

Архитектура фреймворка MIRF

Алексей Фефелов

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

рецензент: главный рентгенолог клиник Скандинавия, к. м. н., Е. П. Магонов

13.06.2020

Здравствуйте, меня зовут Алексей Фефелов. Тема моей дипломной работы “архитектура фреймворка MIRF”.

Введение

- ▶ MIRF — Medical Image Research Framework
- ▶ Анализ снимков МРТ, результатов ЭКГ-исследований
- ▶ Объем изображения 800Мб — 10Гб
- ▶ Сейчас монолитная Pipe-And-Filter архитектура
- ▶ Хочется иметь микросервисную архитектуру

Два года назад был создан фреймворк MIRF – библиотека для обработки медицинских данных. В основном он предназначен для обработки медицинских изображений, таких, как снимки МРТ. Поддерживает форматы NifTI и DICOM. В этом году также добавлена поддержка ЭКГ исследований. Обработка медицинских данных является достаточно сложной вычислительной задачей. Например, размер МРТ снимков может достигать до 10Гб, а время их обработки занимать часы. Сейчас MIRF имеет монолитную Pipes and Filters архитектуру и для того, чтобы обработать данные необходимо скачивать библиотеку, собирать ее из исходников и писать программу для их обработки. Поэтому было принято решение перевести MIRF на микросервисную архитектуру.

Мотивация

- ▶ Удобство для пользователей
- ▶ Ускорение вычислений
- ▶ Масштабируемость

Что это нам дает: во-первых, удобство (как для конечных пользователей – врачей, поскольку обработка данных теперь будет происходить не на их компьютере, а на наших серверах, так и для программистов, которые будут разрабатывать десктоп или веб приложение, ведь теперь им теперь не нужно разбираться с внутренним устройством библиотеки, а достаточно просто воспользоваться REST-интерфейсом).

Обработка медицинских данных имеет такую особенность, что зачастую отдельные этапы обработки выполняются сильно медленнее остальных, поэтому микросервисная архитектура позволит во многих случаях добиться ускорения вычислений за счет грамотного конфигурирования сети.

Постановка задачи

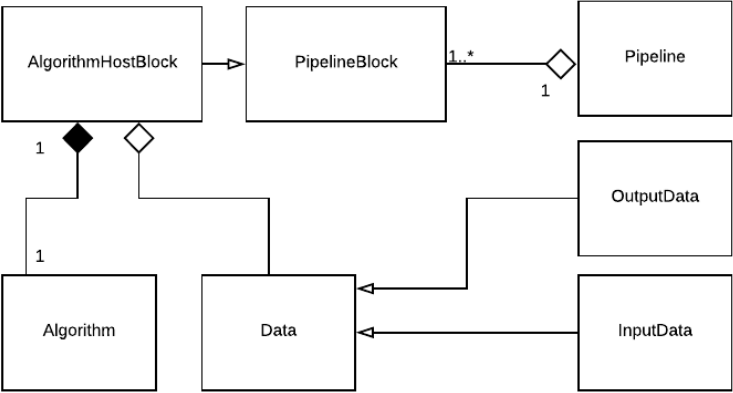
Цель работы

- ▶ Разработать и внедрить микросервисную архитектуру в фреймворк MIRF
- ▶ Сделать обзор предметной области
- ▶ Разработать микросервисную архитектуру
- ▶ Провести архитектурный рефакторинг в соответствии с предложенной архитектурой
- ▶ Провести тестирование и апробацию

Цель данной дипломной работы — разработать и внедрить микросервисную архитектуру в фреймворк MIRF.
Для ее достижения поставлены следующие задачи:

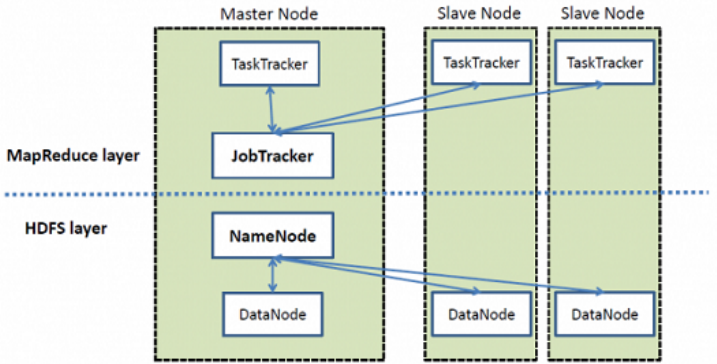
- Сделать обзор предметной области
- Разработать микросервисную архитектуру
- Провести архитектурный рефакторинг
- Провести тестирование и апробацию

MIRF Framework



Сейчас MIRF устроен таким образом, что есть набор алгоритмов и блоки, инкапсулирующие в себе эти алгоритмы. Для обработки данных пользователь строит конвейер из этих блоков, который является направленным графом, имеющим один вход и один выход, и запускает его.

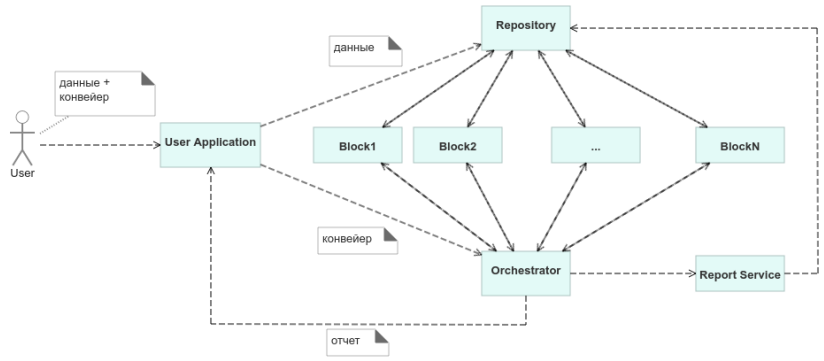
High Level Architecture of Hadoop



Одними из наиболее популярных систем для высокопроизводительных вычислений являются Apache Hadoop и Apache Spark — это инструменты для организации распределенных вычислений. Они используют парадигму Map-Reduce и вычисления в них организованы таким образом, что есть множество узлов, которые занимаются вычислениями и, как правило, один единственный узел, который распределяет задачи и занимается балансировкой нагрузки. Данные хранятся в распределенной файловой системе. Принципиальное отличие Apache Spark от Apache Hadoop состоит в том, что в Apache Spark данные хранятся в оперативной памяти, благодаря чему скорость их обработки, по заверению авторов, возрастает в разы.

источник изображения: <https://clck.ru/N9Ark>

Архитектура системы (1)



Предложенная архитектура MIRF имеет 3 основные сущности: репозиторий хранит промежуточные данные вычислений и предоставляет удобный доступ к ним. По умолчанию один, при необходимости их может быть и несколько. Блоки – это микросервисы, инкапсулирующие в себе алгоритмы. Не имеют внутреннего состояния. Оркестратор следит за внутренним состоянием сети, занимается управлением и балансировкой нагрузки. ReportService – это блок, генерирующий отчет для пользователя. Оркестратор получает от пользовательского приложения конвейер, строит по нему граф, отправляет свободным блокам команду, в которой содержится информация о том, из какого репозитория взять данные и куда отправить результат.

Архитектура системы (2)

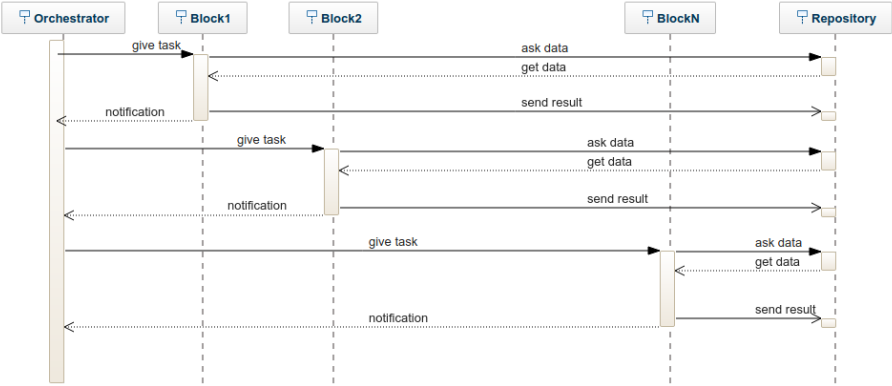
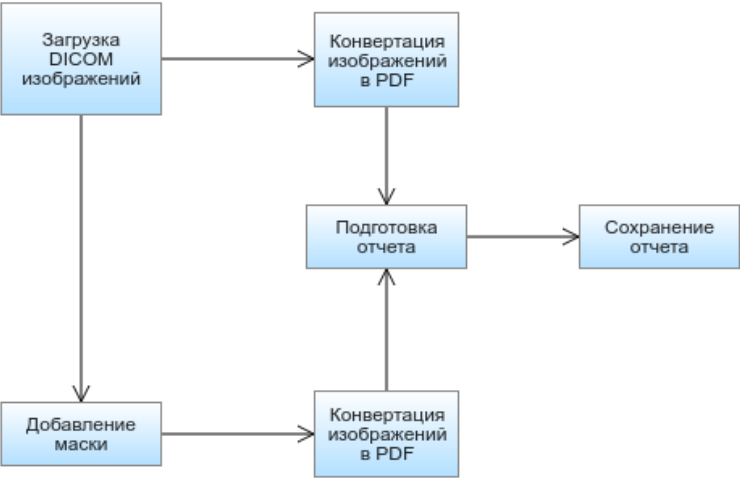


Схема взаимодействия сервисов в рамках одного цикла обработки данных следующая: Оркестратор в каждый момент времени берет из графа блоки, данные для которых уже готовы, отправляет им команду на обработку этих данных. Как только они завершают обработку и загружают результат, все повторяется. Оркестратор смотрит, кто должен обрабатывать данные дальше и отправляет ему команду.

Пример конвейера: граф



Пример простого конвейера представлен на слайде. Этот конвейер загружает два DICOM-изображения, накладывает на одно из них маску, конвертирует изображения в PDF, объединяет их и сохраняет результат.

Пример конвейера: раньше

```
fun exec(dicomFolderLink: String, resultFolderLink: String) {
    val pipe = Pipeline( name: "apply circle mask to dicom")
    //initializing blocks
    val seriesReaderBlock : AlgorithmHostBlock<RepoRequest, ImageSeries> = AlgorithmHostBlock(
        DicomRepoRequestProcessors.readDicomImageSeriesAlg,
        pipelineKeeper = pipe)
    val addMaskBlock : AlgorithmHostBlock<ImageSeries, ImageSeries> = AlgorithmHostBlock(
        AddCircleMaskAlg().asImageSeriesAlg(),
        pipelineKeeper = pipe)
    val imageBeforeReporter = AlgorithmHostBlock<ImageSeries, PdfElementData>(
        { x -> x.asPdfElementData() },
        name: "image before", pipe)
    val imageAfterReporter = AlgorithmHostBlock<ImageSeries, PdfElementData>(
        { x -> createHighlightedImages(x) },
        name: "image after", pipe)
    val pdfBlock : AccumulatorWithAlgBlock<PdfElementData, FileData> =
        AccumulatorWithAlgBlock(PdfElementsAccumulator( reportName: "report"),
            connections: 2, name: "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) }
    //run
    pipe.rootBlock = seriesReaderBlock
    pipe.run(init)
}
```

Раньше для запуска этого конвейера было необходимо писать программу на языке kotlin.

Пример конвейера: сейчас

```
[  
  { "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": [] }  
]
```

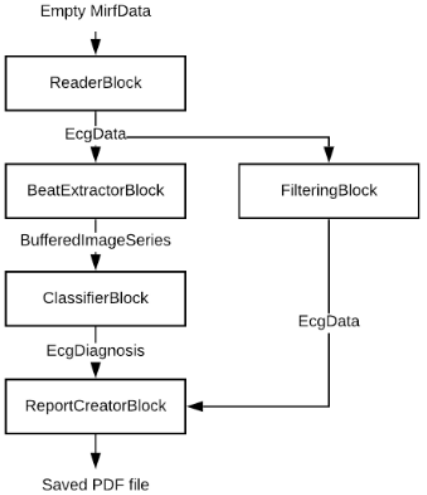
Сейчас для этого достаточно написать один json файл.


Детали реализации

- ▶ Оркестратор непрерывно следит за состоянием системы и распределяет нагрузку
- ▶ Балансировка нагрузки между блоками происходит по принципу Least Connections
- ▶ Выход из строя отдельных узлов не влияет критически на работу системы в целом
 - ▶ Блоки не имеют состояния и принимают команды только от оркестратора
 - ▶ Раз в минуту и перед отправкой любой команды оркестратор опрашивает всю сеть и обновляет статистику


Стоит отметить, что оркестратор постоянно следит за состоянием сети и распределяет нагрузку по принципу Least Connections, то есть если несколько блоков могут обработать текущий набор данных, то задание на обработку будет отправлено наименее загруженному блоку. При этом выход отдельных блоков из строя не повлияет критически на работоспособность системы в целом, поскольку оркестратор регулярно обновляет и актуализирует состояние системы.

Эксперименты: конвейер



Electrocardiogram AI-supported report 

Patient's name: John Smith
Patient's age: 56
Date of ECG recording: 2020-05-09



Conclusion: Your ECG contains mostly premature ventricular contraction beats

Для тестирования был взят конвейер, который обрабатывает результаты ЭКГ-исследований, ищет аритмии и генерирует отчет с предположительным диагнозом.

Эксперименты

- ▶ Была собрана сеть в 3 конфигурациях
- ▶ Запущена обработка серии ЭКГ-исследований, содержащих по 1893 удара
- ▶ В каждой конфигурации произведено 10 запусков по 6 исследований
- ▶ Объем данных 2.5 мегабайта
- ▶ Распараллеливание по данным

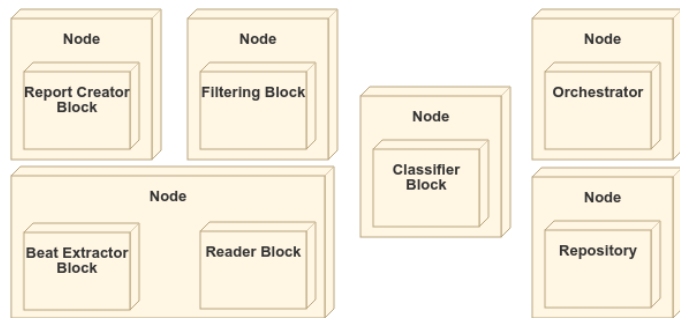
Для этого была собрана сеть в 3 конфигурациях и запущена обработка серии ЭКГ-исследований, содержащих по 1893 удара объемом около 2.5 мегабайт. В каждой конфигурации было произведено 10 запусков по 6 исследований. Распараллеливание делалось по данным.

Эксперименты: конфигурация 1

- ▶ Запуск на старой версии библиотеки (монолитная архитектура)
- ▶ Среднее время работы 840 секунд
- ▶ 85% времени занимает работа ClassifierBlock

Сначала были сделаны замеры времени работы на старой версии библиотеки с монолитной архитектурой. Среднее время обработки 6 исследований получилось равно 840 секундам, причем 85% времени занимала работа блока ClassifierBlock.

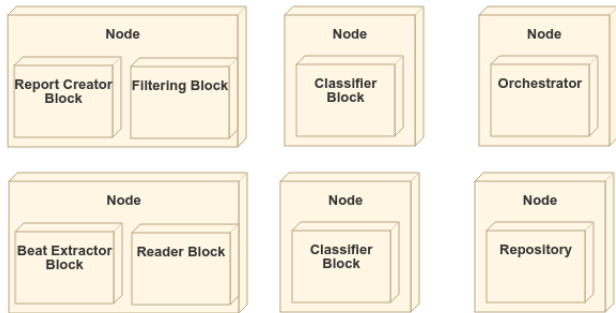
Эксперименты: конфигурация 2



После этого была развернута сеть из 6 компьютеров, представленная на слайде. Здесь Node – это физический компьютер, а внутри него находятся поднятые микросервисы. Среднее время работы в этой конфигурации составило 686 секунд.

- ▶ Node — физический узел (компьютер), внутри сервисы
- ▶ Среднее время работы 686 секунд

Эксперименты: конфигурация 3



Учитывая тот факт, что большая часть работы сосредоточена в одном блоке, мы переконфигурировали сеть: запустили сервисы ReportCreatorBlock и FilteringBlock на одном компьютере, а на освободившемся компьютере подняли ещё один ClassifierBlock. Среднее время работы в такой конфигурации составило 384 секунды.

- ▶ Среднее время работы 384 секунды

Эксперименты: результаты измерений

- ▶ E — математическое ожидание
- ▶ σ — среднеквадратическое отклонение
- ▶ Время указано в секундах

	конфигурация 1	конфигурация 2	конфигурация 3
E	840	686	384
σ	17	28	25

Таблица: результаты измерений

Результаты измерений сведены в таблицу и представлены на слайде. Таким образом видно, что при грамотном конфигурировании сети можно добиться существенного ускорения вычислений.

Результаты

- ▶ Сделан обзор предметной области
- ▶ Разработана микросервисная архитектура
- ▶ Проведен архитектурный рефакторинг
- ▶ Проведено тестирование и апробация
- ▶ Статья принята к публикации на SEIM-2020
 - ▶ MIRF 2.0 — a framework for distributed medical images analysis
- ▶ Код: <https://github.com/alexeevna/MIRF2/tree/alex>

Все поставленные задачи были успешно выполнены, а также написана и принята к публикации статья MIRF 2.0 — a framework for distributed medical images analysis.