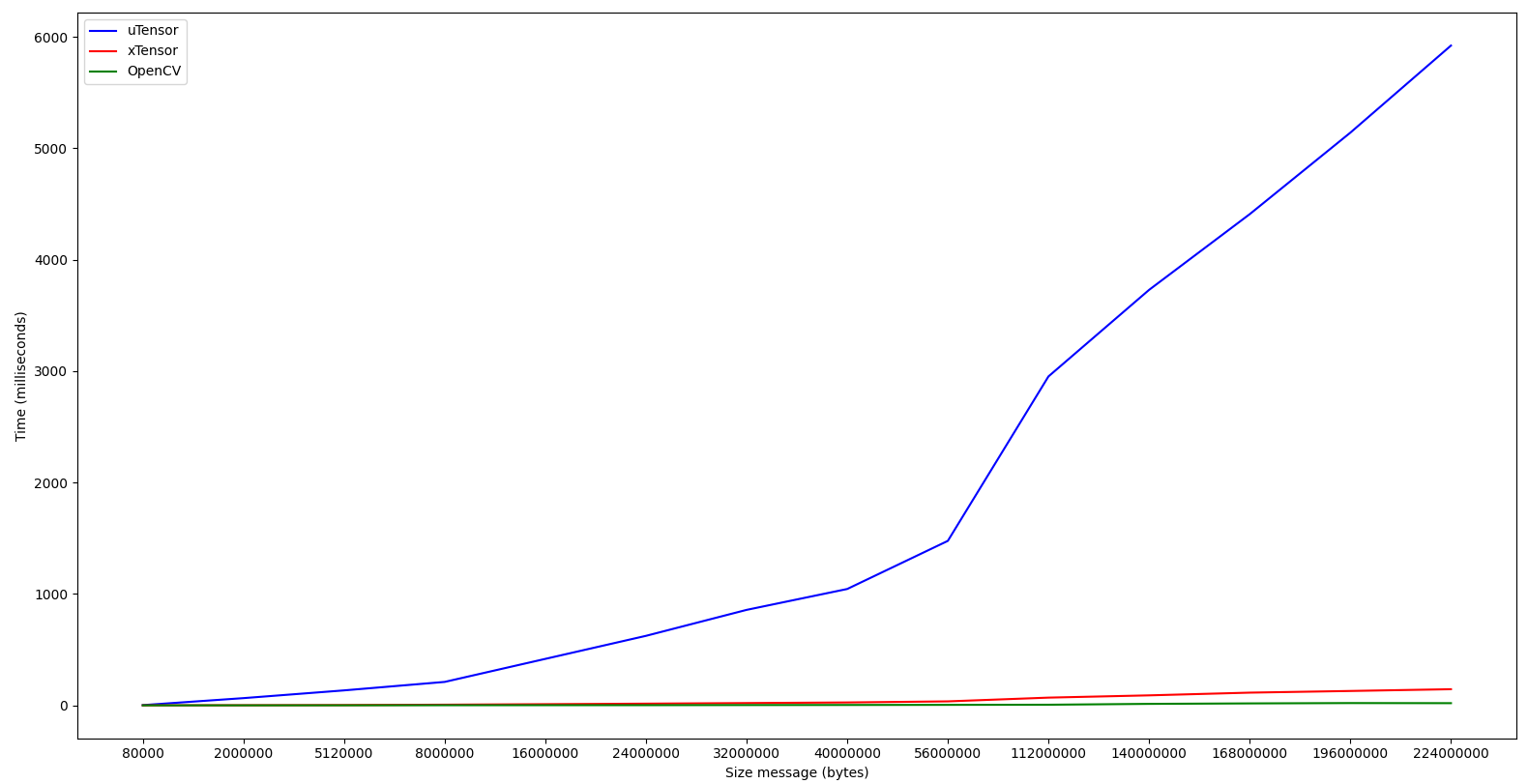


Рисунок 4 – Бенчмарк рассматриваемых библиотек в режиме Debug

На графике видно, что с ростом объема данных, которыми инициализируется объект, время, затрачиваемое на его создание в случае с библиотекой boost::ublas::numeric::tensor растет на порядки. Поэтому было решено использовать библиотеку xTensor вместо boost::ublas::numeric::tensor.

Xtensor

xtensor [23] - библиотека, предоставляющая функционал для представления в памяти многомерных данных, а также методы для их обработки. При разработке этой библиотеки, авторы пытались сделать ее максимально похожей на библиотеку numpy, а это означает, что при работе с этой библиотекой отсутствует перегруженный синтаксис.

Изначально, для проекта предполагалось взять библиотеку boost::numeric::ublas::tensor [24]. Но при подключении и использовании ее в проекте возникало много сложностей и проблем. Во-первых, документация к этой библиотеке написана плохо, в ней мало примеров, а также технической информации о внутреннем устройстве контейнера. Во-вторых, при инициализации контейнера уходило слишком много времени на копирование данных внутрь структуры тензора. При этом, не было найдено удобных способов инициализации тензора, например, при помощи итераторов, удовлетворяющих тегу forward\_iterator\_tag, или при помощи умных указателей, которые позволяют вовсе избавиться от копирования данных внутри тензора. В-третьих, библиотека написана под старый стандарт с++, и, не исключено, что при ее использовании могут возникнуть проблемы совместимости между кодом проекта и кодом используемой библиотеки. Библиотека xtensor лишена этих всех недостатков, поэтому была выбрана вместо библиотеки тензора из буста.

OpenCV

OpenCV - довольно популярная библиотека, предоставляющая функционал для обработки цифровых изображений и видео. Сюда входит огромное множество алгоритмов компьютерного зрения, в том числе, для видеоанализа. Библиотека способна работать с большинством форматов, которые используются для хранения изображений и видео. Для удобной работы, в библиотеке присутствует вспомогательный модуль, позволяющий создавать простой и удобный пользовательский интерфейс [25].

Из библиотеки OpenCV для настоящего проекта будут использованы алгоритмы, которые предоставляет библиотека, а также структура данных cv::Mat, предназначенная для хранения цифрового изображения или матриц. Преимуществом структуры cv::Mat является встроенный счетчик ссылок, который уменьшает количество фактических копирований объекта (идиома copy-on-write).

Еще одним преимуществом библиотеки является использование типов прокси-классов (например, cv::InputArray) для аргументов функций. Они позволяют вместо конкретного типа для аргумента функции использовать любой, который способен преобразоваться в прокси-класс. Таким образом, функции openCV принимают на вход не только тип cv::Mat, но и, например, тип std::vector<> из стандартной библиотеки с++, или большинство массиво-подобных объектов.

Финальный результат тестирования отобранных библиотек приведен на рисунке 5, и вполне удовлетворяет условиям проекта.

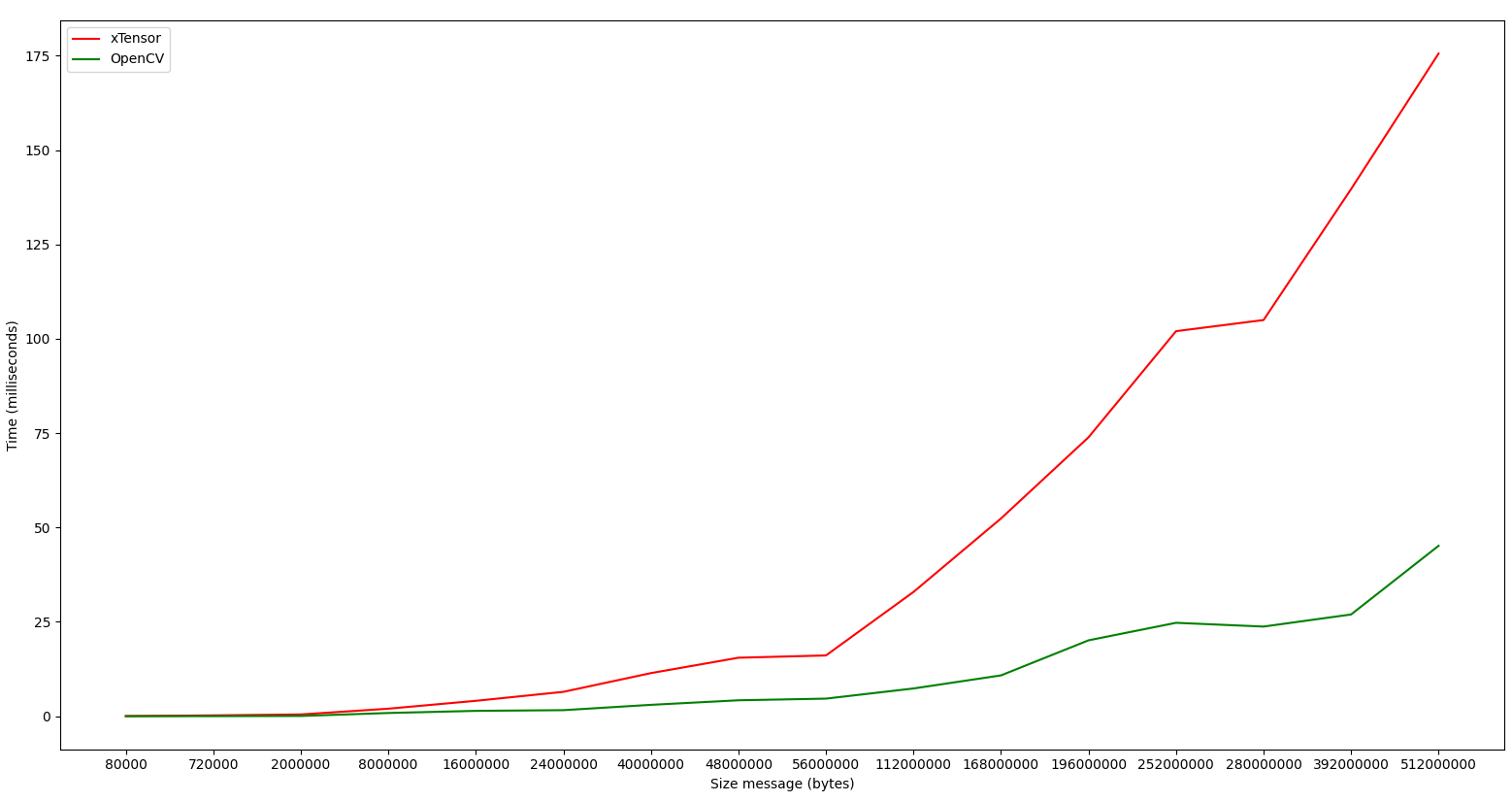


Рисунок 5 – Финальное тестирование отобранных библиотек в режиме Release

## Брокер сообщений

CppZMQ

При написании кода, отвечающего за сетевое взаимодействие и передачу данных по сети используются сокеты. Из-за разнообразия вычислительных машин, код при работе с низкоуровневыми сокетами не будет кроссплатформенным, то есть потребуется его разные вариации, каждая для конкретной архитектуры ЭВМ. Одна из причин этому, порядок байт, который может быть прямым (big endian) или обратным (little endian), и его следует учитывать при отправке сообщения с одной машины на другую. Поэтому для написания кода, отвечающего за отправку и прием сообщений по сети будет использована библиотека cppzmq [26]. Библиотек для данных задач много, и они не имеют особого отличия в производительности. Однако, оборудование (цифровые камеры) на предприятии заказчика, использует библиотеку zmq для передачи снимаемых данных. Поэтому была выбрана вариация библиотеки zmq - cppzmq, которая являлась наиболее популярной и документированной. Cppzmq является c++ - оберткой (wrapper) над библиотекой zmq, написанной на языке программирования C. При реализации cppzmq были применены идиомы, присущие языку c++. Одна из таких - идиома RAII (Resource Acquisition Is Initialization), смысл которой заключается в том, что класс должен захватывать ресурс при создании объекта - в конструкторе и освобождать при уничтожении объекта класса - в деструкторе. Также класс должен сохранять инварианты класса на протяжении всего времени жизни (lifetime) объекта.

CppZmq - библиотека, предоставляющая разработчику инструменты для создания собственной системы очереди сообщений, что позволит создать распределенное и параллельное приложение. Библиотека представляет собою программный интерфейс, позволяющий создать свою mq-систему с компромиссом между сложностью реализации и высокой производительностью. Cppzmq имеет огромное количество пользователей, а также гайдов по вариации использования и статей [27].

## Соединение стадий конвейера между собой

При построении конвейера в фреймворке, будет предоставляться возможность создания как пакетного типа, так и параллельного типа конвейера. От типа конвейера зависит вариация структуры данных, которая будет использоваться при реализации коннектора, отвечающего за доставку данных от одной стадии к другой. При параллельной организации конвейера, коннектор должен поддерживать многопоточную запись и многопоточное чтение. В качестве коннектора будет использоваться такая структура данных, как очередь, так как она построена по принципу FIFO (first in - first out), что позволит соблюдать порядок при передаче данных.

В случае пакетной обработки данных будет использоваться spsc queue (single producer single consumer). Это потокобезопасная очередь, которая предназначена для условий использования, когда с очередью работают только два потока - один писатель (producer) и один читатель (consumer). В случае, если конвейер построен параллельно, то будет использоваться mpmc queue (multi producer multi consumer). Mpmc очередь - потокобезопасная очередь, в которую может производить, и запись, и чтение одновременно более одного потока. Для сравнения очередей, был написан бенчмарк, в котором отслеживались скорость работы очереди и объем потребляемой памяти. Результат бенчмарка приведен в таблице 2.

Таблица 2

Тестирование различных видов lockfree очередей.

|  |  |  |  |
| --- | --- | --- | --- |
| Тип очереди | Время в конфигурации Debug (секунд) | Время в конфигурации Release (секунд) | Количество потребляемой памяти (мегабайт) |
| SPSC queue | 8,813 | 0,535 | 565 |
| MPSC queue | 9,961 | 2,112 | 1120 |
| MPMC queue | 10,473 | 2,125 | 1120 |

При тестировании очередям задавался фиксированный размер, чтобы избежать обращений к ядру, что довольно долго. Размер всех очередей был одинаков и позволял хранить 512 мегабайт. Далее создавалось по одному потоку, один из которых будет записывать данные в очередь, а другой считывать. Измерялось время, с момента начала записи потоком писателем до момента считывания последнего элемента потоком читателем. Согласно результатам, можно сделать несколько следующих выводов:

* + SPSC очередь лучше оптимизируется, в отличие от mpmc очереди;
  + MPSC и MPMC очереди потребляют в два раза больше памяти. Это связано с тем, что к элементу может обращаться более одного читающего потока или более одного записывающего потока. Поэтому необходимо на каждый элемент, помимо самих данных элемента, хранить структуру, предоставляющую механизм синхронизации, обеспечивающий потокобезопасное обращение к данным элемента;
  + между MPSC и MPMC очередями разница во времени работы незначительна, а количество потребляемой памяти одинаково. Поэтому в проекте будет использована более функциональная из них, то есть MPMC.

Итого, для пакетной организации конвейера будет использоваться spsc очередь, потому что этот тип очереди работает быстрее версии mpmc, а также лучше поддается оптимизациям. Для параллельного типа конвейера будет использоваться реализация mpmc.

Реализация spsc queue

В качестве spsc очереди была взята реализация из сборника библиотек boost (boost::lockfree::spsc\_queue) [28]. Эта реализация является одной из самых популярных и документированных, что доступны на момент разработки проекта.

Реализация mpmc queue

Для mpmc очереди было найдено две реализации: boost::lockfree:queue [29] и cds::container::VyukovMPMCCycleQueue [30]. Для использования в проекте была выбрана вторая, так как первая библиотека устанавливает строгие требования на тип, который хранится в очереди. Согласно одному из требований к типу, очередь должна содержать тип, который имеет тривиальный деструктор. Но так как очередь будет использоваться для хранения сложных типов, таких как std::shared\_ptr и cv::Mat, которые имеют нетривиальный деструктор (ввиду того, что они хранят в себе ресурс), то использование реализации очереди из библиотеки boost не подходит для текущего проекта.

Библиотека cds является библиотекой для языка c++, которая узко специализируется на lockfree алгоритмах. Она реализовывает множество lockfree структур данных, а также множество алгоритмов, предназначенных для использования в многопоточных системах [31].

## Оптимизация процесса управления потоками

Boost.Thread Management

Для создания пула потоков-воркеров, который хранит в себе объекты-потоки, а также предоставляет интерфейс для управления ими будут использоваться реализации boost::thread и boost::thread\_group. На данный момент в стандарте языка c++ есть реализация std::thread, но boost::thread предоставляет удобный интерфейс для остановки потока за его пределами (со стороны вызывающего кода), при помощи метода interrupt [32] у объекта потока.

## Обработка и манипуляция большим количеством многомерных данных

hdf5 [33] - библиотека, позволяющая работать с файлами формата hdf5. HDF (Hierarchical Data Format) - формат файлов, разработанный специально для хранения большого объема цифровой информации. Из-за того, что формат позволяет очень быстро обрабатывать разнородные данные любого размера и количества, он используется в научных вычислениях на суперкомпьютерах, где вычисления происходят параллельно и генерируется очень большое количество информации (порядка терабайта в минуту). Изначальный спрос на такой формат данных был порожден Министерством энергетики США. А после им активно начали пользоваться и в других сферах, например в производстве, аэрокосмических вычислениях и сфере финансов. Библиотека hdf5 имеет реализацию на многих языках программирования и предоставляет ООП-интерфейс для работы с hdf-файлами.

## Отслеживание утечек памяти в проекте

При написании программ на низкоуровневых языках программирования, таких, как C++ или C (си), где программист управляет памятью вручную, очень важно следить за тем, чтобы вся выделенная память была освобождена. Для предотвращения утечек памяти была придумана концепция умных указателей (smart pointers), которые захватывают указатель на сырую память, и освобождают при разрушении объекта (идиома RAII). Такие указатели активно используются в проекте. Они особенно полезны, когда в программе используется механизм исключений. При возникновении исключения, происходит раскручивание стека (stack unwinding), во время которого вызываются деструкторы объектов, и в этот момент происходит освобождение памяти, которой владеют умные указатели.

Для отслеживания аллокаций и деаллокаций памяти в проекте, а также профилактики утечек памяти будет использоваться утилита valgrind [34]. Это самая распространенная утилита, завоевавшая доверие большинства разработчиков. На данный момент valgrind уже перерос в фреймворк, выполняющий динамическую компиляцию (или трансляцию), то есть он преобразовывает код в промежуточное представление, а затем передает его специальному инструменту [35]. Инструмент представляет собой подпрограмму, которая отслеживает нужные метрики, что позволит в дальнейшем произвести анализ памяти, используемой проектом. Самым популярным инструментом является MemCheck. Он позволяет выявить попытки использования неинициализированной памяти или памяти, которая не принадлежит адресному пространству программы проекта.

**Вывод по главе**

В ходе написания данной главы были подобраны инструменты, в том числе для автоматизации процесса сборки исходного кода, а также для отслеживания утечек памяти. Для выявления более подходящих и эффективных инструментов были написаны бенчмарки, либо проанализированы уже существующие.

# Разработка примитивных модулей для будущего фреймворка

Перед разработкой фреймворка необходимо реализовать отдельные его модули на более высоком языке программирования, чтобы убедиться в том, что при реализации этих модулей на более низком языке не возникнет проблем и несостыковок.

## Генератор данных

Для работы конвейера необходим поток данных. В реальных условиях программы, данные поступают с сенсоров. Но на этапе разработки будет использоваться генератор данных, написанный на языке программирования Python. Использование генератора упрощает процесс отладки, так как позволяет очень быстро настроить частоту кадров, их размерность и разрешение. Генератор получает эти параметры при помощи аргументов командной строки, при этом он имеет значения по умолчанию

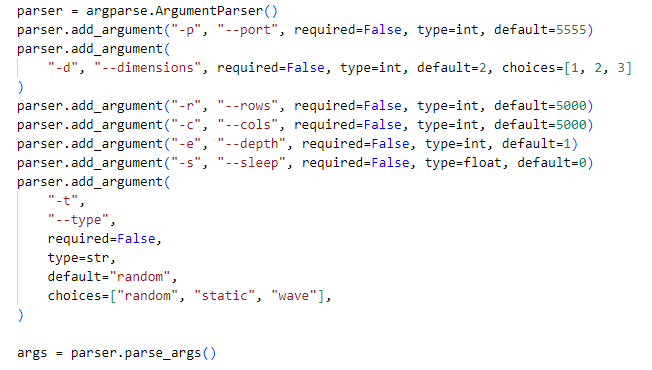


Рисунок 6 – Получение аргументов командной строки в скрипте генератора данных

Генератор используется для эмуляции функций сенсора (цифровых камер, которые используются на предприятии заказчика), поэтому он сериализует данные при помощи библиотеки Cap’n Proto, а для их отправки использует брокер сообщений ZMQ – тот же самый стек используется и на реальном сенсоре, который установлен на предприятии заказчика. Данная реализация генератора генерирует массив требуемой размерности, в котором типом элементов является float64.

## Примитивная реализация однопоточного конвейера

Перед написанием проекта на с++ было решено написать создание конвейера на языке программирования python, так как это не займет много времени и позволит убедиться на практике в верности видения будущей структуры конвейера.

Для представления конвейера в коде, был создан класс с названием “Node”, который представляет собою одну стадию конвейера. Пример структуры конвейера приведен на рисунке 7. Каждый прямоугольник представляет собою отдельный экземпляр класса “Node”.

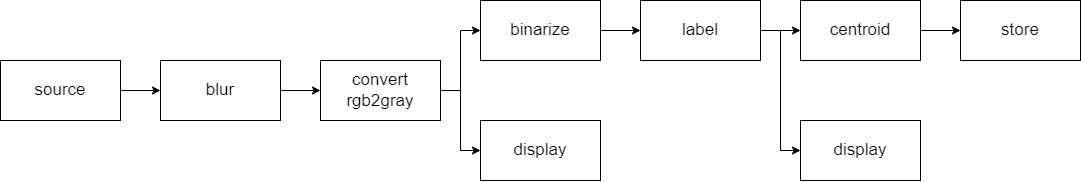


Рисунок 7 – Пример структуры конвейера.

Каждый экземпляр стадии хранит в себе его имя (поле “name”), объект функции (поле “callable”), а также список детей (поле “childs”). Поле с именем стадии потребуется, когда конвейер будет собираться при помощи конфигурации. Поле с функцией представляет собою действие, которое стадия делает над данными. А список детей необходим для того, чтобы после обработки данных, передать их дальше по конвейеру, то есть потомкам текущей стадии. Часть реализации класса “Node” приведена на рисунке 8.

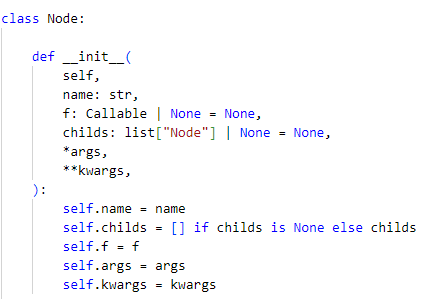


Рисунок 8 – Конструктор класса “Node”

Для того, чтобы собрать последовательность конвейера, для начала необходимо создать несколько экземпляров класса “Node” и передать им необходимые параметры. На данном этапе имя стадии не влияет на семантику работы программы. На рисунке 9 приведен код для создания конвейера со структурой, схожей с той, что продемонстрирована на рисунке 7.



Рисунок 9 – Создание конвейера

Для его запуска был определен метод “\_\_call\_\_”, в котором принимаемый в качестве аргумента массив передается функции, которую хранит объект стадии, а затем результат, возвращаемый данной функцией, передается дочерним стадиям. Когда такая цепочка вызовов дошла до последней стадии, конвейер завершает свою работу.

## Набросок конфигурационного файла

Построение конвейера напрямую в коде неудобно, а также требует перекомпиляции программы. Одно из требований проекта – возможность изменения цепочки вызовов без перекомпиляции фреймворка. Эту возможность можно реализовать, если использовать конфигурационный файл, в котором будет содержаться информация о структуре конвейера, который необходимо построить. Фреймворк сможет во время выполнения считывать данные из конфигурационного файла, а затем построить конвейер и запустить его. Пример получившегося json файла с конфигурацией продемонстрирован на рисунке 10.



Рисунок 10 – Пример простого файла конфигурации

Конфигурационный файл содержит в себе два объекта: “stages” и ”connections”. Объект “stages” содержит в список “list”, который в свою очередь содержит объекты, представляющие собой макет стадии. Эти объекты содержат информацию, которая необходима для создания стадии, то есть ее имя и идентификатор функтора. Значимость имени для стадии будет раскрыта далее. Теперь, когда возможно создать объекты стадий, которые содержатся в конвейере, необходимо собрать из них необходимую последовательность для организации необходимой цепочки вызовов. Для этой задачи нужен второй объект, находящиеся в конфигурации – “list”. Он представляет собою список, в котором находятся объекты, содержащие только два свойства: “from” и ”to”. Эти свойства содержат имена стадий, и служат для установления счвязей между стадиями. Следует обратить внимание, что свойство “to” может являться как строкой, так и массивом. Для установления стартовой (первой) стадии, используется свойство “start” в объекте “stages” конфигурации.

# Выводы по главе

В этой главе был описан процесс реализации примитивных модулей, которые позволят упростить разработку фреймворка на целевом языке, и сократить количество ошибок. Также их реализация упрощает дальнейшее проектирование сущностей будущего программного кода.

# Разработка фреймворка для построения конвейера обработки данных

## Разработка примитивного конвейера на с++

При разработке примитивного конвейера на языке с++ основная задача состояла в том, чтобы перенести работающий код с языка python. Результат продемонстрирован на рисунке 11.

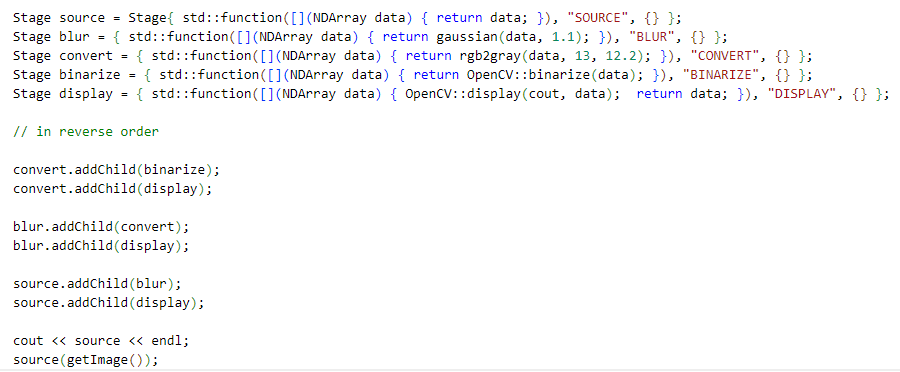


Рисунок 11 – Создание примитивного конвейера в коде с++

После того, как реализован конвейер, необходимо написать парсер, который будет принимать на вход файл с конфигурацией, а возвращать собранный на ее основе конвейер. Для своей работы, парсер одним из аргументов принимает функтор, который возвращает callable объект согласно переданному ему идентификатору. Этот функтор передается парсеру из пользовательского кода и пишется пользователем фреймворка. Он позволяет соотнести идентификатор стадии конвейера и предназначенную для нее функцию. Алгоритм парсера следующий:

* + - 1. парсер считывает с файла конфигурации объекты “stages” и “connections”, а также идентификатор первой стадии в последовательности конвейера;
      2. создается первая стадия;
      3. создается очередь, которая будет хранить в себе ссылки на созданные далее стадии. В эту очередь добавляется стартовая стадия;
      4. в цикле, пока очередь не пуста, изымается стадия;
      5. для изъятой стадии по ее идентификатору находится список идентификаторов ее дочерних стадий. Далее при помощи этого списка создаются дочерние стадии и складываются в очередь;
      6. по завершении цикла, парсер возвращает собранный конвейер.

Ключевым моментом, который необходимо учитывать при реализации парсера является то, что структура данных, хранящая в себе стадию конвейера, использует копирующую семантику. То есть, при добавлении стадии ребенка (дочерней стадии), дочерняя стадия будет скопирована, и ее последующее изменение в коде парсера не приведет к изменению копии, которую мы скопировали нашей стадии. Поэтому, очередь хранит в себе ссылку на созданную стадию.

При написании парсера были использованы алгоритмы поиска стандартной библиотеки, которые поддерживаются библиотекой для работы с json файлами.

## Осуществление получения данных с генератора

После реализации примитивного конвейера и парсера для его сборки предстояла задача осуществить передачу потока данных с генератора в конвейер. Для этого была установлена библиотека cppZmq и написан код для приема сообщений, продемонстрированный на рисунке 12.

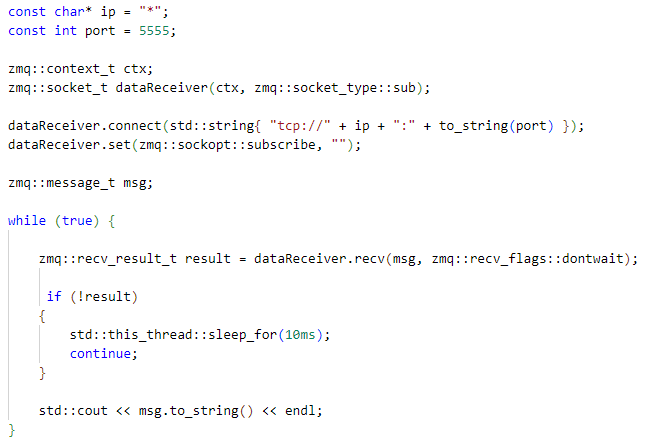


Рисунок 12 – Код для приема сообщений при помощи cppZmq

Для приема сообщений, происходит подключение к генератору посредством библиотеки cppZmq, указываем свою роль – subscriber (подписчик), а затем в цикле делаем запрос к коду cppZmq, запрашивая данные из сети. Если данные не пришли, то устанавливаем небольшую задержку, в противном случае выводим последовательность байт на экран.

Так как по сети передаются сериализованные (запакованные) данные, то далее предстояла задача их десериализации. Для этого была установлена библиотека Cap’n proto и написан код для десериализации сообщения в структуру данных cv::Mat и xt::xarray. Обе структуры работают с многомерным типом данных, однако первая из них содержит методы и алгоритмы для работы с цифровыми изображениями и видео, а вторая структура содержит методы для работы с тензорами. Выбор структуры, в которую будут десериализовываться данные, предоставляется пользователю в момент компиляции фреймворка. Часть кода, которая отвечает за десериализацию данных в структуру opencv::Mat продемонстрирована на рисунке 13.



Рисунок 13 – Код, десериализующий данные в структуру cv::Mat

## Построение многопоточного конвейера

После того, как у фреймворка была реализована возможность принимать поток данных с генератора, необходимо было модифицировать конвейер. Для эффективной обработки данных конвейером, необходимо каждой стадии работать независимо от других, то есть в своем потоке. Для этого необходимо создать пул потоков, в котором будут хранится потоки, выполняющие функции стадий. Пул потоков также позволит в будущем реализовать возможность расширяемости конвейера (scaling), которая в свою очередь позволит регулировать количество потоков, которые выделены на каждую стадию конвейера.

Конвейер, в котором стадии выполняются параллельно и не зависят друг от друга, называется параллельным. Для такой организации конвейера необходимо предусмотреть коннекторы для стадий. Так как необходимо соблюдать порядок обработки данных между стадиями, то в качестве коннекторов будет использоваться структура данных очередь. А так как в дальнейшем планируется реализация конвейера, в которой на одну стадию может приходится более одного потока, то необходима реализация очереди, которая будет поддерживать потокобезопасную запись и чтение, при которых производить модификацию очереди могут два потока и более.

Для создания потоков и запуска в них обработки функций необходимы вспомогательные структуры данных, которые будут хранить структуры данных, необходимые для запуска стадий в потоке. На рисунке 14 приведен код, в котором создаются такие структуры данных:

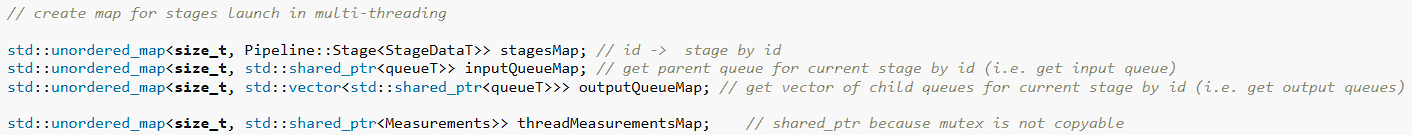


Рисунок 14 – Структуры данных, содержащие необходимые данные для запуска стадий в параллельном режиме

Данные структуры являются ассоциативными контейнерами, где ключом является id стадии конвейера, а значением нужная структура. Структура “stagesMap” хранит в себе копию структуры данных “Stage”, в которой находится идентификатор и callable объект для стадии. Структура “inputQueueMap” хранит очередь, в которую будут поступать входные данные для стадии. Структура “outputQueueMap” хранит вектор очередей, в которые стадия записывает свой результат работы, то есть возвращающее значение ее функтора. Эти же очереди являются входными очередями для дочерних стадий по отношению к текущей. Структура “threadMeasurementsMap” хранит объект, в который поток, на котором запущена стадия, будет записывать значения, предназначенные для мониторинга стабильности и скорости работы конвейера. Заполнение всех этих структур происходит в цикле.

После того, как эти структуры сформированы, происходит создание потока, которому выдаются необходимые структуры данных. Все созданные потоки хранятся в пуле потоков, который предоставляет удобный интерфейс для управления ими. Отрывок части кода цикла, в котором происходит создание потока приведен на рисунке 15.

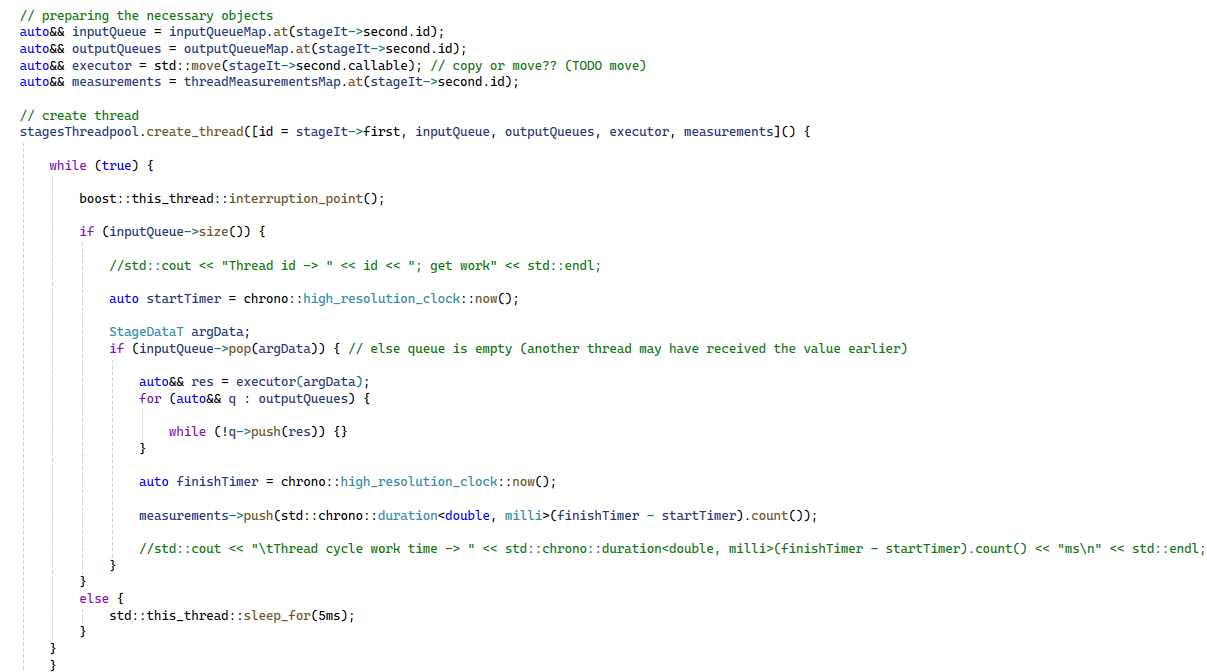


Рисунок 15 – Создание потока для стадии конвейера

После того, как потоки были созданы, производилось несколько оптимизаций для избегания лишних копирований.

## Мониторинг загруженности конвейера

Для того, чтобы иметь возможность расширять конвейер в дальнейшем и отслеживать узкие места при его работе, необходимо производить процесс мониторинга работы конвейера. В качестве отслеживаемых параметров были выбраны:

* количество элементов во входных очередях стадий;
* среднее время, затрачиваемое на обработку одного пакета данных.

Для отслеживания этих показателей, необходимо, чтобы потоки-воркеры, которые обрабатывают функции стадий, снимали текущие показатели и записывали в структуру данных, которую будет считывать в дальнейшем поток мониторинга. После считывания, этот поток будет формировать json объект и отправлять на заданный хост при помощи технологий zmq или opc ua.

Для хранения показателей была реализована структура данных “ Measurements”. “Measurements” содержит два метода, позволяющих записать показание или считать. На рисунке 16 продемонстрированы поля структуры, а также метод push.

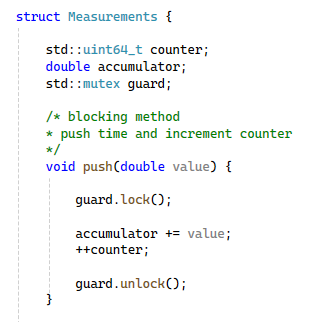


Рисунок 16 – Часть реализации структуры “Measurements”

Так как к одним и тем же данным будут обращаться сразу несколько потоков, то при реализации данной структуры данных были использованы механизмы синхронизации. В языке с++ для этого предназначен объект mutex (mutual exclusion). Защита данных происходит непосредственно при обращении к ним, то есть в методах структуры. Структура, помимо поля типа mutex, также имеет поля типа double и std::uint64\_t. Последние два предназначены для подсчета показателя среднего времени. Метод push является блокирующим, и при его вызове он инкрементирует счетчик, а также прибавляет переданное ему значение аргумента к аккумулятору.

Метод pull также является блокирующим. В нем происходит блокировка только на момент снятия данных и последующего сброса их значений. Возвращает среднее значение аккумулятора, используя для этого счетчик, который подсчитывает количество внесенных записей.

Код метода pull продемонстрирован на рисунке 17.

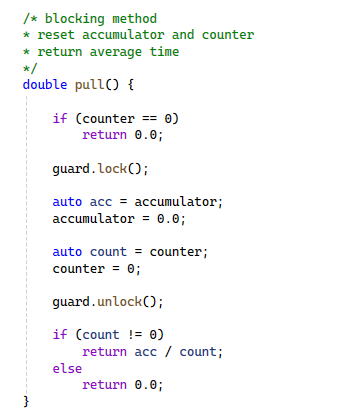


Рисунок 17 – Код метода pull структуры “Measurements”

После того, как готова структура для передачи показаний от потоков воркеров к потоку мониторинга, необходимо создать последнего. Поток будет использовать шаблон-заготовку json объекта. Результирующий json объект будет содержать в себе три объекта:

* в первом объекте “IdToCallable” содержится информация, при помощи которой можно получить идентификатор стадии по ее id, который был присвоен ей фреймворком;
* во втором объекте “QueueLoad” содержится информация о загруженности входных очередей стадий. По id стадии хранится целочисленный тип, представляющий собой размер очереди;
* в третьем объекте “ThreadTimePerOp” содержится информация о среднем значении времени, которое затрачивается на выполнение функтора стадии для пакета принимаемых данных. По id стадии хранится значение типа double;

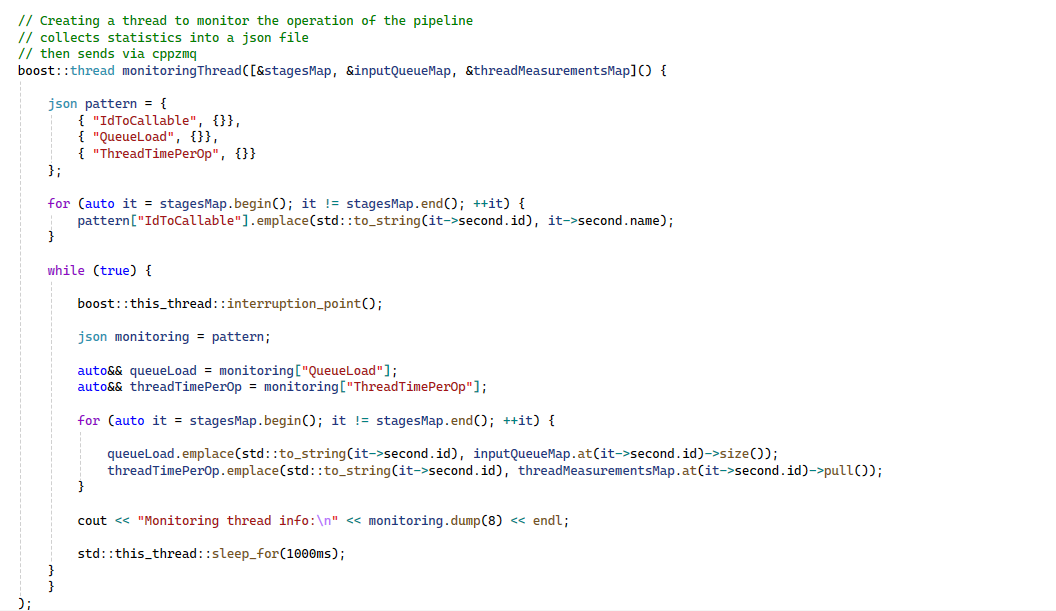


Рисунок 18 – Код создания потока для мониторинга

На данный момент поток мониторинга формирует json файл и выводит его содержимое на экран консоли.

## Написание CMake скрипта

Для автоматизации процесса сборки проекта необходимо написать файл CMakeLists.txt.

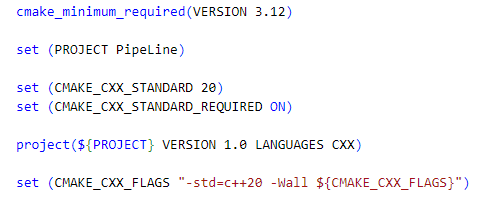


Рисунок 19 – Начало файла CMakeLists.txt

В самом начале файл должен содержать минимальную версию CMake, которая использовалась при разработке проекта. Далее следует название разрабатываемого проекта, стандарт используемого языка, а также декларация целей сборки. Одна из возможностей CMake – искать нужные для проекта зависимости на машине, на которой осуществляется сборка проекта.

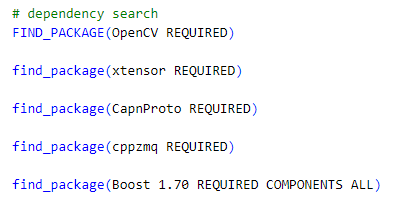


Рисунок 20 – Синтаксис для объявления нужных зависимостей проекта в файле CMakeLists.txt

Для поиска зависимостей используется команда find\_package, которой передается название зависимости, и опциональный флаг, наличие которого будет вызывать ошибку сборки, в случае если библиотека не найдена. Для целей сборки проекта при помощи команд target\_include\_directories и target\_link\_libraries указывается, какие директории и файлы каких библиотек должны быть подключены, то есть указаны компилятору в момент компиляции цели. Синтаксис, используемый при объявлении целей и подключении к ним библиотек продемонстрирован на рисунке 21.

Для сборки проекта создается папка в корневой директории с названием build. Затем, используя консольное приложение необходимо перейти в директорию build и вызвать команду CMake, передав ей в качестве аргумента директорию уровнем выше, то есть в которой содержится файл CMakeLists.txt. Также CMake позволяет генерировать файлы сборки для разных генераторов, в проекте использовались генераторы Ninja и Visual studio 2022. Задать конкретный генератор, например Ninja можно опцией -GNinja.



Рисунок 21 – Декларация целей сборки в файле CMakeLists.txt

## Архитектура сущностей фреймворка

Одной из ключевых целей архитектуры, при написании данного фреймворка является уменьшение связанности его компонент. Для достижения этого эффекта в проекте использовались классы-интерфейсы или механизм концептов [36], который появился в языке с++, начиная со стандарта с++20.

Классы-интерфейсы применялись в проекте в случаях, когда возвращаемые значения методов классов наследников совпадали и не зависели от реализации класса. Механизм концептов применялся в случаях, когда возвращаемые значения методов классов, которые предназначены для одной задачи, но имеют при этом разную реализацию, различались. Не исключены случаи, когда оба этих способа применялись вместе, например при реализации класса “ZmqReceiveManager”. Экземпляр данного класса будет передаваться в высокоуровневые политики. И для избегания зависимости высокоуровневых политик от низкоуровневых используются классы-интерфейсы. Таким образом, высокоуровневая политика будет принимать объект, типом которого будет выступать тип класса-интерфейса. Такой прием называется инверсией зависимостей (dependency-inversion) и позволяет в любой момент подменить реализацию низкоуровневой политики, при этом не меняя высокоуровневую политику, а это значит, что основная логика приложения затронута не будет. Код класса-интерфейса “INetworkReceiveManager”, от которого наследуется “ZmqReceiveManager” приведен на рисунке 22.

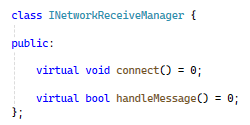


Рисунок 22 – Класс -интерфейс “INetworkReceiveManager”

В свою очередь, класс “ZmqReceiveManager” принимает аргументом объект, который отвечает за десериализацию сообщения, полученного при помощи сетевого брокера, в данном случае cppZmq. Тип данных, который возвращает объект, десериализующий сообщение, может отличаться, так как в конвейере могут использоваться разные типы, например cv::Mat или xt::xarray. В данном случае классы-интерфейсы не подходят и на их замену приходит механизм концептов. Он позволяет проверить характеристики типа, которым обобщается класс. Пример использования механизма концептов приведен на рисунке 23.

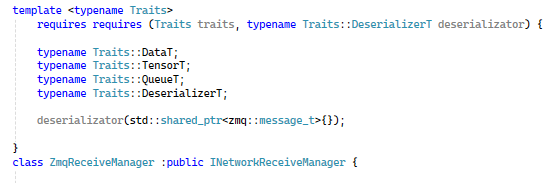


Рисунок 23 – Использование механизма концептов при определении шаблонного класса “ZmqReceiveManager”

**Вывод по главе**

В ходе написания текста данной главы был реализован основной функционал фреймворка, который позволит в дальнейшем расширить возможности уже существующего решения. Был написан код для построения конвейера параллельного типа. Был также реализован скрипт CMake для автоматизации процесса сборки исходного кода.

# Заключение

В ходе выполнения данной работы были рассмотрены аналогичные проекты, которые позволяют задать порядок вычислений в виде графа или в виде простой последовательности вызовов. Были выявлены особенности реализации, преимущества и недостатки. Также был подобран язык программирования для реализации проекта. Сформирован набор необходимого инструментария, который позволит написать программный код более эффективно. Помимо библиотек были подобраны инструменты для проверки кода на наличие ошибок, например проверка утечек памяти во время выполнения программы.

Был реализован основной функционал фреймворка. Написан код для построения пакетного и параллельного типа конвейера. Реализованы коннекторы для стадий, которые необходимы при параллельном типе построения конвейера. Помимо этого, написан поток, который производит мониторинг работы конвейера. Это позволит предотвратить появление ошибок во время работы конвейера или реализовать в дальнейшем возможность масштабируемости (scaling) конвейера. Для автоматической сборки проекта из исходного кода был написан скрипт для CMake.

# Список использованных источников

* 1. Что такое конвейер данных? И почему вы должны это знать [Электронный ресурс], URL: https://habr.com/ru/sandbox/188820/ (дата обращения: 20.02.2024)
  2. Apache Airflow: docker, python, DAG, конвейер обработки данных [Электронный ресурс], URL: https://ivan-shamaev.ru/apache-airflow-docker-python-dag-data-pipeline/#i (дата обращения: 22.02.2024)
  3. How Flink Works: Architecture, Use Cases, and Benefits [Электронный ресурс], URL: https://www.confluent.io/learn/apache-flink/ (дата обращения: 24.02.2024)
  4. github/flink [Электронный ресурс], URL: https://github.com/apache/flink (дата обращения: 24.02.2024)
  5. Docs | Apache Flink [Электронный ресурс], URL: https://nightlies.apache.org/flink/flink-docs-release-1.18/docs/ (дата обращения: 25.02.2024)
  6. Github/luigi [Электронный ресурс], URL: https://github.com/spotify/luigi (дата обращения: 26.02.2024)
  7. Обзор фреймворка Luigi для построения последовательностей выполнения задач [Электронный ресурс], URL: https://habr.com/ru/companies/otus/articles/339904/ (дата обращения: 27.02.2024)
  8. Почтальон в мире IT: для чего нужен брокер сообщений Apache Kafka и как он устроен [Электронный ресурс], URL: https://practicum.yandex.ru/blog/broker-soobsheniy-apache-kafka/ (дата обращения: 29.02.2024)
  9. Apache Kafka: основы технологии [Электронный ресурс], URL: https://habr.com/ru/companies/slurm/articles/550934/ (дата обращения: 1.03.2024)
  10. Python — это медленно. Почему? [Электронный ресурс], URL: https://habr.com/ru/companies/ruvds/articles/418823/ (дата обращения: 2.03.2024)
  11. GIL и его влияние на многопоточность Python [Электронный ресурс], URL: https://habr.com/ru/articles/592189/ (дата обращения: 3.03.2024)
  12. C++ VS Java benchmarks [Электронный ресурс], URL: https://programming-language-benchmarks.vercel.app/cpp-vs-java (дата обращения: 4.03.2024)
  13. Cmake Tutorial [Электронный ресурс], URL: https://cmake.org/cmake/help/book/mastering-cmake/cmake/Help/guide/tutorial/index.html (дата обращения: 5.03.2024)
  14. Введение в CMake [Электронный ресурс], URL: https://habr.com/ru/articles/155467/ (дата обращения: 6.03.2024)
  15. installation - How to properly set CMAKE\_INSTALL\_PREFIX from the command-line - Stack Overflow [Электронный ресурс], URL: https://stackoverflow.com/questions/6241922/how-to-properly-set-cmake-install-prefix-from-the-command-line (дата обращения: 6.03.2024)
  16. CMake и C++ — братья навек [Электронный ресурс], URL: https://habr.com/ru/articles/461817/ (дата обращения: 20.03.2024)
  17. nlohmann/json: JSON for Modern C++ [Электронный ресурс], URL: https://github.com/nlohmann/json (дата обращения: 6.03.2024)
  18. miloyip/nativejson-benchmark: C/C++ JSON parser/generator benchmark [Электронный ресурс], URL: https://github.com/miloyip/nativejson-benchmark#parsing-time (дата обращения: 7.03.2024)
  19. CppSerialization [Электронный ресурс], URL: https://www.opensourceagenda.com/projects/cppserialization (дата обращения: 7.03.2024)
  20. Fast Binary Encoding (FBE) | FastBinaryEncoding [Электронный ресурс], URL: https://chronoxor.github.io/FastBinaryEncoding/ (дата обращения: 8.03.2024)
  21. Cap'n Proto: Introduction [Электронный ресурс], URL: https://capnproto.org/ (дата обращения: 8.03.2024)
  22. Cap'n Proto: Encoding Spec [Электронный ресурс], URL: https://capnproto.org/encoding.html (дата обращения: 9.03.2024)
  23. Introduction — xtensor documentation [Электронный ресурс], URL: https://xtensor.readthedocs.io/en/latest/index.html (дата обращения: 9.03.2024)
  24. Tensor - master [Электронный ресурс], URL: https://www.boost.org/doc/libs/master/libs/numeric/ublas/doc/tensor.html (дата обращения: 10.03.2024)
  25. OpenCV: Introduction [Электронный ресурс], URL: https://docs.opencv.org/4.x/d1/dfb/intro.html (дата обращения: 10.03.2024)
  26. zeromq/cppzmq: Header-only C++ binding for libzmq [Электронный ресурс], URL: https://github.com/zeromq/cppzmq (дата обращения: 11.03.2024)
  27. «ZeroMQ».Глава 1: Приступая к работе [Электронный ресурс], URL: https://habr.com/ru/articles/198578/ (дата обращения: 11.03.2024)
  28. Class template spsc\_queue - master [Электронный ресурс], URL: https://www.boost.org/doc/libs/master/doc/html/boost/lockfree/spsc\_queue.html (дата обращения: 12.03.2024)
  29. Class template queue - master [Электронный ресурс], URL: https://www.boost.org/doc/libs/master/doc/html/boost/lockfree/queue.html (дата обращения: 12.03.2024)
  30. cds: cds::container::VyukovMPMCCycleQueue< T, Traits > Class Template Reference [Электронный ресурс], URL: https://libcds.sourceforge.net/doc/cds-api/classcds\_1\_1container\_1\_1\_vyukov\_m\_p\_m\_c\_cycle\_queue.html (дата обращения: 13.03.2024)
  31. Lock-free структуры данных. Извне: введение в libcds [Электронный ресурс], URL: https://habr.com/ru/articles/196834/ (дата обращения: 13.03.2024)
  32. Thread Management - master [Электронный ресурс], URL: https://www.boost.org/doc/libs/master/doc/html/thread/thread\_management.html#thread.thread\_management.thread.interrupt (дата обращения: 14.03.2024)
  33. ess-dmsc/h5cpp: C++ wrapper for the HDF5 C-library [Электронный ресурс], URL: https://github.com/ess-dmsc/h5cpp?ysclid=ls1idxe86j54611701 (дата обращения: 15.03.2024)
  34. Ловим утечки памяти в С/С++ [Электронный ресурс], URL: https://habr.com/ru/articles/480368/ (дата обращения: 16.03.2024)
  35. Valgrind — Википедия [Электронный ресурс], URL: https://ru.wikipedia.org/wiki/Valgrind (дата обращения: 16.03.2024)
  36. Constraints and concepts (since C++20) [Электронный ресурс], URL: https://en.cppreference.com/w/cpp/language/constraints (дата обращения: 18.03.2024)

# Приложения

**Приложение А**

**Ежедневные записи студента по практике**

|  |  |  |
| --- | --- | --- |
| Дата | Описание работы, выполненной студентом | Отметка  руководителя |
| 19.02 | Ознакомительная лекция по практике |  |
| 20.02 – 24.02 | Получение и разбор кода, который генерирует данные. Выбор и изучение библиотек: zmq, capnp. Написанеи первой реализации соединения конвейера и генератора данных. Написание кода для десериализации сырых данных, поступаемых с генератора. Корректировка дельнейшего плана работы. |  |
| 26.02 – 02.03 | Изучение CMake. Установка библиотек (boost, zmq, cap'n proto, cppzmq). Корректировка CMakeLists.txt файла (добавление поиска библиотек-зависимостей). Изучение документации установленных библиотек. Написание пула потоков для приема данных с генератора. Осуществление наброска перед написанием кода по созданию пула потоков для построения стадий конвейера |  |
| 04.03 – 09.03 | Создание пула потоков для стадий конвейера и их отладка. Рефакторинг кода, написание структур для поддержания необходимых данных в структурах данных. Написание пула потоков для осуществления распараллеливания выполнения стадий конвейера. Установка библиотеки FastBinaryEncoding, создание и компиляция схемы |  |
| 11.03 – 16.03 | Неудачная попытка внедрения библиотеки FBE в проект. Прочтение статьи (DKIST Visible Broadband Imager Data Processing Pipeline). Сборка проекта в режиме Release (+ перекомпиляция библиотек зависимостей). Установка библиотек xtensor и OpenCV, изучение документации, внедрение в проект. Установка библиотеки для многопоточной очереди типа mpmc из библиотеки cds, а также ее интеграция в проект Поиск инструмента для отслеживания утечек памяти. Оптимизация копирований при приеме данных. Написание потока для мониторинга работы стадий конвейера |  |
| 18.03 – 23.03 | Оптимизация передачи данных между стадиями конвейера (оптимизация копирования). Усовершенствование потока для мониторинга конвейера: добавлено снятие замеров времени, которое стадии затрачивают на выполнение функтора. Создание проекта для проведения бенчмарков различных библиотек, а также написание бенчмарка для тестирования скорости коннекторов (очередей) и тестирования структур данных, в которых хранятся данные, которые обрабатываются стадиями конвейера. Рефакторинг и отладка кода бенчмарков |  |
| 25.03 – 29.03 | Написание Менеджера для десериализации данных (для cv::Mat и для xTensor). Написание скрипта на python для построения графика по данным результатов работы бенчмарка. Декомпозиция кода. Написание генератора для выдачи 4к изображений из видео. Интеграция функций cv::Mat в проект и тестирование работы конвейера. |  |
| 30.03 | Защита практик |  |

Отзыв руководителя:

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# приложение б