Московский государственный университет имени М. В. Ломоносова

Факультет Вычислительной Математики и Кибернетики

Кафедра Автоматизации Систем Вычислительных Комплексов

Бодров Антон Олегович

**Построение масштабируемого сервиса сбора и обработки данных на примере медицинской телеметрии**

**Building a scalable data collection and processing service: medical telemetry case study**

Выпускная квалификационная работа

Научный руководитель: к.ф.-м.н., в.н.с. Бахмуров A.Г.

Консультант: Любицкий А.А.

Москва

2022

Аннотация

Данная работа посвящена улучшению производительности серверной части платформы для сбора и обработки физиологических данных о человеке с помощью технологий масштабирования. Проведён обзор способов виртуализации, на основании которого были выбраны метод и ПО для виртуализации. В рамках работы реализована автомасштабируемость ПО, предоставляющего возможность хранения и обработки данных мониторинга, и балансировка нагрузки между копиями серверной части платформы. Экспериментальное исследование подтвердило гипотезу об улучшении производительности платформы после применения технологий масштабирования.

Оглавление

[1 Введение 4](#_Toc103623641)

[2 Постановка задачи 7](#_Toc103623642)

[3 Обзор технологий виртуализации 8](#_Toc103623643)

[3.1 Цель обзора 8](#_Toc103623644)

[3.2 Способы виртуализации 8](#_Toc103623645)

[3.3 Обоснование выбора способа виртуализации 9](#_Toc103623646)

[3.4 Результаты обзора 12](#_Toc103623647)

[3.5 Выводы из обзора 12](#_Toc103623648)

[4 Обзор технологий оркестрации 13](#_Toc103623649)

[4.1 Цель обзора 13](#_Toc103623650)

[4.2 Способы оркестрации 13](#_Toc103623651)

[4.3 Обоснование выбора ПО для автомасштабирования 14](#_Toc103623652)

[4.4 Результаты обзора 15](#_Toc103623653)

[4.5 Выводы из обзора 15](#_Toc103623654)

[5 Исследование и решение задачи 16](#_Toc103623655)

[5.1 Исходные данные 16](#_Toc103623656)

[5.2 Обоснование схемы масштабирования 18](#_Toc103623657)

[6 Описание реализации 19](#_Toc103623658)

[6.1 Балансировка нагрузки при помощи HAproxy 19](#_Toc103623659)

[6.1.1 Общая схема работы 19](#_Toc103623660)

[6.1.2 Алгоритмы балансировки HAproxy 19](#_Toc103623661)

[6.2 Виртуализация компонент платформы 21](#_Toc103623662)

[6.3 Автоматическое масштабирование платформы. Docker swarm и orbiter 23](#_Toc103623663)

[7 Экспериментальное исследование 27](#_Toc103623664)

[8 Заключение 30](#_Toc103623665)

[9 Список литературы 31](#_Toc103623666)

# 1 Введение

В современном мире IT с каждым годом всё большее распространение получает такая сфера, как IoT или Internet of Things (Интернет вещей). Данное понятие представляет собой компьютерную сеть физических предметов («вещей»), которые могут взаимодействовать друг с другом и внешней средой. Широкое распространение IoT находит в медицине, где он получил известность, как Интернет медицинских вещей (IoMT, Internet of Medical Things). IoMT используется для мониторинга состояния человеческого организма в реальном времени, позволяет улучшить процессы лечения, восстановления пациента и их мониторинг, а также усовершенствовать процесс физических тренировок.

Проблема мониторинга состояния человека актуальна не только в медицинским учреждениях, но и в других сферах. Сбор и обработка информации о текущем функционировании организма человека в реальном времени позволяют быстро анализировать медицинские показатели и выявлять патологии, что может быть определяющим для здоровья конкретного человека при критическом состоянии пациента. Помимо выгоды для самого пациента, мониторинг может быть чрезвычайно актуальным для специалистов в области медицины, которым данная информация будет полезна, как дополнительный источник для исследования поведения организма при различных заболеваниях. При тренировках мониторинг состояния необходим для грамотного распределения нагрузки на организм.

Одной из актуальных задач является создание платформы сбора и обработки физиологических данных с открытым исходным кодом, так как несмотря на обширность рынка портативных носимых устройств большинство из них являются проприетарными проектами. Помимо этого, данные проекты являются зарубежными, что в текущих реалиях не гарантирует их стабильного к ним доступа.

На текущий момент существует ряд платформ мониторинга здоровья человека, которые широко используются на практике. В их число входят и собственные платформы производителей носимых устройств, такие как Fitbit, Omron, Apple Health и многие другие, а также такие интеграционные системы как Validic, Seqster, CareSimple, Capsule Technologies.

Все перечисленные программные продукты являются проприетарными, из них невозможно получить данные или они выдаются уже в обработанном виде. Это решения не являются бесплатными и открытыми проектами.

Существует также ряд открытых решений, однако их лицензия не позволяет развивать эти разработки, поэтому было принято решение создать собственную платформу мониторинга здоровья человека на факультете вычислительной математики и кибернетики Московского Государственного Университета.

В качестве носимого устройства в данной работе предлагается использовать умную одежду Hexoskin компании Carre Technologies Inc [1], которая способна считывать и обрабатывать показатели состояния человеческого организма. Hexoskin имеет открытый интерфейс для получения данных по Bluetooth, поэтому именно она выбрана для работы.

Ограничение на число клиентов платформы заключит её возможности в рамки, поэтому для функционирования платформы с различным числом клиентов следует обеспечить её масштабируемость. Способность увеличивать производительность системы пропорционально дополнительным ресурсам (оборудованию) может обеспечить применение современных технологий виртуализации и балансировки нагрузки.

Масштабируемость можно разделить на горизонтальную и вертикальную. Под вертикальной масштабируемостью понимается увеличение производительности каждого компонента системы с целью повышения общей производительности. Масштабируемость в этом контексте означает возможность заменять в существующей вычислительной системе компоненты более мощными и быстрыми по мере роста требований и развития технологий.

Горизонтальной масштабируемостью называют разбиение системы на более мелкие компоненты и разнесение их по отдельным машинам или увеличение количества серверов, параллельно выполняющих одну и ту же функцию. Масштабируемость в этом контексте означает возможность добавлять к системе новые узлы, серверы для увеличения общей производительности. Этот способ масштабирования может требовать внесения изменений в программы, чтобы последние могли в полной мере пользоваться возросшим количеством ресурсов. В данной работе рассмотрена именно горизонтальная масштабируемость.

Для оптимизации использования ресурсов, горизонтального масштабирования и обеспечения отказоустойчивости используется балансировка нагрузки – процесс распределения задач между несколькими серверами.

Таким образом мы сможем избавиться от ограничения на число клиентов платформы мониторинга, что позволит осуществлять мониторинг состояния значительного количества пациентов одновременно. Однако выделение ресурсов сверх меры приведёт к тому, что значительная часть выделенных мощностей будет простаивать, поэтому более удачным решением будет обеспечение масштабирования в автоматическом режиме, когда масштабирование будет производиться по мере увеличения или уменьшения количества клиентов.

Данная работа посвящена регулированию, в том числе увеличению производительности платформы для сбора и обработки медицинской телеметрической информации путём автоматического масштабирования. В рамках работы достигнуто улучшение ранее разрабатывавшейся на факультете ВМК платформы по сбору и обработки медицинских данных.

# 2 Постановка задачи

Целью данной работы является улучшение производительности и отказоустойчивости серверной части платформы для сбора и обработки физиологических данных о человеке путём применения технологий масштабирования, в том числе масштабирования в автоматическом режиме. Серверная часть представляет собой программное обеспечение, которое предоставляет интерфейс загрузки данных мониторинга состояния человека с мобильного устройства, интерфейс для получения данных, инструмент для хранения и обработки данных мониторинга, способное выдерживать различную клиентскую нагрузку.

Для достижения поставленной цели необходимо выполнить следующие задачи:

1. Спроектировать архитектуру автомасштабируемой серверной платформы.
2. По итогу обзоров выбрать алгоритм балансировки, ПО для масштабирования и балансировки нагрузки.
3. Реализовать масштабирование и балансировку сервиса сбора и обработки данных.
4. Выполнить виртуализацию всех компонент платформы
5. Реализовать масштабирование сервиса сборка и обработки данных в автоматическом режиме.
6. Провести экспериментальное исследование с целью проверки гипотезы об улучшении производительности платформы.

# 3 Обзор технологий виртуализации

## 3.1 Цель обзора

Для обеспечения переносимости платформы с одной инфраструктуры на другую и независимости её от характеристик хоста используют виртуализацию.

Целью обзора является выбор способа виртуализации для дальнейшей реализации масштабирования платформы на его основании.

## 3.2 Способы виртуализации

В настоящее время осуществлять виртуализацию можно несколькими способами. Для этого можно использовать:

1. Виртуальные машины (VM)

2. Контейнеризацию

*Виртуальные машины (VM)*

Виртуальная машина – это программная и/или аппаратная система, [эмулирующая](https://ru.wikipedia.org/wiki/%D0%AD%D0%BC%D1%83%D0%BB%D1%8F%D1%86%D0%B8%D1%8F) [аппаратное обеспечение](https://ru.wikipedia.org/wiki/%D0%90%D0%BF%D0%BF%D0%B0%D1%80%D0%B0%D1%82%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5) некоторой [платформы](https://ru.wikipedia.org/wiki/%D0%90%D0%BF%D0%BF%D0%B0%D1%80%D0%B0%D1%82%D0%BD%D0%B0%D1%8F_%D0%BF%D0%BB%D0%B0%D1%82%D1%84%D0%BE%D1%80%D0%BC%D0%B0_%D0%BA%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%B0) (называемой гостевой платформой) и исполняющая программы для гостевой платформы на платформе-хозяине, изолирующая от платформы-хозяина приложения, бинарные файлы и операционные системы. Виртуальная машина обладает всеми необходимыми устройствами и виртуальным жёстким диском, на который устанавливается гостевая операционная система, драйверы устройств и прочие компоненты. Виртуальное оборудование функционирует для гостевой ОС, как реальное, при этом виртуальная машина изолирована от реального компьютера, однако может иметь общие каталоги файловой системы c ним.

Использование VM вызывает дополнительные расходы на виртуальное оборудование, запуск гостевой ОС и поддержку необходимого окружения для работы приложений.

*Контейнеры*

Контейнеризация – это метод легковесной виртуализации, при котором вместе упаковываются и изолируются от домашней ОС приложение, бинарные файлы и библиотеки. В отличие от виртуальных машин контейнеры обеспечивают виртуализацию на уровне операционной системы, а не аппаратного обеспечения [Рис.1]. Благодаря виртуализации на уровне ОС, то есть изоляции только приложения, бинарных файлов и библиотек без изоляции ОС, возможно быстрое развёртывание и простое масштабирование. Также при этом уменьшается объём занимаемого места на жёстком диске по сравнению с виртуализацией с помощью виртуальных машин.

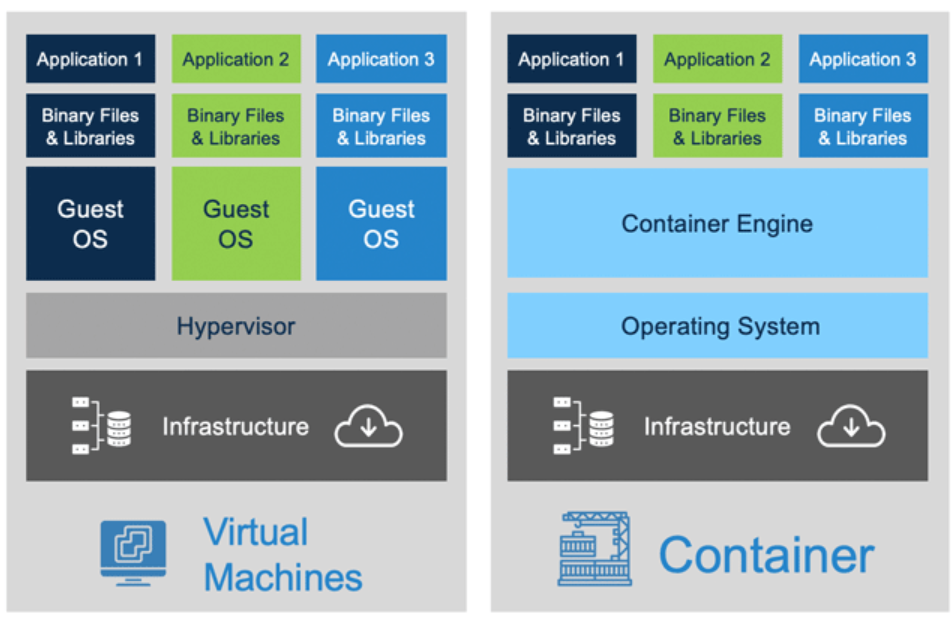


Рисунок 1 - Сравнение технологий виртуализации виртуальных машин и контейнеров [2]

## 3.3 Обоснование выбора способа виртуализации

Ниже, в таблице 1, приведено сравнение технологий виртуальных машин и контейнеризации [3]. В качестве критериев сравнения способа виртуализации отобраны:

* Уровень виртуализации;
* Вид операционной системы;
* Время начальной загрузки;
* Использование ресурсов хоста;
* Доступность предварительно построенных образов (в данной работе будут использованы предварительно построенные образы, поэтому этот критерий особенно важен);
* Возможность создания пользовательских предварительно настроенных образов (для настоящей работы должно быть удобно настраивать свои образы с целью упаковки в них компонент платформы);
* Размер образа;
* Мобильность;
* Время создания (образ не должен создаваться слишком долго);
* Производительность.

|  |  |  |
| --- | --- | --- |
|  | Виртуальные машины | Контейнеры |
| Уровень виртуализации | Аппаратный | На уровне ОС |
| Операционная система | Отдельная у каждой виртуальной машины | Общая |
| Время загрузки | Больше | Мене |
| Использование ресурсов хоста | Больше | Меньше |
| Доступность пользовательских предварительно настроенных образов | Ограниченное количество образов | Большая коллекция различных образов (например, на Dockerhub) |
| Возможность создания пользовательских предварительно настроенных образов | Трудно настроить | Легко настроить |
| Размер образа | Больше, потому что виртуализация происходит с операционной системой полностью | Меньше |
| Миграция | Легко перемещается с одного сервера на другой | Уничтожается и пересоздаётся вместо перемещения |
| Время создания | Дольше (в среднем минуты) | Меньше (в среднем секунды) |
| Производительность | Накладные расходы при передаче инструкций из гостевых ОС в хостовую ОС | Производительность, близкая к ОС хоста |

Таблица 1 - Сравнение технологий виртуальных машин и контейнеризации

Основными преимуществами контейнеров перед виртуальными машинами для данной задачи является доступность предварительно построенных образов и их лёгкое создание. Также преимуществами контейнеров перед виртуальными машинами является незначительность накладных расходов при загрузке и построении. В связи с этим в качестве основного решения выбрана контейнеризация.

Далее, в таблице ц, сравнивались различные программные средства для виртуализации. В первую очередь для простоты работы с программным средством в данной работе от ПО требуется наличие подробной документация, крупного сообщества и свободная лицензия. Как было сказано выше, безусловным преимуществом технологии контейнеризации являются маленькие накладные расходы, поэтому этот критерий так же будет важным.

Проводилось сравнение следующего ПО для виртуализации:

* OpenVZ [4];
* LXC [5];
* Docker [6];
* Virtual Box [7].

В качестве критериев сравнения отобраны:

* Размер сообщества пользователей (ПО с большим сообществом легче использовать при разработке);
* Лицензия;
* Год основания технологии (более старая технология лучше апробирована пользователями);
* Размер накладных расходов;
* Поддерживаемые ОС (ПО должно запускаться на любой из распространённых ОС);
* Подробность документации (недостаточно подробная документация может не позволить применить технологию для решения задачи выпускной квалификационной работы).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Open VZ | LXC | Docker | Virtual Box |
| Размер сообщества | маленькое | среднее | большое | большое |
| лицензия | [GNU GPLv2](https://en.wikipedia.org/wiki/GNU_General_Public_License) (принадлежит проприетарной Virtuozzo) | [GNU GPLv2](https://en.wikipedia.org/wiki/GNU_General_Public_License) | [Apache License 2.0](https://en.wikipedia.org/wiki/Apache_License) | [GNU GPLv2](https://en.wikipedia.org/wiki/GNU_General_Public_License) |
| Год основания технологии | 2005 | 2008 | 2013 | 2007 |
| Легковесная виртуализация | + | + | + | - |
| Поддерживаемые ОС | Linux | Linux | Linux, FreeBSD, Windows, macOS, | Linux, Windows, acOS, Solaris, Genode OS Framework |
| Документация | подробная | подробная | Очень  подробная | подробная |

Таблица 2 - Сравнение ПО для виртуализации

## 3.4 Результаты обзора

В качестве ПО для виртуализации выбрана технология Docker в связи с простотой в работе и доступностью документации в сравнении с другими технологиями контейнеризации. Также Docker дополнительно уменьшает занимаемый объём на жёстком диске за счёт переиспользования образов, то есть способности использовать один образ для нескольких сервисов.

## 3.5 Выводы из обзора

В данном разделе был проведён обзор четырёх программных средств для виртуализации, в результате которого для дальнейшего рассмотрения и реализации была выбрана технология контейнеризации и технология Docker.

# 4 Обзор технологий оркестрации

## 4.1 Цель обзора

Под оркестрацией будем понимать координацию взаимодействия контейнеров и автоматизацию их жизненного цикла. Целью обзора является выбор наиболее удобного способа управления контейнерами для последующей реализации автомасштабирования в контексте данной работы.

## 4.2 Способы оркестрации

Для реализации масштабирования в автоматическом режиме наиболее популярны две технологии – Kubernetes [8] и интегрированная в экосистему Docker docker swarm. Их сравнение проведено в таблице 3.

В качестве критериев сравнения отобраны:

* Интеграция с экосистемой docker (предлагаемое решение построено на технологиях docker, поэтому крайне желательно, чтобы средство было легко встраиваемым в данную технологию);
* Открытость;
* Возможность запуска на любой инфраструктуре;
* Лёгкость процесса установки;
* Возможности конфигурации;
* Простота конфигурации;
* Наличие встроенных средств автомасштабирования (будет удобнее не использовать дополнительно стороннее ПО);
* Область применения – размер проектов, в которых применяется технологий (настоящий проект на текущий момент не требует большого количество виртуальных, физических серверов или множества нод).

|  |  |  |
| --- | --- | --- |
|  | Docker swarm | k8s |
| Docker CLI | + | - |
| Открытость | Open source | Open source |
| Возможность запуска на любой инфраструктуре | + | - |
| Лёгкий установочный процесс | + | - |
| Возможности конфигурации | Ограничены | Широкие (Pods, ReplicaSets, Services и т.д.) |
| Лёгкая конфигурация | + | - |
| Встроенные средства автомасштабирования | - | + |
| Использование в отрасли | Небольшие проекты | Большие компании, требующие гибкости настройки инфраструктуры |

Таблица 3 - Сравнение технологий Docker swarm и k8s

## 4.3 Обоснование выбора ПО для автомасштабирования

Предлагаемая платформа в настоящий момент имеет визуально понятную архитектуру и сравнительно небольшое количество клиентов. Поэтому в данном случае использование docker swarm не наложит ограничения на отказоустойчивость решения. Экспериментальное исследование подтвердило, что платформа способна справляться с определённым количеством одновременно отправляющих запросы клиентов.

Интеграция с экосистемой docker существенно облегчает настройку системы, так как именно она была выбрана для виртуализации системы. При этом конфигурация самих средств оркестрации и автомасштабирования в случае использования docker swarm тоже становится проще. Сложность настройки k8s уменьшается при пользовании услуг облачных провайдеров, однако подобное решение проблемы требует дополнительных финансовых затрат. Минус использования docker swarm перед k8s в том, что у решения от Docker нет встроенных средств автомасштабирования. Однако, эту проблему возможно устранить с помощью использования сторонних программных средств. Одним из популярных средств является Orbiter [9], с помощью которого и реализовано автомасштабирование в работе.

## 4.4 Результаты обзора

В результате обзора для дальнейшего рассмотрения и реализации выбрана технология Docker swarm за счёт простоты конфигурации и отсутствия необходимости настройки сложных абстракций, требующих гибкости инфраструктуру.

## 4.5 Выводы из обзора

В данном разделе был произведён обзор двух программных средств для оркестрации и автомасштабирования, в результате которого для дальнейшего рассмотрения и программной реализации был выбран Docker swarm и Orbiter.

# 5 Исследование и решение задачи

## 5.1 Исходные данные

В рамках работы Юлии Аникевич была разработана немасштабируемая серверная платформа, которая является исходными данными для настоящей работы [10].

Общая схема работы всего приложения представлена на рисунке 2.

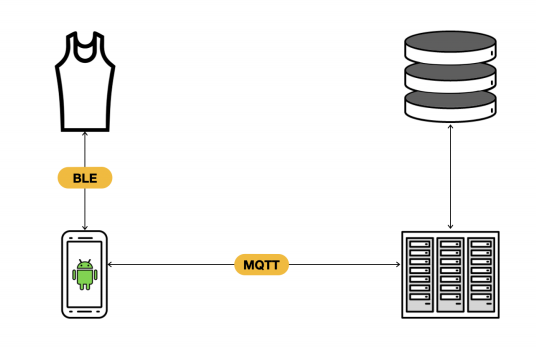


Рисунок 2 - Схема работы приложения

Умная жилетка Hexoskin соединена по протоколу Blotooth Low Energy с мобильным устройством, которое подключается через протокол MQTT к серверной части платформы. MQTT – сетевой протокол, работающий поверх TCP/IP [11]. MQTT ориентирован на обмен сообщениями между устройствами по принципу издатель-подписчик. Клиент может быть, как издателем или подписчиком, так и иметь обе роли одновременно. На рисунке 3 изображена простая схема взаимодействия между брокером и двумя клиентами. Один из клиентов является издателем и публикует данные, а другой – совершает подписку на топик, в который публикует сообщения первый клиент. Для реализации выбран брокер Eclipse Mosquitto [12]– брокер с открытым исходным кодом, реализующий различные версии протокола MQTT.

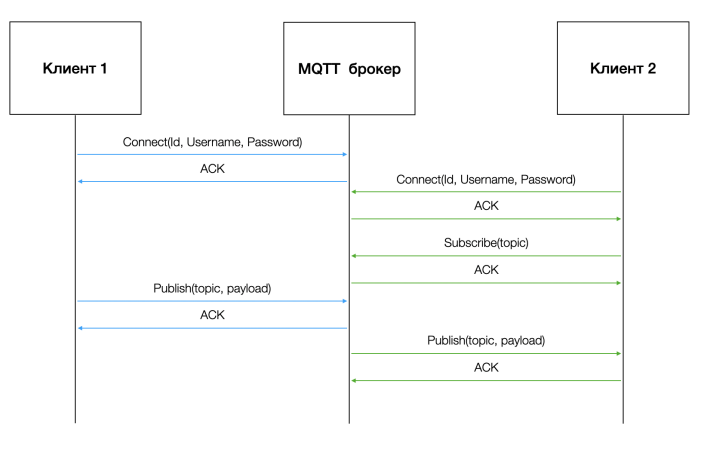


Рисунок 3 - Взаимодействие между клиентами и MQTT-брокером

Схема взаимодействия между компонентами серверной платформы указана на рисунке 4. Веб-приложение получает данные из базы данных мониторинга, роль которой исполняет Clickhouse [13].

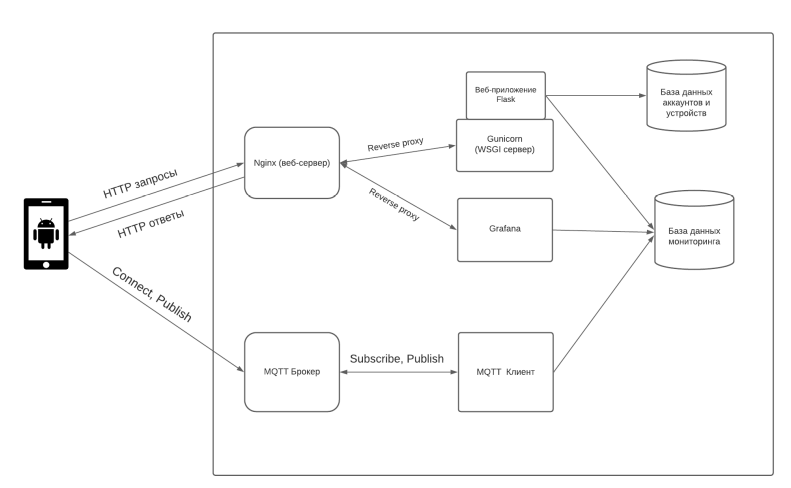


Рисунок 4 - Схема работы серверной части платформы

## 5.2 Обоснование схемы масштабирования

Основная задача работы – обеспечить масштабирование сервиса сбора данных run.py и подключенного к нему MQTT-брокера, потому что основная нагрузка системы падает именно на эту пару компонент в связи с тем, что публикация данных в Clickhouse идёт через MQTT-брокер и MQTT-клиент. Каждую копию приложения можно подключить к одному брокеру или ко всем сразу. В настоящей работе при распределении нагрузки необходимо избежать дублирования данных, иначе в базе данных Clickhouse пришедшие данные со стороны клиента будут повторяться, а избавляться от дублей непосредственно в базе данных сложнее. При этом также важно избежать потери данных.

При подключении одного брокера ко всем копиям серверной части платформы возникает проблема «узкого места» при взаимодействии клиентов с брокером. Также возникает дублирование попадающих в базу данных топиков. Если избежать этого и распределить клиентов по копиям сервисов, то возникает проблема синхронизации. При выходе из строя одной из копий серверной части приложения необходимо сообщить об этом остальным (иначе произойдёт потеря данных), а это само по себе является нетривиальной задачей.

В случае прекращения работы MQTT-брокера или MQTT-клиента возникает проблема установки соединения между ним и приложением. Если каждое приложение подписывается на каждый брокер, то также возникает проблема дублирования. Поэтому каждая копия приложения соединена с одним брокером, и каждая такая пара упакована в Docker-контейнер. Клиент устанавливает TCP-соединение через балансировщик с одним из брокеров, которые не перегружены. Балансировщик при этом так же упакован в контейнер. При выходе из строя одного из контейнеров, приложений или брокеров, TCP-соединение для соответствующим MQTT-брокером рвётся, и переустанавливается на другой работающий и свободный брокер. Docker-контейнеризация обеспечит изоляцию, благодаря которой получится избежать конфликтов библиотек. Orbiter регулирует количество контейнеров с парой MQTT-брокер и MQTT-клиент.

# 6 Описание реализации

## 6.1 Балансировка нагрузки при помощи HAproxy

6.1.1 Общая схема работы

В качестве балансировщика нагрузки был выбран HAProxy [14].

HAProxy — серверное программное обеспечение для обеспечения [высокой доступности](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%81%D0%BE%D0%BA%D0%B0%D1%8F_%D0%B4%D0%BE%D1%81%D1%82%D1%83%D0%BF%D0%BD%D0%BE%D1%81%D1%82%D1%8C) и [балансировки нагрузки](https://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%BB%D0%B0%D0%BD%D1%81%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D0%BD%D0%B0%D0%B3%D1%80%D1%83%D0%B7%D0%BA%D0%B8) посредством распределения входящих запросов на несколько обслуживающих серверов.

HAProxy зарекомендовал себя, как надёжное программное обеспечение. Программа используется в ряде высоконагруженных сайтов, включая Twitter, Instagram, GitHub, StackOverflow. HAProxy является программой с [открытым исходным кодом](https://ru.wikipedia.org/wiki/%D0%9E%D1%82%D0%BA%D1%80%D1%8B%D1%82%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5) и распространяется в соответствии с [GNU General Public License](https://ru.wikipedia.org/wiki/GNU_General_Public_License). В связи с этим она выбрана в качестве решения для поставленной задачи.

HAProxy может быть запущено в двух различных режимах – TCP и HTTP. Режимы различаются доступной функциональностью. В режиме TCP каждая публикация клиента и каждый приход топика в рамках одного TCP-соединения происходит к одной конкретной копии брокера. В режиме HTTP каждый новый топик приходит на новый брокер. В настоящей работе стоит задача балансировки самого TCP-соединения, а протокол MQTT работает поверх TCP, поэтому представлено использование HAProxy в режиме TCP.

Клиент отправляет запросы HAProxy, выступающему в качестве балансировщика, и получает ответ на каждый из них. HAProxy распространяет запросы по узлам, взаимодействуя в каждом случае с одним MQTT-брокером [15]. Узел может либо принять запрос, если он не перегружен, либо вернуть null или ошибку. Тогда HAproxy передаст запрос следующему узлу. Если ни один из узлов не сможет предоставить услугу, то HAproxy вернёт null-значение или ошибку клиенту [16].

6.1.2 Алгоритмы балансировки HAproxy

Распределение нагрузки между контейнерами идёт в соответствии с выбранным для HAproxy алгоритмом балансировки. Существует несколько таких алгоритмов:

* Round Robin

При работе алгоритма HAproxy по очереди перебирает доступные серверы (в данном случае контейнеры) и равномерно распределяет нагрузку. Количество серверов ограничено 4095.

Также есть дополнение данного алгоритма - weighted Round Robin. В случае его использования балансировщик способен распределять нагрузку в зависимости от физических возможностей сервера. Сервер с весом 2 будет получать в два раза больше запросов, чем сервер с весом 1.

* Static-rr

Static-rr представляет собой тот же Roundrobin без ограничения на количество серверов.

* Least Connections

Least Connections предназначается для длительных соединений. Он динамический, и каждый запрос передаётся на наименее нагруженный сервер.

* First

Данный алгоритм заполняет свободные серверы по порядку от первого и до последнего.

* Source

Такой алгоритм распределяет запросы от источника с одним IP всегда к одному серверу.

* URI

Алгоритм выполняет балансировку в зависимости от адреса страницы.

* HDR

Выбирается сервер на основе заголовка HTTP запроса. Если искомое значение отсутствует в заголовке, то используется принцип Round Robin.

* rdp-cookie

 Задача алгоритма - передавать запросы от одного и того же пользователя на один и тот же сервер с идентификацией по cookie.

На практике обычно используются два алгоритма – Round Robin и Least Connections. Последние четыре алгоритма имеют специфичные цели применения, поэтому в работе проведено сравнение алгоритмов Round Robin, Least Connections, First и Static-rr (см. раздел Экспериментальное исследование), в ходе сравнения для реализации был выбран Least Connections.

## 6.2 Виртуализация компонент платформы

Для беспроблемной переносимости архитектуры с одной инфраструктура на другую и удобного управления всеми компонентами платформы будет удобно провести их виртуализацию путём виртуализации всех компонент с помощью Docker.

Виртуализация компонент платформы происходит с помощью выбранной ранее технологии Docker. Dockerfile – специальный файл, состоящий из процедурных инструкций для построения Docker-контейнера [17]. Можно воспринимать Dockerfile, как исходный код для контейнеров. В настоящей работе используется несколько Dockerfile:

* для балансировщика HAproxy;
* для flask-приложения под управлением gunicorn [18] (веб-сайта платформы и nginx [19] в одном контейнере под управлением supervisord);
* для приложения run.py и mqtt-брокера под управлением superivisord.

Supervisord [20] используется в данной работе для обеспечения работы нескольких процессов в одном контейнере. Из Dockerfile запускается supervisord, как управляющий процесс, а он стартует и контролирует другие процессы в контейнере. Процессы указываются в конфигурационном файле supervisord, который выглядит следующим образом:

[program:app]

command = python3 /iomt-project/mqtt-daemon/run.py

[program:mosquitto]

command = /usr/sbin/mosquitto -c /etc/mosquitto.conf

autorestart=true

Dockerfile, например для приложения run.py и MQTT-брокера представляет собой структуру, показанную ниже:

FROM ubuntu:18.04 AS ubuntu-app

RUN sed -i'' 's/archive\.ubuntu\.com/us\.archive\.ubuntu\.com/' /etc/apt/sources.list

RUN apt-get update && apt-get -y install sudo

RUN useradd -m docker && echo "docker:docker" | chpasswd && adduser docker sudo

RUN apt-get install -y build-essential libssl-dev python3-dev rustc libffi-dev

RUN apt-get install -y musl-dev

RUN apt-get install -y python3-pip

COPY requirements.txt /tmp/requirements.txt

RUN pip3 install --no-cache-dir -r /tmp/requirements.txt

ADD http://repo.mosquitto.org/debian/mosquitto-repo.gpg.key /root/

ADD http://repo.mosquitto.org/debian/mosquitto-jessie.list /etc/apt/sources.list.d/

RUN apt-key add /root/mosquitto-repo.gpg.key

RUN apt-get update && apt-get -y install mosquitto supervisor

RUN mkdir -p /var/log/supervisor

COPY web/supervisord.conf /etc/supervisor/conf.d/supervisord.conf

COPY mqtt-daemon /iomt-project/mqtt-daemon

COPY mqtt-daemon/mosquitto.conf /etc/mosquitto.conf

RUN apt-get update --fix-missing && apt-get install -y git vim

RUN git clone https://github.com/wiomoc/mosquitto-jwt-auth.git && \

    cd mosquitto-jwt-auth && \

    cargo build --release

RUN find / -name "libmosquitto\_jwt\_auth.so"

RUN cp /mosquitto-jwt-auth/target/release/libmosquitto\_jwt\_auth.so /etc/mosquitto/libmosquitto\_jwt\_auth.so

COPY .env .env

EXPOSE 1883

CMD ["supervisord", "-n", "-c", "/etc/supervisor/supervisord.conf"]

## 6.3 Автоматическое масштабирование платформы. Docker swarm и orbiter

Docker Swarm – это средство для управления кластерами и контейнерами, которое непосредственно интегрировано в Docker Engine. Также данное средство известно, как Swarm-kits.

При создании сервиса указываются его параметры: количество копий, доступ к сети, открытые порты и сервисы, доступные извне.

Преимущество архитектуры docker swarm перед отдельно работающими контейнерами в том, что при прекращении работы одной из нод, её задачи перекладываются на другую, а не теряются.

Ещё одним преимуществом использования swarm-режима является то, что при изменении конфигурации контейнеров отсутствует необходимость перезапускать их вручную. Docker сам обновит конфигурацию и перезапустит контейнеры с новыми параметрами.

Кластер состоит из docker-хостов, которые называются нодами: каждая нода может быть управляющей (manager) или подчиняться управляющей (worker) [Рис. 5].

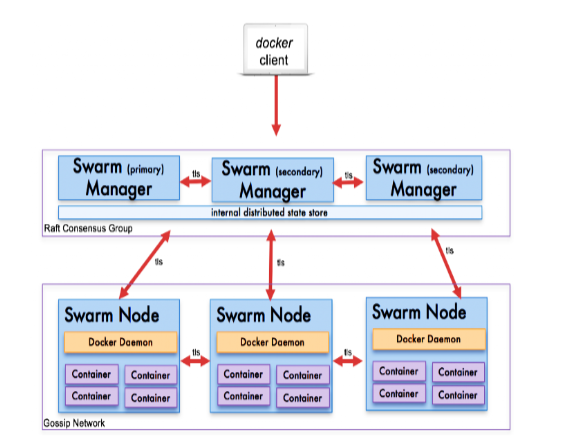


Рисунок 5 - Структура кластера docker swarm [21]

Ноды можно запускать на одном физическом или виртуальном хосте или же распределять их по нескольким. Чтобы развернуть приложением на нодах всегда нужна управляющая нода (manager).

Сервисом называются задачи, которые назначены какой-то из нод. Сервис – ключевое понятие экосистемы swarm и основной источник взаимодействия пользователя с системой.

Workers получают и исполняют запросы от менеджера. По умолчанию менеджер тоже может запускать сервисы на себе, но можно сконфигурировать его так, чтобы выполнение сервисов было возможно только на управляемых нодах. Workers уведомляют менеджера о текущем состоянии данных им заданий.

По умолчанию запуск контейнеров в swarm-режиме недоступен, чтобы запустить swarm-режим достаточно проинициализировать swarm на одну текущую ноду:

docker swarm init

Затем необходимо создать две сети: одну – для мониторинга, вторую – для непосредственной работы приложения:

docker network create -d overlay monitoring

docker network create -d overlay iomt

После выполнения данной команды текущая нода является менеджером. Далее можно присоединять новые ноды (как managers, так и workers) к созданному swarm.

Для настройки контейнеров будем использовать стандартное средство Docker docker-compose. Для этого опишем контейнеры Orbiter, iomt\_01 (контейнер с сервисом run.py и mqtt-брокером), web-site (контейнер с nginx и gunicorn) и haproxy. Все контейнеры, кроме Orbiter, используют свой Dockerfile для построения пользовательского образа. В swarm-режиме недоступно построение образов «налету», поэтому необходимо создать хранилище, куда и откуда будут загружаться построенные образы. Сделать это можно следующей командой:

docker service create --name registry --publish published=5000,target=5000 registry:2

Загрузить образы в хранилище можно следующей командой:

docker-compose push

Затем, следующая команда непосредственно разворачивает ноду, используя загруженные в созданный выше репозиторий образы и публичные образы с Dockerhub, и сервисы на ней:

docker stack deploy --compose-file docker-compose.yml iomt

Чтобы провести масштабирование в автоматическом режиме, нужно настроить систему мониторинга, для которой была создана сеть monitoring. Система мониторинга позволит контролировать загруженность сервиса сбора и обработки данных через специальные метрики. В настоящей работе будем анализировать количество запросов, проходящих через HAproxy, так как данная метрика напрямую коррелирует с загруженностью непосредственно серверов. В зависимости от величины этой метрики будет производиться масштабирование.

Контейнер с Orbiter увеличивает или уменьшает количество контейнеров с MQTT-брокером и MQTT-клиентом после запросов, которые ему отсылает Alertmanager [22], программное средство, которое обрабатывает предупреждения по изменяющимся системным параметрам. Alertmanager следит за метриками на платформе мониторинга нашей системы. В качестве такой платформы использован Prometheus [23]. Он получает системные метрики балансировщика HAproxy через Haproxy-exporter [24], специальное ПО для экспорта статистики и системных метрик HAproxy в Prometheus.

Таким образом, автомасштабируемая платформа, представленная в настоящей работе, имеет архитектуру, представленную на рисунке 6.

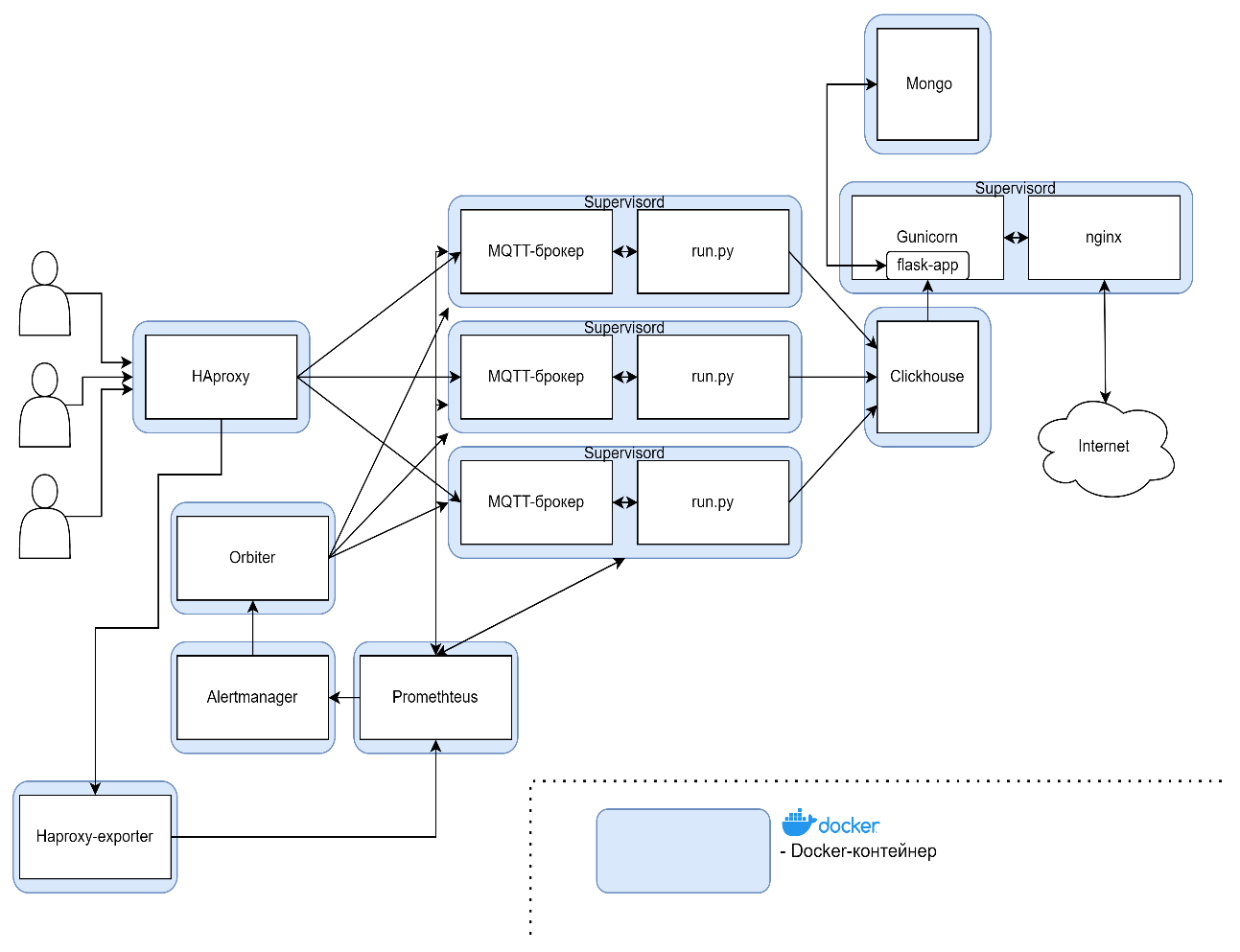


Рисунок 6 - Архитектура автомасштабируемой серверной платформы для сбора и обработки данных

# 7 Экспериментальное исследование

В качестве экспериментального исследования проведено нагрузочное тестирование с помощью утилиты mqtt-benchmark [25]. Цель тестирования – замерить среднюю скорость доставки запросов от клиента до приложения в зависимости от количества контейнеров. В экспериментах участвует масштабируемая версия платформы. Запросы отправляются через haproxy на одну из копий MQTT-брокера.

Параметр QoS (Quality of Service) обозначает соглашение между отправителем и получателем, которое гарантирует доставку сообщений [26]. Выделяют три уровня гарантии доставки сообщений QoS:

* Максимум один раз (0);
* Хотя бы один раз (1);
* Ровно один раз (2).

По умолчанию запуск происходит с QoS=1. В проводимом эксперименте мы так же будем запускать тестирование с таким параметром (QoS=1 по умолчанию).

Нагрузочное тестирование выполняется посредством команд следующего вида:

mqtt-benchmark --broker tcp://172.30.7.214:1344 --count 20 --clients 200 --wait 10 --topic c/test --format text --ramp-up-time 5

Здесь

--count - количество запросов от каждого из клиентов (что обозначает, по сути, длительность эксперимента);

--clients - количество клиентов, запущенных параллельно и, соответственно, параллельно отправляющих запросы;

--wait – время ожидания подтверждения прихода сообщения (мс);

--topic – топик для отправляемых сообщений;

--ramp-up-time – таймаут на создание клиентов (с).

Сообщения от каждого клиента по умолчанию отправляются с интервалом в 1 секунду.

В качестве измеряемых величин используем количество дошедших сообщений до MQTT-брокеров в течение времени, равному wait, и среднее время доставки сообщений до них. Будем изменять количество контейнеров и параллельно подключённых клиентов (--clients) и наблюдать, как изменяются заданные нами метрики в зависимости от данных величин.

В результате экспериментального исследования были получены результаты, изображённые на рисунках 7 и 8. Как видно из диаграмм, гипотеза об увеличении доли дошедших сообщений и уменьшении среднего времени доставки сообщения с увеличением числа контейнеров для разработанного решения поставленной задачи, подтвердилась.

Рисунок 7 - Доля дошедших до MQTT-брокера сообщений

Процент дошедших сообщений ни в одном из случаев не достиг 100%, так как мы использовали 10 мс в качестве времени ожидания подтверждения доставки сообщения, в то время, как индустриальным стандартом принято считать 100 мс. Это было сделано для наглядности разницы при работе различного количества контейнеров.

Рисунок 8 - Среднее время доставки сообщений до MQTT-брокера

Также было проведено сравнение производительности алгоритмов с помощью той же утилиты на большом числе параллельно подключенных клиентов. При 800 параллельных клиентах и использовании 2 контейнеров получены результаты, представленные в таблице 4. Алгоритм Least connections показал лучшее время доставки сообщений. Также другие работы показывают, что для решения поставленной задачи данный алгоритм лучше справится с нагрузкой [27]. Таким образом, это показывает эффективность реализованного решения на основе алгоритма Least connections.

|  |  |
| --- | --- |
| Алгоритм | Время доставки сообщений, мс |
| Round Robin | 2,017 |
| Least connections | 2,005 |
| Static-rr | 2,025 |
| First | 2,544 |

Таблица 4 - Сравнение времени доставки сообщений разных алгоритмов балансировки HAproxy

# 8 Заключение

В ходе исследования была улучшена производительность открытой платформы для сбора и обработки физиологических данных о человеке путём применения технологий масштабирования.

В рамках настоящей работы была спроектирована и разработана автомасштабируемая платформа для сбора и обработки медицинской телеметрии, в том числе реализована виртуализация всех компонентов платформы, масштабируемость сервиса сбора данных и его масштабируемость в автоматическом режиме. Проведено экспериментальное исследование, подтверждающее улучшение характеристик производительности платформы при применении технологий масштабирования и балансировки нагрузки. По результатам дипломной работы опубликована статья.

# 9 Список литературы

1. Hexoskin [Электронный ресурс]. – режим доступа:

<https://www.hexoskin.com/> (дата обращения: 20.09.2021).

1. Eric Arrington “What’s the difference between VMs & containers?” Akfpartners [Электронный ресурс] – режим доступа

<https://akfpartners.com/growth-blog/vms-vs-containers> (дата обращения: 25.09.2021).

1. A.M.Potdar, D.G.Narayan, S.Kengond, and M.M.Mulla, “Performance Evaluation of Docker Container and Virtual Machine,” Procedia Comput. Sci., vol. 171, no. 2019, pp. 1419–1428, 2020, doi: 10.1016/j.procs.2020.04.152.
2. OpenVZ Documentation. [Электронный ресурс]. - режим доступа:

<https://docs.virtuozzo.com/master/> (дата обращения: 11.11.2021).

1. LXC Documentation. [Электронный ресурс]. - режим доступа:

<https://linuxcontainers.org/ru/lxc/documentation/> (дата обращения: 11.11.2021).

1. Docker Documentation. [Электронный ресурс]. - режим доступа:

<https://docs.docker.com/> (дата обращения: 12.11.2021).

1. Virtual Box Documentation. [Электронный ресурс]. - режим доступа:

<https://www.virtualbox.org/wiki/Documentation (дата обращения: 11.11.2021).>

1. Kubernetes Documentation. [Электронный ресурс]. – режим доступа:

[https://kubernetes.io/ru/docs](https://kubernetes.io/ru/docs%20) (дата обращения: 20.09.2021).

1. Orbiter GitHub. [Электронный ресурс]. – режим доступа:

<https://github.com/gianarb/orbiter> (дата обращения: 30.12.2021).

1. Ю.В. Аникевич “ Разработка и реализация серверной части платформы для сбора и обработки медицинской телеметрии”, Дипломная работа, 2021. [Электронный ресурс]. – режим доступа:

<https://drive.google.com/file/d/1cdTAQjV7uynd5N90CJDL1wrhwguX1cQ0/view?usp=sharing> (дата обращения: 02.09.2021)

1. MQTT Version 3.1.1 documentation. [Электронный ресурс]. - режим доступа:

[MQTT Version 3.1.1 (oasis-open.org) (дата обращения: 09.01.2022).](http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf)

1. Eclipse Mosquitto [Электронный ресурс] – режим доступа:

<https://mosquitto.org/documentation/> (дата обращения: 10.01.2022).

1. Clickhouse Documentation. [Электронный ресурс]. – режим доступа:

<https://clickhouse.com/docs/ru/> (дата обращения: 20.12.2021).

1. HAproxy Documentation. [Электронный ресурс]. - режим доступа:

<http://docs.haproxy.org/> (дата обращения: 15.09.2021).

1. “How to build an high availability MQTT cluster for the Internet of Things”, Medium. - [Электронный ресурс] – режим доступа:

[https://medium.com/@lelylan/how-to-build-an-high-availability-mqtt-cluster-for-the-internet-of-things-8011a06bd000](https://medium.com/@lelylan/how-to-build-an-high-availability-mqtt-cluster-for-the-internet-of-things-8011a06bd000%20) (дата обращения: 25.02.2022).

1. Marischa Elveny, Ari Winata, Baihaqi Siregar and Rahmad Syah “A tutorial: Load balancers in a container technology system using docker swarms on a single board computer cluster”, EEO, vol. 19, issue 4, no. 2020, pp.744-751, doi: [10.17051/ilkonline.2020.04.178](http://dx.doi.org/10.17051/ilkonline.2020.04.178)
2. Hideaki Azuma, Shinsuke Matsumoto, Yasutaka Kamei, Shinji Kusumoto “An empirical study on self-admitted technical debt in Dockerfiles”, Empirical Software Engineering, vol. 27, issue 49, no. 2022, doi: [10.1007/s10664-021-10081-7](http://dx.doi.org/10.1007/s10664-021-10081-7)
3. Gunicorn Documentation. – [Электронный ресурс] – режим доступа:

[https://gunicorn.org/#docs](https://gunicorn.org/%23docs) (дата обращения: 26.03.2021).

1. Nginx Documentation. – [Электронный ресурс] – режим доступа:

<https://nginx.org/ru/docs/> (дата обращения: 27.03.2021).

1. Supervisord Documentation. – [Электронный ресурс] – режим доступа:

<http://supervisord.org/> (дата обращения: 05.09.2021).

1. T. Yudi Hadiwandra, Feri Candra “High Availability Server Using Raspberry Pi 4 Cluster and Docker Swarm”, IT JOURNAL RESEARCH AND DEVELOPMENT, 2021, vol. 6. Issue 1, pp.43-51, doi: 10.25299/itjrd.2021.vol6(1).5806
2. Alertmanager Documentation. – [Электронный ресурс] – режим доступа:

<https://prometheus.io/docs/alerting/latest/alertmanager/> (дата обращения: 01.05.2022).

1. Prometheus Documentation. – [Электронный ресурс] – режим доступа:

<https://prometheus.io/docs/introduction/overview/> (дата обращения: 30.04.2022).

1. Haproxy-exporter Github. – [Электронный ресурс] – режим доступа:

<https://github.com/prometheus/haproxy_exporter> (дата обращения: 06.05.2022).

1. MQTT-benchmark Github. – [Электронный ресурс] – режим доступа:

<https://github.com/krylovsk/mqtt-benchmark> (дата обращения: 02.04.2022).

1. “Quality of Service (QoS) 0,1, & 2 MQTT Essentials: Part 6”, hivemq. – [Электронный ресурс] – режим доступа:

<https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/> (дата обращения: 27.03.2021).

1. Agung Prasetijo, Eko Didik Widianto, Ersya T. Hidayatullah “Performance comparisons of web server load balancing algorithms on HAProxy and Heartbeat”, 2016, 2016 3rd International Conference on Information Technology, Computer, and Electrical Engineering (ICITACEE), doi: 10.1109/ICITACEE.2016.7892478