Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем
Системное программирование

Алексей Андреевич Фефелов

Архитектура фреймворка MIRF

Бакалаврская работа

Научный руководитель:
доц. кафедры СП, к. т. н., Ю. В. Литвинов

Рецензент:
ООО "НМЦ-Томография", заведующий отделением МРТ, врач-рентгенолог,
к. м .н, Е. П. Магонов

Санкт-Петербург
2020

SAINT-PETERSBURG STATE UNIVERSITY

Software and Administration of Information Systems
Software Engineering

Aleksei Fefelov

MIRF Framework architecture

Bachelor's Thesis

Scientific supervisor:
C.Sc., assistant prof. Yurii Litvinov

Reviewer:
C.Sc., NMC-Tomography LLC, Head of MRI department in Veteran's hospital, radiologist
Evgenii Magonov

Saint-Petersburg
2020

Оглавление

Введение

Обработка медицинских исследований является достаточно сложной задачей. К примеру, размер МРТ снимков может достигать 10Гб, а время их обработки на домашнем компьютере — восьми и более часов.

В 2018 году была создана система MIRF [8] (Medical Image Research Framework) с открытым исходным кодом, предназначенная для обработки медицинских изображений. В данный момент она позволяет обрабатывать снимки МРТ и результаты ЭКГ-исследований (поддержка ЭКГ добавлена весной 2020 года).

Для того, чтобы быстро разработать прототип, а также обеспечить необходимую гибкость, авторы выбрали Pipes & Filter [10] в качестве основного архитектурного стиля.

Сейчас фреймворк имеет монолитную архитектуру, из-за чего возникают проблемы, связанные с масштабируемостью и скоростью обработки медицинских данных: обработка одного набора данных может занимать часы и обработать большое количество данных на одном компьютере достаточно затруднительно. При этом многие этапы обработки МРТ снимков являются независимыми, но сейчас вычисления можно сделать параллельными только в пределах одного компьютера, чего зачастую недостаточно.

Современным способом решать такие проблемы являются микросервисы. Микросервисная архитектура является одной из разновидностей сервис-ориентированной архитектуры [5], направленной на разработку микросервисов — небольших, легко изменяемых модулей, слабо (насколько это возможно) связанных между собой.

Для того, чтобы обеспечить возможность выполнения отдельных этапов обработки медицинских данных независимо друг от друга и поддерживать производительность на высоком уровне при любом количестве пользователей, было решено разработать микросервисную архитектуру, где каждый сервис предоставляет возможность выполнить один из этапов обработки данных, реализовать и интегрировать ее в MIRF.

1. Постановка задачи

Целью данной работы является разработка микросервисной архитектуры для фреймворка MIRF. Для достижения этой цели были поставлены следующие задачи:

- сделать обзор предметной области: рассмотреть существующую архитектуру фреймворка MIRF, архитектуру Pipes and Filters, популярные системы для высокопроизводительных вычислений;
- разработать микросервисную архитектуру для фреймворка MIRF;
- провести архитектурный рефакторинг в соответствии с разработанной архитектурой;
- провести тестирование и апробацию.

2. Обзор предметной области

2.1. Pipes and Filters

Pipes and Filters [10] (Pipeline) — это достаточно простая, но при этом надежная и гибкая архитектура. Основные ее сущности — это каналы (Pipe), по которым передаются данные, и фильтры (Filters), которые обрабатывают, преобразуют или фильтруют данные перед тем, как отправить их по каналу другим компонентам (Рис. 1). Зачастую эта архитектура используется как простая последовательность каналов и фильтров, однако она может применяться для любых, сколь угодно сложных, структур.

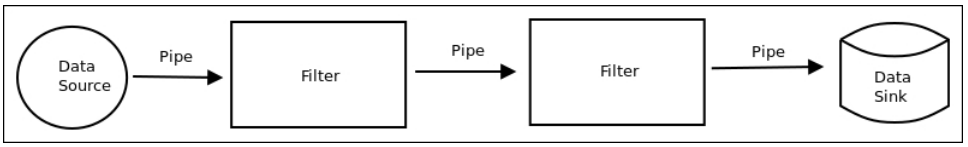

Рис. 1: Pipeline. Источник изображения: `https://clck.ru/NiHDR`. Дата обращения: 25.05.2020.

2.2. Фреймворк MIRF

MIRF [8] (Medical Image Research Framework) был создан с целью упрощения процесса создания медицинских приложений, использующих и обрабатывающих медицинские данные (в первую очередь, поддержаны МРТ-снимки, но есть архитектурная поддержка и других типов данных). Данная библиотека написана на языке программирования Kotlin, при необходимости может быть легко интегрирована в любые JVM-совместимые проекты, включая даже мобильные приложения.

2.2.1. Структура фреймворка

Библиотека MIRF содержит два основных пакета — Features и Core.

- Core содержит минимальный набор сущностей, необходимый для работы библиотеки — интерфейсы, абстрактные классы, а также

классы, реализующие внутреннее представление данных (MedImage, ImageSeries и др.), их обработку и передачу.

- Features — это набор модулей с пользовательской функциональностью, которая используется для облегчения разработки: классы для доступа к данным, анализа изображения, адаптеры для разных форматов данных, фильтры, механизмы генерации отчетности и др. Пользователи могут расширять пакет Features.

Диаграмма классов пакета Core представлена на Рис. 2. На ней видно, что есть набор алгоритмов и блоки. Блоки инкапсулируют в себе алгоритмы. Алгоритмы принимают на вход один объект (InputData) и возвращают один объект (OutputData). Для обработки данных пользователь строит конвейер (Pipeline) из этих блоков, который является направленным графом, имеющим один вход и один выход, и запускает его.

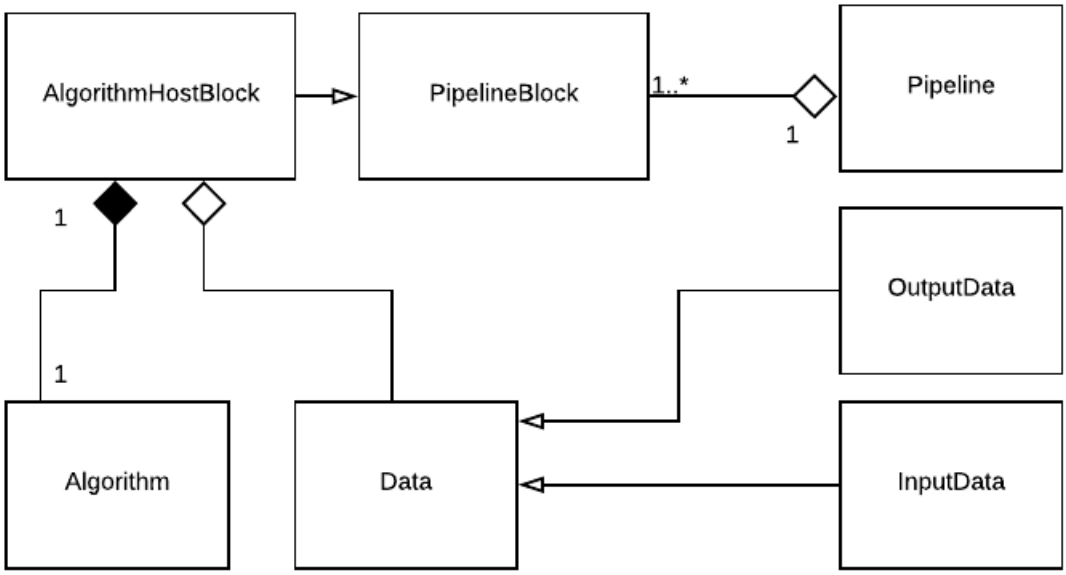

Рис. 2: диаграмма классов (источник: [9])

2.2.2. Обработка данных

Базовый сценарий использования библиотеки предполагает создание конечным пользователем небольших программ для обработки данных (МРТ-снимков, результатов ЭКГ-исследований и др.). Пользователи пишут конвейеры (Pipeline), то есть последовательность блоков, которые обрабатывают исходные данные. Конвейеры обязательно имеют один вход (входные данные) и один выход (выходные данные). Каждый блок обработки принимает объект с входными данными и может использовать какие-то статические ресурсы. Например, блок, загружающий изображения в память, принимает путь к файлам и требует наличия хотя бы одного изображения, находящегося по заданному пути на диске. Внутренние блоки могут иметь несколько входов и выходов и также могут использовать данные на диске (загружать или сохранять нужную им информацию).

Пример простейшего конвейера, обрабатывающего медицинские изображения, представлен на Рис. 3.

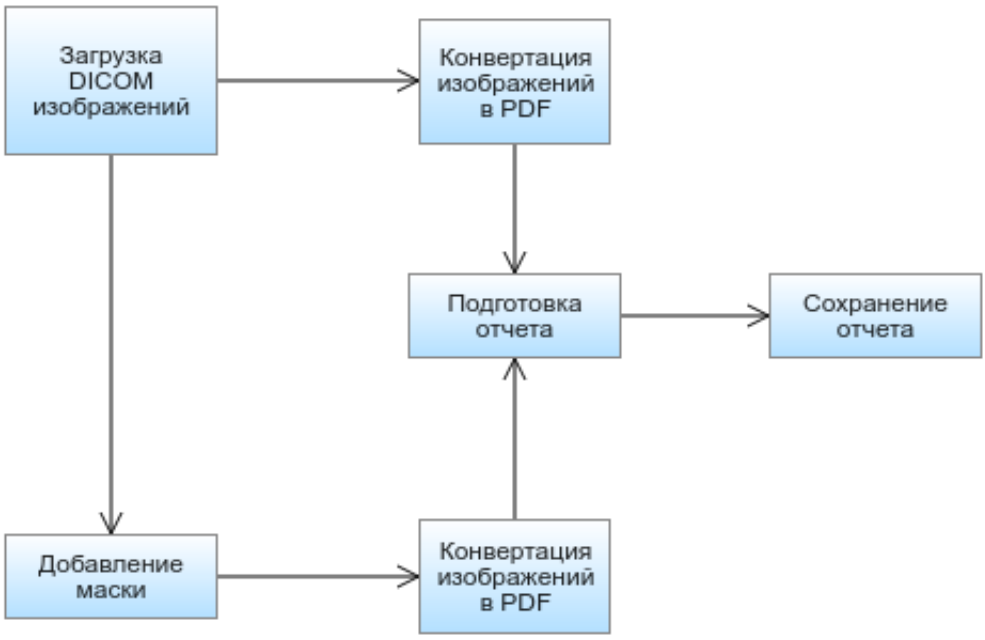

Рис. 3: пример простого конвейера

Здесь первый блок загружает из памяти серию DICOM изображе-

ний, на часть из них накладывает маску, после этого конвертирует все изображения в PDF формат и сохраняет результат.

2.2.3. Особенности фреймворка MIRF

Сейчас весь цикл обработки одного набора данных может быть осуществлен только на одном компьютере, что неудобно, поскольку многие этапы являются независимыми. Кроме того, некоторые этапы занимают существенно больше времени, но в данный момент есть возможность сделать вычисления параллельными только в пределах одного компьютера, а при одновременной обработке нескольких наборов данных с использованием библиотеки MIRF наблюдается сильное ухудшение производительности. Помимо этого, предполагается, что конечными пользователями данной библиотеки будут врачи. Не все из них умеют программировать и даже при наличии приложения с удобным пользовательским интерфейсом им будет намного удобнее и проще воспользоваться веб-сервисом, чем устанавливать приложение локально на свой компьютер и запускать обработку данных на нем.

2.3. Популярные системы для высокопроизводительных вычислений

Нам необходимо разработать систему микросервисов, которая будет обрабатывать большие объемы данных, поэтому стоит рассмотреть системы для высокопроизводительных вычислений. Одними из наиболее популярных таких систем являются Apache Hadoop [1] и Apache Spark [2]. Они были созданы для того, чтобы обеспечить возможность параллельной обработки больших объемов данных[1] на кластерах.

2.3.1. Apache Hadoop и Apache Spark

Apache Hadoop [1] представляет из себя фреймворк для создания параллельных приложений, исполняемых на кластерах из тысяч серве-

[1]Данные системы целесообразно использовать при объеме обрабатываемых данных более нескольких терабайт

ров. Проект разработан на Java и использует MapReduce [4] в качестве основной вычислительной парадигмы.

Данные хранятся в распределенной файловой системе HDFS [3], а вычисления производятся на кластерах.

Типичный кластер состоит из следующих сущностей: JobTracker, NameNode, Secondary NameNode, Datanode и TaskTracker.

- JobTracker принимает запросы от клиентов, занимается балансировкой нагрузки и непосредственно следит за исполнением задач.
- NameNode является мозгом системы, хранит в себе все данные, необходимые для обеспечения вычислений, знает, какие данные лежат на каждом узле Datanode. Обычно, такой узел один на кластер (в противном случае это Namenode Federation [11])
- Secondary NameNode хранит в себе состояние файловой системы и логи. Он необходим для быстрого восстановления NameNode, если тот выйдет из строя.
- DataNode — узлы с данными. Непосредственно хранят файлы, раз в час отправляют отчет о своем состоянии NameNode.
- TaskTracker принимают задачи от JobTracker в виде jar-файла и исполняют их. Физически находятся на одних серверах с DataNode.

Все, кроме последних двух сущностей, находятся на отдельных узлах.

Apache Spark [2] во многом похож на Hadoop. Основное отличие состоит в том, что данные не сохраняются на диск, а хранятся и обрабатываются прямо в оперативной памяти. За счет этого скорость их обработки повышается в 10-100 раз по сравнению с Apache Hadoop.

3. Архитектура

Основные требования, предъявляемые к новой архитектуре — это:

- масштабируемость — мы должны иметь возможность запуска любого количества серверов для обработки данных пользователей;
- скорость — один набор входных данных должен обрабатываться почти с той же скоростью, что и раньше, или быстрее, если в конвейере есть параллельные участки;
- простота перехода — требуется как можно меньше изменять существующий код библиотеки.

Архитектура имеет три основные сущности: Orchestrator (Оркестратор), Block (Блок) и Repository (Репозиторий), как и в Apache Hadoop, но в связи с тем, что данные обрабатываются почти всегда последовательно, а не параллельно, мы отказались от распределённого хранения данных и паттерна MapReduce.

3.1. Хранение данных

Учитывая специфику системы, у нас нет нужды в постоянном хранении каких-либо данных и разработке распределенной файловой системы. Нам необходимо хранить только промежуточные результаты вычислений при обработке одного набора МРТ-снимков (или других данных).

Сервис Repository представляет из себя микросервис, который хранит промежуточные данные и предоставляет доступ к ним. Обычно он один на всю систему, но при большой нагрузке их может быть и несколько.

Сервис Repository является отдельным сервисом и не занимается ничем, кроме хранения данных.

3.2. Блоки и алгоритмы

Для обеспечения простоты перехода была выделена сущность Block.

Block — это микросервис, инкапсулирующий в себе один (и только один) алгоритм. Не имеет внутреннего состояния — это необходимо для обеспечения масштабируемости. Его задача — получать данные из репозитория, обрабатывать их, сериализовать результат и загружать его обратно в сервис Repository.

3.3. Управление системой

Управление всей системой осуществляет сервис Orchestrator. Он получает задания от пользователей (пользовательского приложения), отправляет задания сервисам Block, следит за текущим состоянием системы, занимается балансировкой нагрузки.

Сервис “Report Service” готовит отчет для пользователя.

Схема взаимодействия микросервисов представлена на Рис. 4.

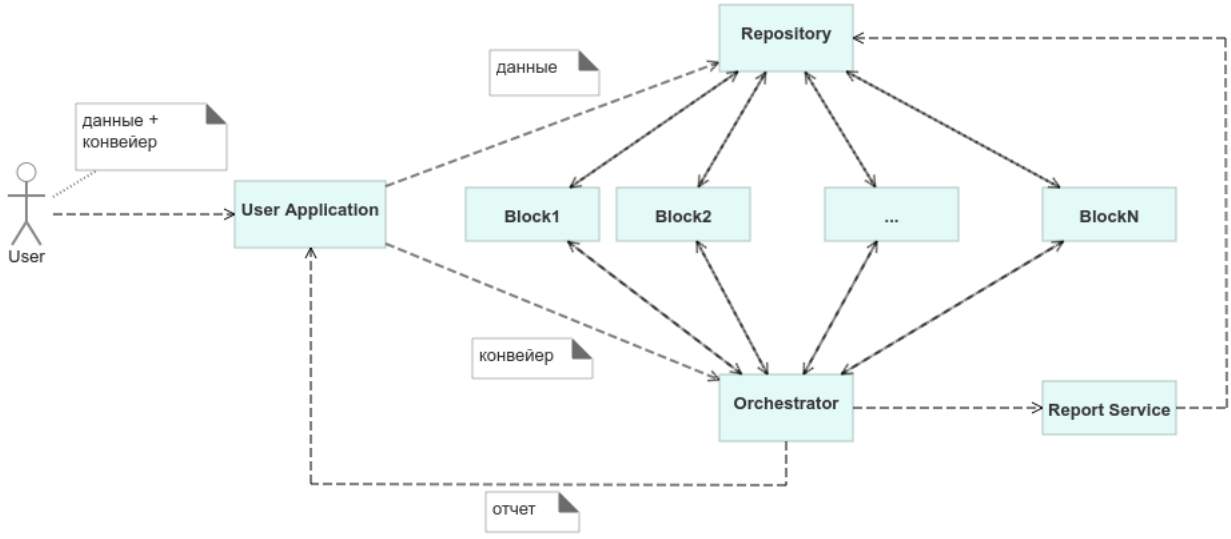

Рис. 4: схема взаимодействия микросервисов

3.4. Обработка данных

Схема обработки одного набора данных следующая: пользователь (пользовательское приложение) загружает данные в сервис Repository и отправляет конвейер сервису Orchestrator. Далее сервис Orchestrator принимает конвейер в обработку, отправляет первому блоку задание начать обработку. Как только сервис Block завершает обработку, он уведомляет об этом сервис Orchestrator, который, в свою очередь, отправляет задание следующему блоку, и так пока данные не будут полностью обработаны. Независимые ветки конвейера могут обрабатываться параллельно разными блоками.

Диаграмма последовательности для одного набора входных данных представлена на Рис. 5.

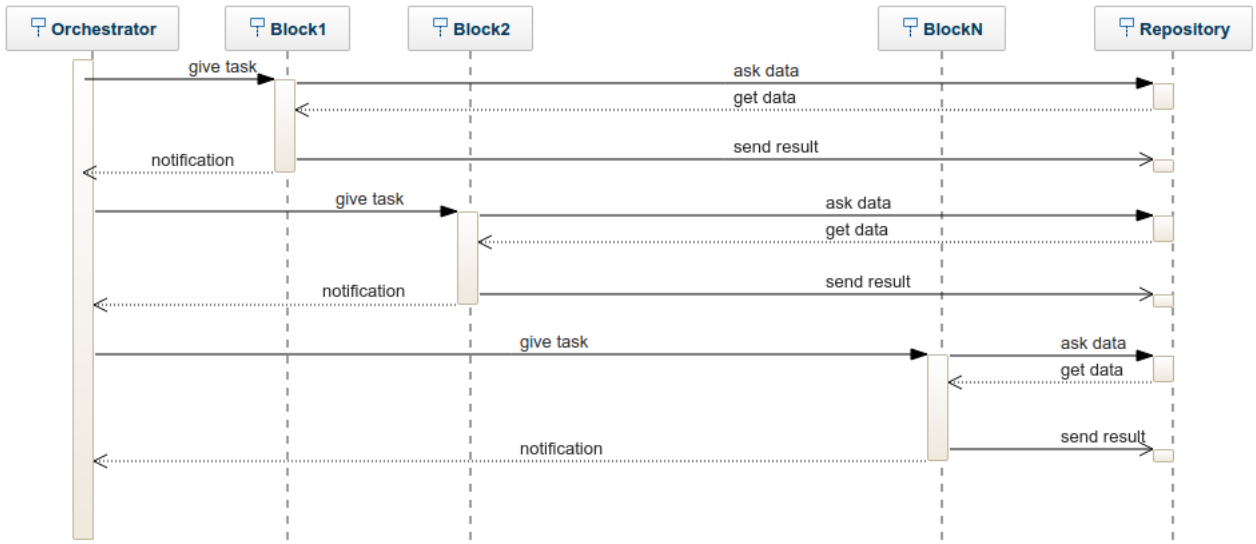

Рис. 5: диаграмма последовательности

4. Детали реализации

В данной главе будут рассмотрены детали реализации микросервисной архитектуры.

4.1. Хранение данных

Промежуточные данные хранятся в репозиториях в виде zip-архивов. Доступ к ним имеют микросервисы, занимающиеся обработкой данных, и сервис Orchestrator. Структура zip-архива с данными представлена на Рис. 6. InputObject — это сериализованный объект, входные данные алгоритма. Папка Data может содержать дополнительные статические файлы (например, DICOM изображения, или метаинформацию, необходимую алгоритму для корректной обработки входных данных).

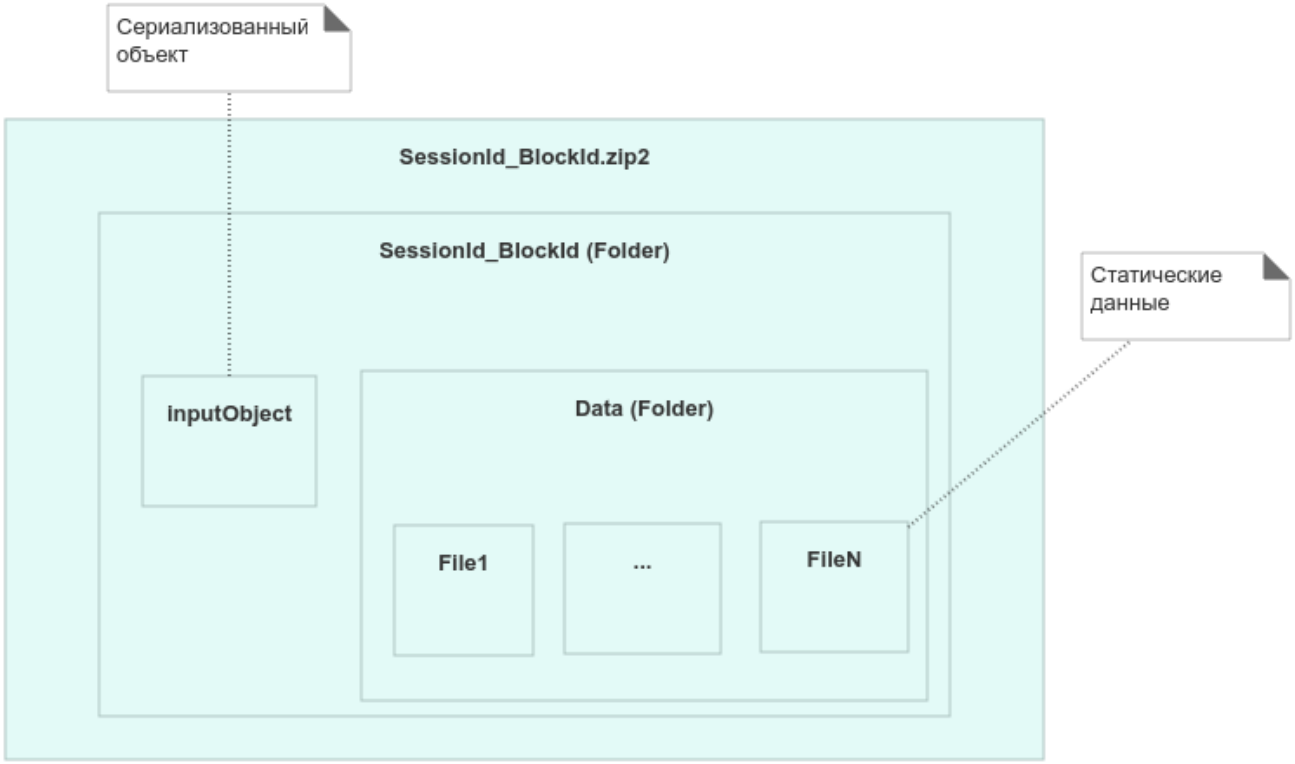

Рис. 6: структура архива с данными

4.2. Взаимодействие сервисов

Схема взаимодействия микросервисов (а также пользовательского приложения) в рамках одного цикла обработки данных следующая:

1. Пользователь запрашивает уникальный идентификатор сессии у сервиса Orchestrator (SessionId).

2. Сервис Orchestrator генерирует SessionId, привязывает один из микросервисов для хранения промежуточных данных (Repository) к данному SessionId и уведомляет об этом его. После этого отправляет приложению SessionId и адрес репозитория, в который необходимо загрузить данные пользователю.

3. Пользователь загружает отправляет данные сервису Repository в виде архива с требуемой структурой (см. 4.1) и конвейер сервису Orchestrator в виде JSON-строки.

4. Сервис Orchestrator проверяет корректность конфигурации и строит по ней граф (конвейер). Если в конфигурации содержатся ошибки, удаляет сессию и отправляет отчет об ошибке.

5. Сервис Orchestrator берет из конфигурации свободные узлы (то есть узлы, входные данные для которых готовы и загружены в сервис Repository), выбирает соответствующие наименее загруженные микросервисы для дальнейшей обработки этих данных и говорит им начать обработку.

6. Сервисы Block скачивают архив из сервиса Repository, берут оттуда сериализованный объект input и, при необходимости, статические данные, находящиеся рядом в папке Data, обрабатывают их, сериализуют результат, архивируют и отправляют обратно в Repository; уведомляют сервис Orchestrator об успешном или неуспешном завершении, а также о причинах падения в случае ошибки.

7. Далее, повторяются пункты 5-6 до тех пор, пока конвейер не закончится.

8. Как только конвейер заканчивается (последний блок обработал данные) сервис Orchestrator удаляет сессию, генерирует и отправляет отчет.

4.3. Конфигурирование сервисов

Микросервисы Repository и Orchestrator — это стандартные Spring Boot приложения и не требуют никакой дополнительной конфигурации. Для запуска сервиса Block необходимо рядом с исполняемым файлом положить конфигурационный файл configuration.json, который имеет следующий вид:

```
{
  ''blockType'': ''ReadDicomImageSeriesAlg'',
  ''inputClassName'': ''com.mirf.features.repositoryaccessors.data.RepoRequest'',
  ''outputClassName'': ''com.mirf.core.data.medimage.ImageSeries'',
  ''algorithmClassName'': ''com.mirf.features.dicomimage.util.
    ReadDicomImageSeriesAlg'',
  ''taskLimit'': 2,
  ''orchestratorUri'': ''http://localhost:5000/''
}
```

- blockType — это тип блока. Используется сервисом Orchestrator при поиске сервисов для обработки данных. Пользователь указывает в конвейере именно его;

- inputClassName, outputClassName, algorithmClassName — полное имя соответствующих классов (со всеми пакетами). Здесь мы говорим, что обрабатывать данные будет класс algorithmClassName, который имеет метод execute, принимающий на вход объект класса inputClassName и возвращающий объект класса outputClassName;

- taskLimit — максимально допустимое количество одновременно обрабатываемых заданий. Рекомендуется выставлять значение, равное числу процессоров на сервере;

- orchestratorUri — адрес сервиса Orchestrator.

4.4. Конвейеры

Раньше конвейеры конфигурировались непосредственно в коде: создавался конвейер, добавлялись блоки и связи между ними. Более подробно это описано в выпускной квалификационной работе А.В. Ломакина в 2019 году [6]. Например, код, конфигурирующий конвейер, представленный на Рис. 3, выглядел следующим образом:

```
fun exec(dicomFolderLink: String, resultFolderLink: String) {
    val pipe = Pipeline("apply circle mask to dicom")
    //initializing blocks
    val seriesReaderBlock = AlgorithmHostBlock(DicomRepoRequestProcessors
            .readDicomImageSeriesAlg, pipelineKeeper = pipe)
    val addMaskBlock = AlgorithmHostBlock(AddCircleMaskAlg()
            .asImageSeriesAlg(), pipelineKeeper = pipe)
    val imageBeforeReporter = AlgorithmHostBlock<ImageSeries, PdfElementData>
            ({ x -> x.asPdfElementData() }, "image before", pipe)
    val imageAfterReporter = AlgorithmHostBlock<ImageSeries, PdfElementData>
            ({ x -> createHighlightedImages(x) }, "image after", pipe)
    val pdfBlock = AccumulatorWithAlgBlock(
            PdfElementsAccumulator("report"), 2, "Accumulator", pipe)
    val reportSaverBlock = RepositoryAccessorBlock<FileData, Data>(
            LocalRepositoryCommander(), RepoFileSaver(), resultFolderLink)
    //making connections
    seriesReaderBlock.dataReady += addMaskBlock::inputReady
    seriesReaderBlock.dataReady += imageBeforeReporter::inputReady
    addMaskBlock.dataReady += imageAfterReporter::inputReady
    imageBeforeReporter.dataReady += pdfBlock::inputReady
    imageAfterReporter.dataReady += pdfBlock::inputReady
    pdfBlock.dataReady += reportSaverBlock::inputReady
    //create initial data
    val init = RepoRequest(dicomFolderLink, LocalRepositoryCommander())
    //print every new session record
    pipe.session.newRecord += { _, b -> println(b) }
    pipe.rootBlock = seriesReaderBlock //run
    pipe.run(init)
}
```

Теперь конвейеры задаются в виде JSON-файлов. Например, конвейер для того же самого примера будет выглядеть следующим образом:

```
[
  { ''id'': 0, ''blockType'' : ''ReadDicomImageSeriesAlg'', ''children'': [1, 2] },
  { ''id'': 1, ''blockType'' : ''AddMaskAlg'', ''children'': [3] },
  { ''id'': 2, ''blockType'' : ''ConvertImagesToPdfAlg'', ''children'': [4] },
  { ''id'': 3, ''blockType'' : ''ConvertImagesToPdfAlg'', ''children'': [4] },
  { ''id'': 4, ''blockType'' : ''PrepareReportAlg'', ''children'': [5] },
  { ''id'': 5, ''blockType'' : ''SaveReportAlg'', ''children'': [] }
]
```

По сути, здесь задается направленный ацикличный граф, имеющий одну начальную (входную) вершину и одну конечную (выходную) вершину. Теперь мы имеем возможность легко генерировать такие конвейеры внутри пользовательского приложения, или веб-сервиса. Для пользователя (врача) это будет выглядеть подобно Рис. 3.

Фронтенд не является частью данной работы, но подобный формат представления позволяет легко подключить графовый редактор, например, REAL.NET Web [7].

5. Допущения и ограничения

Предложенная архитектура накладывает некоторые ограничения:

1. Все типы входных и выходных данных для всех используемых алгоритмов должны быть сериализуемы. В противном случае их не получится передавать по сети.

2. Необходимо учитывать структуру хранения файлов и внутренние соглашения. Например, статические данные всегда лежат в папке Data архива с данными, а файл, который содержит сериализованный объект, являющийся параметром алгоритма, всегда называется “input”.

3. В случаях, когда конвейер достаточно большой, а самих обрабатываемых данных не много, время передачи данных между микросервисами может превышать время обработки данных. При n каналах в конвейере для обработки данных необходимо, как минимум, $2n + 2$ передачи данных между микросервисами.

6. Тестирование и апробация

Для тестирования был взят конвейер, обрабатывающий результаты ЭКГ-исследований (Рис. 7).

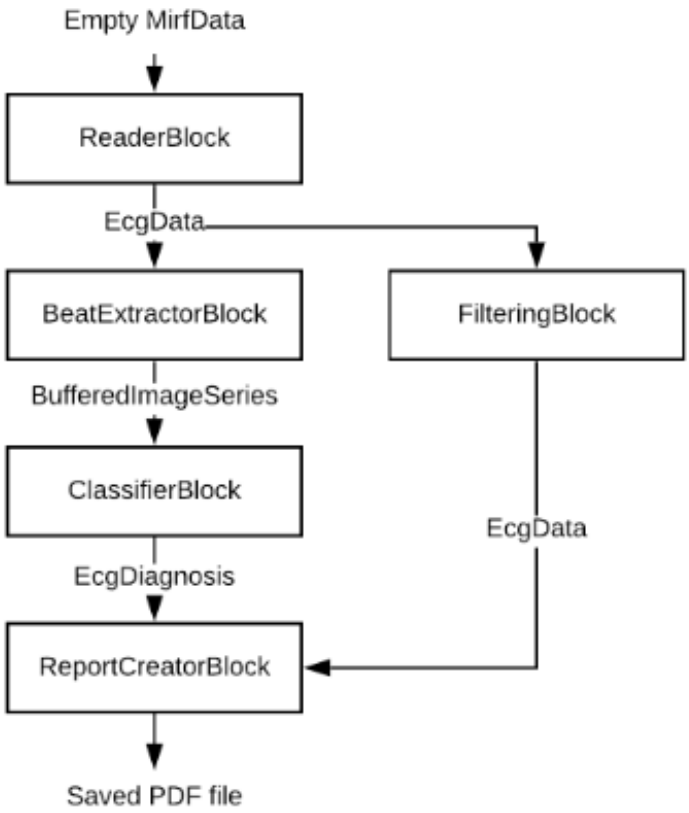

Рис. 7: тестовый конвейер

Данный конвейер ищет аритмии и генерирует отчет с предположительным диагнозом.

6.1. Производительность

Для тестирования производительности были взяты три ЭКГ исследования, содержащие по 1893 удара.

Было произведено 10 запусков, в каждом из которых обрабатывалось 6 конвейеров.

1. Сначала были произведены запуски конвейеров на предыдущей версии библиотеки, использующей монолитную архитектуру. Среднее время обработки 6 исследований составило 840 секунд, при среднеквадратическом отклонении 17 секунд. При этом исполнение блока ClassifierBlock занимало 85% от всего времени исполнения конвейера.

2. После этого была собрана сеть из 7 микросервисов на 6 компьютерах (Рис. 8).

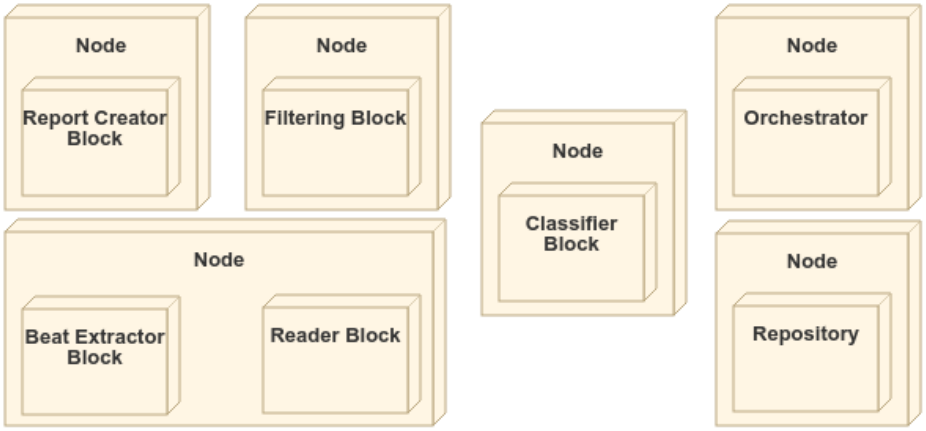

Рис. 8: диаграмма развертывания 1

Node — это физический узел (компьютер), а Block, Orchestrator и Repository — микросервисы. На всех узлах, кроме одного, поднято по одному микросервису.

Среднее время обработки 6 исследований составило 686 секунд, при среднеквадратическом отклонении 28 секунд.

3. Учитывая тот факт, что исполнение блока ClassifierBlock занимает больше всего времени, система была переконфигурирована: сервисы ReportCreatorBlock и FilteringBlock были подняты на одном компьютере, а на освободившемся компьютере был поднят ещё один ClassifierBlock (Рис. 9).

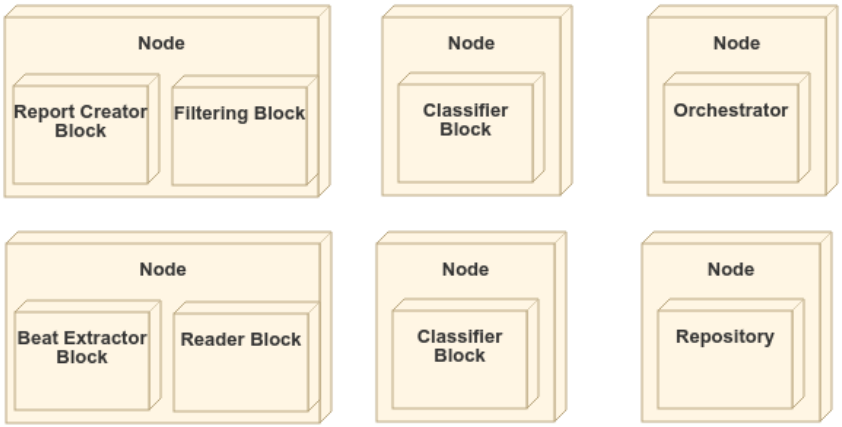

Рис. 9: диаграмма развертывания 2

Среднее время обработки 6 исследований составило 384 секунды, при среднеквадратическом отклонении 25 секунд.

Результаты измерений сведены в таблицу 1 (время указано в секундах). В ней E —мат. ожидание, а σ — среднеквадратическое отклонение. Колонки “монолит”, “микросервисы 1” и “микросервисы 2” соответствуют пунктам 1, 2 и 3 соответственно.

	монолит	микросервисы 1	микросервисы 2
E	840	686	384
σ	17	28	25

Таблица 1: Результаты измерений

Таким образом, видно, что переход к микросервисной архитектуре позволяет увеличить скорость обработки данных при правильном конфигурировании сети.

6.2. Масштабируемость и балансировка нагрузки

Для тестирования масштабируемости была запущена сеть из сервисов Orchestrator, Repository и пяти блоков (Block), представленных на Рис. 7 (по 1 каждого типа). После этого поочередно запускались дополнительные блоки до тех пор, пока не стало по 5 блоков каждого типа.

После этого было запущено 15 конвейеров и каждые 10 секунд делались снимки состояния системы, по которым быо видно, что нагрузка распределяется равномерно.

Заключение

В ходе данной работы были получены следующие результаты:

- сделан обзор фреймворка MIRF и других популярных систем для высокопроизводительных вычислений;
- разработана микросервисная архитектура, позволяющая организовать распределенные вычисления;
- архитектурный рефакторинг проведен, в частности, реализован механизм конфигурирования и запуска веб-сервиса, предоставляющего доступ к алгоритмам библиотеки MIRF, а также алгоритм взаимодействия сервисов;
- проведено тестирование и апробация, которые показали увеличение скорости вычислений, а также возможность масштабирования системы, что подтверждает оправданность микросервисной архитектуры;
- статья “MIRF 2.0 — a framework for distributed medical images analysis” принята к публикации на конференции SEIM-2020.

Список литературы

[1] Apache Hadoop. — Accessed: 2020-05-25. URL: https://hadoop.apache.org/.

[2] Apache Spark. — Accessed: 2020-05-25. URL: https://spark.apache.org/.

[3] The Architecture of Open Source Applications / Juliana Freire, D Koop, Emanuele Santos et al. — 2011. — 01.

[4] Bhandarkar Milind. MapReduce programming with apache Hadoop // 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS) / IEEE. — 2010. — P. 1–1.

[5] Komoda Norihisa. Service oriented architecture (SOA) in industrial systems // 2006 4th IEEE International Conference on Industrial Informatics / IEEE. — 2006.

[6] Lomakin Alexander. MIRF library and its application in multiple sclerosis analysis. — 2019. — Accessed: 2019-12-14. URL: http://se.math.spbu.ru/SE/diploma/2019/bmo/444-Lomakin-report.pdf.

[7] M. Kidiankin Y. Litvinov V. Ivasheva et al. REAL.NET Web — web-based multilevel domain-specific modeling platform. — 2020.

[8] MIRF main page. — Accessed: 2019-12-14. URL: https://github.com/MathAndMedLab/Medical-images-research-framework.

[9] Musatian Sabrina. Library for development of medical images processing software. — 2019. — Accessed: 2019-12-15. URL: http://se.math.spbu.ru/SE/diploma/2019/pi/Musatian-report.pdf.

[10] Vermeulen Allan Beged-dov Gabe Thompson Patrick. The Pipeline Design Pattern. — 1995.

[11] Vorapongkitipun C., Nupairoj N. Improving performance of small-file accessing in Hadoop // 2014 11th International Joint Conference

on Computer Science and Software Engineering (JCSSE). — 2014. — P. 200–205.