Аннотация

В данной работе была рассмотрена разработка сетевого сервиса, для картографирования и локализации роботов. Сервис расширяет возможности бортового компьютера робота за счет взаимодействия с сетевым сервисом.

Основная задача работы: создать и продемонстрировать работу простейшего облачного сервиса, с возможностью удаленной настройки, наличием графического интерфейса пользователя и удобным получением информации с данного сервиса в формате данных ROS2 топика. В процессе разработки были рассмотрены:

- Программные продукты, реализующие методы одновременного картографирования и локализации.
- Технологии дистрибуции сообщений, в частности FastDDS Discovery Server.
- Технологии контейнеризации, в частности Docker.
- Реализация графического интерфейса с использованием программной платформы Qt5.
- ROS2 экосистема и ее возможности для создания вычислительной сети.

Все алгоритмы в данной работе реализованны на языке C++, конфигурирование ROS2 файлов запуска пакетов на языке Python, создание микросервисов на языке Dockerfile.

Оглавление

Введение

Актуальность темы

Сетевые технологии и сервисы широко используются для обеспечени многих критически важных инфраструктур в современном мире. Жизнь человека в 21 веке тесно связана с ними, у большинства людей при себе постоянно находится полная по Тьюрингу вычислительная машина с выходом в глобальную сеть. Среди большого числа технологий и устройств, есть те, которые переросли единичное употребление и стали вектором развития для целой серии других открытий. Одна из подобных технологий, Internet of Things (IoT)[1][2], послужила основой для данной работы.

Термин IoT, или Интернет вещей, обозначает некоторую инфраструктуру созданную между устройствами и сетевыми сервисами, часто принимающую вид некой вычислительной сети. Благодаря появлению недорогих компьютерных компонентов, оптоволоконных технологий с высокой пропускной способностью и других сетевых устройств, технология IoT стала набирать популярность. Появилось большое множество устройств, подключенных к Интернету и использующих возможности обработки данных сетевыми сервисами. Для компаний и людей, рынок умных товаров координально поменялся. Устройства, которыми люди пользуются повседневно, такие как зубные щетки, пылесосы, автомобили и автоматические установки, перестали нуждаться в громздких программах и вычислителях на борту. Для обеспечения собственной функциональности стало достаточно иметь необходимые датчики, доступ к глобальной сети и несколько мощных серверов для корректной работы устройств.

В связи с развитием технологии, многие робототехнические компании заинтересовались применением IoT в робототехнике. В результате, синергия классической робототехники и технологии IoT показала хорошие показатели. Дальнейшее изучение вопроса, создало новое направление в робототехнике - Облачная робототехника[3][4].

Облачная робототехника - передний край исследований в области робототехники. В разработке Программного Обеспечения (ПО) по данной теме есть как устаявшиеся практики, так и пространство для новых. В данной работе, будет исследованы существующие решения и создано собственное Программное Обеспечение (ПО), поддерживающее технологию IoT.

В ходе предварительного анализа источников, был проведен сравнительный анализ роботов, использующих стандартную схему взаимодействия программных

компонентов (на одном вычислителе) и роботов, использующих клиент/серверную или подобную архитектуру, для разделения задач между программными компонентами. В результате, роботы использующие разделение задач показывали значительный выйгрыш по ключевым показателям:

- Загруженность CPU/GPU на бортовом компьютере робота. Необходимость объемных вычислений, как например, в алгоритмах одновременного картографирования и локализации, заставляет иметь мощный вычислитель, который повышает себестоимость робота и уменьшает время работы без подзарядки.

- Потребление заряда батареи. Роботы, использующие собственные вычислители потребляют больший ток на обеспечение их работы, к тому же, наличие батареи сильно увеличивает вес робота. В некоторых случаях больший вес, заставляет делать механику платформы в разы более сложной, что также требует дополнительных расходов как на инженерные работы, так и на конструкторские.

- Стоимость. За разработку сетевого сервиса и создание инфраструктуры из серверов платиться единоразовая плата, в то время как создание серии роботов, с мощными вычислителями на борту, заставляет платить за вычислитель каждый раз.

Выбор темы диссертации связан с актуальностью направления облачной робототехники, которая предлагает новые вызовые инженерам и программистам по всему миру. Также, тема связана с работами в области инфраструктурных задач в робототехнике, которые относятся к одному из направлений в области облачной робототехники.

Цели и задачи

Цель данной работы - разработка и реализация сетевого сервиса, который обеспечивал бы эффективное управление и запуск алгоритмов одновременного картографирования и локализации в контейнеризированной среде с использованием ROS2[27] и Docker[5][6], а также создания графического пользовательского интерфейса для взаимодействия с данным сервисом на основе фреймворка Qt5[7][8].
Среди задач данного исследования можно выделить следующие:

1. Анализ существующих решений. Изучение современных методов использующихся при разрабоке сетевых сервисов, графических интерфейсов пользователя, а также рассмотрение технологий для обеспечения сетевого взаимодействия и контейнеризации.

2. Проектирование архитектуры системы. Выработка подхода для создания сетевого сервиса, графического интерфейса пользователя, создания сетевого взаимодействия, обеспечение платформенной независимости.

3. Реализация сетевого взаимодействия.

4. Реализация графического интерфейса пользователя.

5. Тестирование сетевого сервиса. Проверка корректности работы, производительности и оценка эффективности взаимодействия.

6. Анализ результатов. Проанализировать результаты тестирования, выявить преимущества и недостатки, обозначить вектор для дальнейших исследований и разработок.

Объект и предмет исследования

Объектом исследования выступает информационно-техническая система - сетевой сервис, для управления алгоритмами одновременного картографирования и локализации в контейнеризированной среде.

Предметом исследований является методы и средства необходимые для разработки сетевого сервиса, обеспечивающего управление алгоритмами одновременного картографирования и локализации в контейнеризированной среде с использованием технологий ROS2 и Docker.

Методы исследования

Необходимо также указать основные методы исследования, которые были использованы в работе, среди них можно выделить теоретические и практические. Теоретические методы:

1. Анализ и синтез. Данные методы используются для изучения существующих алгоритмов одновременного картографирования и локализации, сетевых протоколов, архитектурных решений и технологий, таких как ROS2, Docker, Qt5, Boost.Asio[9][10] и т.п. Анализ позволяет разложить сложные системы на составляющие, а синтез - объединить полученные знания для разработки собственной системы.

2. Моделирование. Создание моделей, позволяет отразить структуру и возможные взаимодействия системы до ее реализации.

3. Классификация и систематизация. Упорядочивание информации способствует более глубокому пониманию предметной области.

В свою очередь эмпирические методы:

1. Эксперемент. Проведение эксперемента поможет оценить успех выполненной работы.

2. Наблюдение. Слежка за состоянием системы в различных сценарих работы является действенным способом для выявления возможных сбоев и анализа отклика системы.

3. Измерение. Количественная оценка производительности системы, таких как время отклика, потребление ресурсов вычислительной машины, помогает сделать выводы о ценности данной системы для других.

4. Сравнительный анализ - позволяет оценить преимущества и недостатки данного решения от подобных, выявить возможные пути для улучшения и подтвердить или опровергнуть эффективность предложенного подхода.

Содержание и структура работы

В данной работе реализован функционал сетевого сервиса для управления ROS2 узлами на удаленных серверах, применительно к задачам картографирования и локализации.

Анализ существующих решений. Современные архитектурные подходы к созданию сетевых сервисов, такие как микросервисная архитектура[11][12], сервис-ориетированная архитектура (SOA)[13][14], бессервисная архитектура[15][16][17] и архитектура, управляемая событиями (EDA)[18], предлагают решения для обеспечения масштабируемости, гибкости о эффективности систем на их основе. Необходимо принять решение о выборе будущей архитектуры на основе сильных и слабых сторон существующих архитектур. Также существует возможность использования комбинации различных архитектур для обеспечения требуемых результатов. Наиболее распространенной архитектурой, на данный момент, является микросервисная архитектура, сочитающая в себе удобство масштабирования и гибкости, однако имеющая повышеные накладные расходы на запуск и поддержание микросервиса в случае большого количества одновременно запущенных сервисов. Важным моментом в создании решения является не только создание архитектуры сетевого сервиса, но и вопрос сетевого взаимодействия. Существует несколько основных протоколов[19][20] и технологий, использующих их для работы. Среди основных протоколов можно выделить протоколы:

- HTTP - протокол, прикладного уровня, позволяющий работать со сложными программными сущностями (JSON, HTML, XML файлы и т.д.), обеспечивать шифрование и т.п.

- WebSocket - проткол, позволяющий обеспечить постоянное двустороннее соединение между клиентом и сервисом.

Немаловажной частью является рассмотрение существующих решений в области построения графических пользовательских интерфейсов[21][22], где несомненным лидером является фреймворк Qt, позволяющий строить эффективные, кросплатформенные интерфейсы на своей основе.

Проектирование архитектуры системы. Сетевой сервис должен обладать легким и понятным графическим пользовательским интерфейсом, эффективной работой с сетью и траффиком, а также быстро обрабатывать запросы пользователя. Для создания подобного решения необходимо использовать язык программирования использования компилируемого языка программирования[23][24], который эффективно работает с памятью и процессором вычислительной машины за счет низкой абстракции его команд, таким языком программирования является C++[25][26]. Многие крупные компании используют этот язык для написания критически важных секций, т.к. данный язык имеет одну из самых близких

к памяти и процессору их абстракций. Основной архитектурой решения, будет выступать микросервисная архитектура с некоторыми дополнениями из клиент-серверной архитектуры, в качестве сервисов будут выступать вычислительные машиные (далее сервера), которые будут запускать собственную серверную часть решения. В основе сетевого взаимодействия лежит асинхронные задачи связи созданные с помощью библиотеки Boost.Asio, позволяющей создавать TCP/UPD подключение между машинами в сети. Библиотека Boost в целом имеет нативную интеграцию с языком C++, множество решений из этой библиотеки были приняты коммитетом по стандартизации языка C++, поддержка этих решений были включены в новые стандарты C++. Boost.Asio библиотека, которая является дефакто стандартом для создания низкоуровневых абстракции сетевого взаимодействия является отличным решением для создания инфраструктуры сетевого сервиса, однако, так как прямое соединение клиента и микросервисов напрямую возможно лишь в том случае, если микросервисы обладают так называемыми белыми ip адресами, то для тестирования решения будет использоваться локальный интерфейс связи, так называемый loopback, который позволяет тестировать решения не прибегая к созданию серверов с белыми ip адресами. Механизм связи при этом никак не изменится, потому что интерфейс loopback создает подобие локальной сети на компьютере и вполне может использоваться для создания условий в которых должен работать сетевой сервис. Проектирование графического интерфейса пользователя позволяет создать удобный механизм для людей, уровня пользователей ПК, который даст им возможность пользоватся сетевым сервисом не прибегая к изучению командной оболочки и документации к CLI для взаимодействия. Для создания графического интерфейса лучшим выбором будет использования экосистемы Qt5, которая позволяет создавать удобные приложения на своей основе. Работу серверная сторона будет вести над группой контейнеров, которые способны абстрагировать на заданом уровне процессы запущенные в них. Лучше всего для этого подходит решение от компании Docker.inc, с одноименным названием. Контейнера смогут разделить задачи одновременного картографирования и локализации по процессам в контейнерах, а связь между ними обеспечат механизмы взаимодействия ROS2 платформы. Таким образом будет создана гибкая, масштабируемая система из микросервисов, которые будут способны поддерживать выполнение алгоритмов одновременного картографирования и локализации.

Структура работы

Работа состоит из введения, четырех глав и заключения.

Первая глава носит вводный характер и содержит основные понятия, методики работы, обозначения, механизмы взаимодействия и анализ существующих решений, которые позволяют создать необходимую инфраструктуру для сетевого сервиса и некоторые архитектурные моменты.

Вторая глава посвящена архитектуре сетевого сервиса для алгоритмов одновременного картографирования и локализации. В ней раскрыты моменты анализа к требованиям системы, архитектура системы и проектирование графического

интерфейса пользователя.

Третья глава посвящена деталям реализации сетевого сервиса для алгоритмов одновременного картографирования и локализации, показаны ключевые моменты реализации, возможности библиотек, применительно к созданию и примеры использования.

Четвертая глава посвящена тестированию и оценке эффективности. Проведено сравнение с существующими решениями и приведены результаты тестирования.

Обзор литературы

Был проведен поиск публикаций по теме. Далее будет перечислен список статей:

- Supun Kamburugamuve, Leif Christiansen, Geoffrey Fox. - A Framework for Real Time Processing of Sensor Data in Cloud. - Indiana University:April 2015. - 12 p.
- Supun Kamburugamuve, Hegjing He, Geoffrey Fox. - Cloud-based Parallel Implementation of SLAM for Mobile Robots. - Indiana University:December 2017. - 8 p.
- Chanda Monga, Minakshi Sharma, Sakshi Kalra. - Cloud Robotics. - Ferozepur City:March 2017. - 5 p.
- Rajesh Doriya, Paresh Payal, Vibhav Anand, Pavan Chakraborty. - A review on cloud robotics based frameworks to solve simultaneous localization and mapping (slam) problem. - National Institute of Technology:February 2015. - 6 p.
- Weinan Chen, Shilang Chen, Jiewu Leng, Jiankun Wang. - A Review of Cloud-Edge SLAM: Toward Asynchronous Collaboration and Implicit Representation Transmission. - IEEE Transactions on Intelligent Transportation Systems:November 2024. - 17 p.
- Pengfei Zhang, Huaimin Wang, Bo Ding, Suning Shang. - Cloud-Based Framework for Scalable and Real-Time Multi-Robot Slam. - IEEE International Conference on Web Services (ICWS):July 2018. - 8 p.
- Supun Kamburugamuve, Hengjing He, Geoffrey Fox, David Crandall. - Cloud-based Parallel Implementation of SLAM for Mobile Robots. - ICC16:Proceedings of the International Conference on Internet of things and Cloud Computing. - 8 p.
- Taizhi Lv, Juan Zhang, Yong Chen. - A SLAM Algorithm Based on Edge-Cloud Collaborative Computing. - Hindawi Journal of Sensors:April 2022. - 17 p.
- Yanli Liu, Heng Zhang, Chao Huang. - A Novel RGB-D SLAM Algorithm Based on Cloud Robotics. - Microsoft Kinect Sensors: Innovative Solutions, Applications, and Validations:2019. - 28 p.

В статье "Cloud Robotics: A Survey"авторы предоставляют обзор технологий облачной робототехники. Они выделяют следующие основные преимущества облачной робототехники:

- Масштабируемость: Облачная робототехника позволяет легко масштабировать роботизированные операции, добавляя или удаляя роботов по мере необходимости.
- Снижение затрат: Облачная робототехника может снизить затраты на роботизированные операции, поскольку предприятия могут использовать инфраструктуру и ресурсы облака вместо того, чтобы приобретать и поддерживать собственную инфраструктуру.
- Улучшенная безопасность: Облачная робототехника может улучшить безопасность роботизированных операций, поскольку данные и код могут быть централизованы и защищены в облаке.

Авторы также выделяют следующие основные недостатки облачной робототехники:

- Задержка: использование решений из области облачной робототехники могут привести к задержкам в роботизированных операциях, поскольку данные и команды должны передаваться через сеть.
- Безопасность: решения из области облачной робототехники могут представлять новые угрозы безопасности, поскольку зачастую данные предаются в незашифрованном виде, либо зашифрованными недостаточно.
- Сложность: решения из области облачной робототехники могут быть сложными для внедрения и использования, поскольку предприятия должны интегрировать в свои кодовые базы возможности взаимодействия систем робота с облачными сервисами.

Эти статьи описывают типичные подходы для решения задачи одновременного картографирования и локализации в облаке.

Глава 1

Анализ предметной области

1.1 ROS2

Robot Operating System 2 (ROS 2)[27][28] — это современная, модульная и масштабируемая платформа для разработки программного обеспечения в области робототехники и не только. Цифра 2 в названии означает версию данной платформы, данная версия пришла на смену традиционному ROS первой версии, устраняя его недостатки, в частности, недостаток настройки взаимодействия узлов в системе, однако ROS первой версии и ROS2 имеют много общего. Так, для конечного пользователя - программиста, ничего при роботе сильно не поменялось, только если некоторые инструменты сменили название и стали более эффективными, поэтому в дальнейшем, названия ROS и ROS2 будут взаимозаменяемыми, в случае, если это не влияет на правильность повествования. ROS, был создан в 2007 году в Лаборатории Искусственного Интелекта Стэнфордского Университета и с тех пор стала одной из самых популярных платформ для созданиия Программного Обеспечения (ПО) в области робототехники. Данная платформа, активно поддерживается сторонниками открытого программного обеспечения, которые быстро исправляют различные ошибки обнаруженные при работе с ROS и дополняют экосистему новыми инструментами.
ROS, как и ROS2, предоставляет набор инструментов и библиотек для разработки Программного Обеспечения (ПО) для целей робототехники. ROS2 использует микросервисную архитектуру, которая хорошо подходит для создания многокомпонентой, быстро изменяющейся системы. Одним из главных достоинств данной платформы для целей робототехники, является ее, относительно низкий, порог входа, от момента начала изучения до момента написания первого ROS2 узла проходит немного времени, т.к. ROS реализует многие абстракции самостоятельно, например, предоставляя реализацию сериализации данных с датчиков.
Основные возможности ROS 2, которые помогают в создании Программного обеспечения (ПО) для робототехники:

- Поддержки реального времени: так как, ядро ROS2 написано на C++, используя вычисления на этапе компиляции и другие передовые практики при создании ПО обеспечивается низкие накладные расходы на использование.

- Масштабируемости: Использование сетевых возможностей, легко поддерживается работа в распределённых и многокомпонентных системах.
- Безопасности: Использование встроенных механизмов аутентификации и шифрования.
- Кросплатформеность: поддержка различных операционных систем и архитектур процессоров.

.

В основе ROS2 находится Data Distribution Service (DDS)[29][30] — стандартизированном протоколе обмена сообщениями, обеспечивающем надёжную и масштабируемую коммуникацию между компонентами системы. Этот протокол является приемником протокола первой версии ROS, который не только значительно расширил возможности к настройке взаимодействия компонент, но и так же, первый протокол, который не был придуман разработчиками ROS, а просто был взят в качестве базы для ROS2.
Основные элементы архитектуры:

- ROS2 демон (ROS2 daemon) - программа-демон, обеспечивающая работу всех механизмов.
- Узелы (Nodes): минимальные единицы компонентой системы, выполняющие определённые задачи.
- Топик (Topic): механизм обеспечивающий передачу сообщений между узлами.
- Сервис (Service): механизм синхронного взаимодействия между узлами, по аналогии с запросами и ответами в клиент/сервисной архитектуре.
- Действия (Actions): механизм поддержки длительных операций с возможностью отслеживания прогресса.
- Файлы запуска (Launches): механизм запуска группы узлов, позволяющий управлять временем и последовательностью запуска. Позволяет строить сложные последовательности запуска компонентой системы.
- Сервер параметров (Parameters Server): хранилище настроек каждого узла запущенного в системе. Позволяет изменять поведение узлов в процессе работы, а также отлаживать узлы при определенных настройках.
- Коммуникационный граф (Communication Graph) - абстракция взаимодействия узлов посредствам топиков, сервисов и т.п.

.

Замечание 1: *Демоном называется процесс, работающий в фоновом режиме без прямого взаимодействия с пользователем*

Эта архитектура обеспечивает гибкость и модульность, позволяя создавать сложные и надёжные робототехнические системы.

1.1.1 Основы ROS2

При первом использовании ROS2 становится заметной большая разница в работе, если раньше для обеспечения работы механизмов требовался запуск ROS Master сервера - подобия сервера, через которого идет связь, то сейчас запуск компонентов можно осуществлять без него. Дело в том, что сейчас в основе ROS2 лежит Data Distribution Service (DDS), который использует автоматическое обнаружение других компонент, поэтому необходимость в сервере для обработки отпала, однако, в системе существует ROS2 daemon, который автоматически запускается вместе с компонентом, но вместо функции сервера для связи, он исполняет функции сервера для кэширования, позволяя ускорить процесс поиска других компонентов там, где это возможно.
Замечание 2: *Кэшированием называется хранение важные данных в высокоскоростной памяти, так кэшэ*

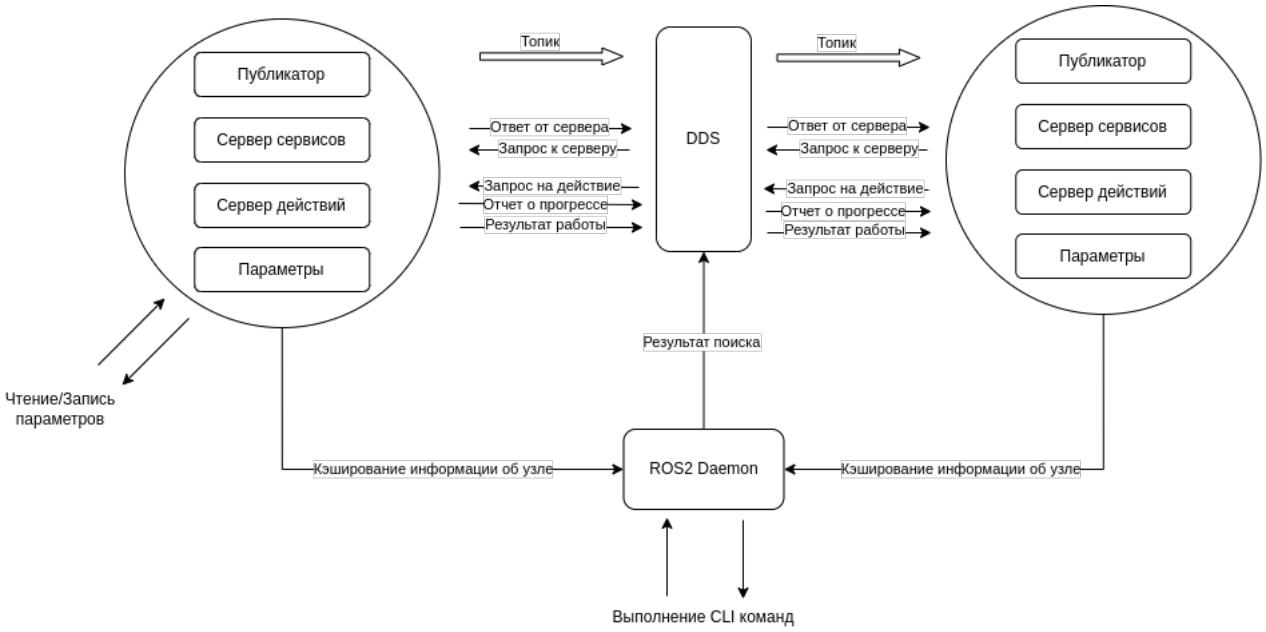


Рис. 1.1: Взаимодействие структурных частей

После запуска хотя бы одной из компонент системы, при условии, что она публикует какие-то данные, используя топики, можно увидеть список всех топиков в системе, для этого в новом окне оболочки системы следует запустить команду:

ros2 topic list

Каждый топик здесь, это отдельный от остальных, поток данных имеющий определенную структуру и тип. Используя ключ -v, где v - от англ. verbosity, можно увидеть отображение имени топика на его тип, подробнее о типе можно узнать

посмотрев соответствующую документацию. Для того, чтобы дублировать поток данных в оболочку существует команда:

$$ros2\ topic\ echo\ [topic\_name]$$

где $[topic\_name]$ - имя необходимого топика.
Существует несколько способов запуска ROS2 компонент, используя запуск единичного компонента и используя launch файлы, который запускают несколько компонент
Запуск единичного узла осуществляется с помощью команды:

$$ros2\ run\ [pkg\_name]\ [exe\_name]$$

Запуск launch файла осуществляется командой:

$$ros2\ launch\ [pkg\_name]\ [launch\_name]$$

где $[pkg\_name]$ - имя необходимого пакета, $[exe\_name]$ - имя исполняемого узла (ноды), $[launch\_name]$ - имя launch файла. Пакетом , в данном случае, называется отдельный от других набор исходного кода, единой направленности.
Команды приведенные выше, являются основной частью Command Line Interface для ROS2 экосистемы, которые помогают управлять работой и ускорять отладкой узлов. Command Line Interface содержит еще много команд, которые представляют интерес для разработчиков, подробнее о этой теме можно узнать по ссылке

1.1.2 Пакеты ROS2

Пакетом в системе ROS2 называется некоторое множество кода единой направленности, которая удовлетворяет следующим критериям:

- Адресуемость в окружении: кроме того, что пакет должен хранится на диске, чтобы с ним можно было взаимодействовать необходимо так же хранить его в определенной области, обычно папке, которая называется окружением. В данном окружении в дальнейшем, ROS будет искать компоненты для запуска и проводить сборку пакетов.

- Структура: файлы в пакете должны хранится согласно описаной в документации структуре, чтобы с ним можно было взаимодействовать.

- Наличие файла сборки: для сборки пакета и трансформации исходного кода в исполняемые файлы (компоненты) необходимо использовать файл, которые хранил бы в себе инструкции о том, каким образом компилировать и линковать файлы пакета

- Наличие файла метаинформации о пакете: для определения пакета ROS2 окружением, необходимо заполнить файл с метаинформацией. Эта информация помогает системе сборки корректно определять синтаксис файла сборки, зависимости от друих пакетов и т.п.

Каждый из критериев следует расширить, для более полного понимания. Говоря о адресуемости, следует заметить, что окружение, обычно папка на компьютере, является местом хранения множества пакетов, которые, строго говоря, должны хранится в подпапке src окружения в отдельных папках, которые емко отражают направленность пакета. Например, название папки nav2, которая хранит исходные коды пакета, явно указывает на то, что данный пакет реализует компоненты системы навигации. Переходя к следующему критерию - структуре, необходимо добавить, что главное правило, которое необходимо соблюдать - наличие файла сборки, дальнейшие указания, носят лишь рекомендательный характер. Файл сборки должен быть написан с использованием одного из языков систем сборки CMake[31][32], Python setuptools, Gargo и т.д. и ,соответсвенно, должны соблюдать и синтаксические правила этих языков. В данной работе, будет использован язык системы сборки CMake, так как именно с помощью этого языка составляются файлы сборки пакетов написанных на языке C++. Кроме того, ROS2 несколько расширяет функциональность стандартного CMake добавляя собственные функции, которые можно использовать при сборке с использованием системы сборки colcon. Файл с метаинформацией о пакете, позволяет сделать пакет видимым для системы сборки colcon. В нем находится как информация о конфигурации файла сборки пакета, его зависимостях от других пакетов, экспортируемых в оболочку переменных окружения, так и информация о разработчиках пакета, сайте проекта и т.п., но раскрытие данной информации происходит исключительно по желанию самих разработчиков.

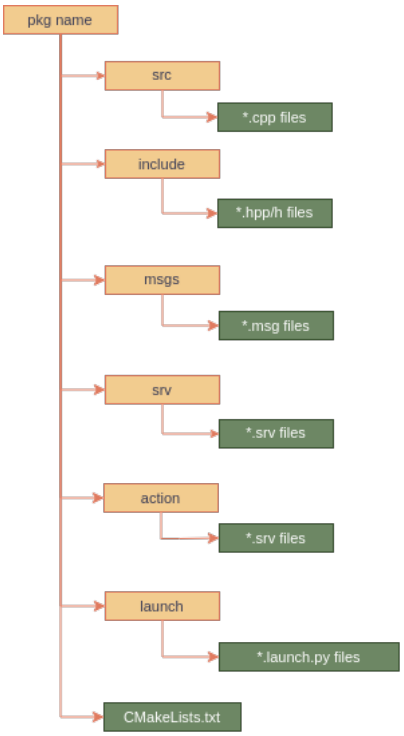


Рис. 1.2: Рекомендуемая структура пакета

Немаловажной деталью темы пакетов является система сборки[33][34]. В ROS2 системой сборки по-умолчанию является colcon. Colcon - система сборки, используя-

щая другие системы сборки, унифицируя их интерфейс для пользователя. Colcon поддерживает файлы сборки написанные на языках CMake, Python setuptools и т.д. Colcon предоставляет Command Line Interface (CLI)[35][36] пользователям, с помощью которого можно удобно и быстро собирать и тестировать пакеты. Местом, в котором система сборки и пакеты непосредственно соприкасаются и взаимодействуют, называется окружением. Окружений может быть любое количество, однако, подобное зачастую не имеет смысла, так как, окружения существуют для удобного размещения пакетов и уже собранных компонент пакетов. Окружение, как и пакет, имеет структуру, однако в отличии от пакета, структура окружения задана в более строгой форме. Каждое окружение должно иметь подпапки src - подпапка в которой храняться пакеты, install - подпапка в которой хранятся символьные ссылки на некоторые файлы из src, build - подпапка в которой храняться файлы необходимые для сборки и log - подпапка в которой храняться файлы с информацией о запусках, сборках и т.п. Если подпапку src, пользователь должен создавать сам, то остальные папки автоматически создаются системой сборки colcon во время первой сборки пакета в данном окружении.

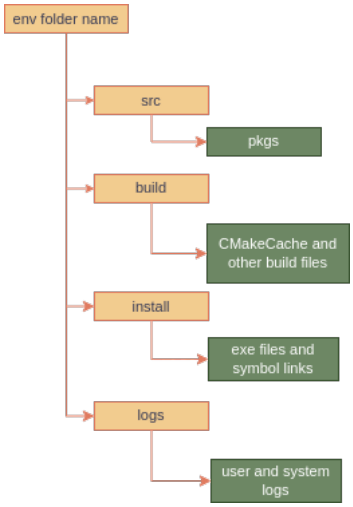


Рис. 1.3: Структура окружения

Сборка с помощью colcon осуществляется в корневой папке окружения с помощью команды:

colcon build

С помощью ключа -h (сокращение от help), можно вывести краткую справку по остальным ключам, среди наиболее полезных и частоиспользуемых ключей находится ключи –symlink-install и –packages-select. Ключ –symlink-install позволяет создавать символьные ссылки на исходный код, который не нуждается в постоянной пересборке, например python исходный код, так как он является интропретируемым, а не компилируемым, следовательно, после его изменения не требуется его пересборка, это позволяет экономить время на сборку всего пакета, т.к. ссылка устанавливается один раз, а далее не требует обслуживания. Ключ –packages-select

позволяет выбрать один или несколько пакетов, которые необходимо собрать, позволяя не собирать все пакеты, находящиеся в подпапке src окружения.
Рассмотрим процесс написания файла сборки на следующем примере:

```
cmake_minimum_required(VERSION 3.16)
project(eureka_odometry LANGUAGES CXX)

if(CMAKE_CXX_COMPILER_ID MATCHES "(GNU|Clang)")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()

if(NOT CMAKE_CXX_STANDARD)
  set(CMAKE_CXX_STANDARD 14)
endif()

# find dependencies
set(THIS_PACKAGE_INCLUDE_DEPENDS
  sensor_msgs
  geometry_msgs
  rclcpp
  tf2_msgs
  tf2_geometry_msgs
  tf2
  realtime_tools
  rcpputils
  nav_msgs
  pluginlib
)

foreach(Dependency IN ITEMS ${THIS_PACKAGE_INCLUDE_DEPENDS})
  find_package(${Dependency} REQUIRED)
endforeach()

include_directories(include/)

add_executable(${PROJECT_NAME}
  src/steering_odometry.cpp
  src/eureka_odometry.cpp
  src/main.cpp
)

ament_target_dependencies(${PROJECT_NAME} ${THIS_PACKAGE_INCLUDE_DEPENDS})

# INSTALL
install(TARGETS
  ${PROJECT_NAME}
  DESTINATION lib/${PROJECT_NAME})

install(DIRECTORY
        launch config
        DESTINATION share/${PROJECT_NAME})

ament_package()
```

Листинг 1.1: Файл сборки пакета eureka_odometry

Данный файл сборки принадлежит пакету eureka_odometry. Структура файла является стандартной для сборки небольших C++ проектов, однако есть и некоторые новые функции, которые отличают этот файл сборки от подобных, предназначенных для сборки обычных проектов, без использования ROS2. Во-первых, разработчики создали инфраструктуру на CMake, которая позволяет находить другие пакеты, необходимые для сборки, используя стандартную функцию поиска find_package, т.е. разрабочику больше не нужно писать собственные субпрограммы на языке CMake для поиска необходимых для сборки пакетов. Во-вторых, добавлены новые функции, которые упрощают процесс линковки библиотек из других пакетов, например, функция ament_target_dependencies(...), которая первым аргументом принимает цель, с которой нужно слинковать библиотеки, а следующими аргументами - имена пакетов из которых будут выделены библиотеки.В-третьих, для обеспечения корректной работы системы сборки, необходимо наличие функции ament_package(...), которая должна быть завершающей функцией в каждом файле сборки ROS2 пакета. Она позволяет, colcon проверять информацию о пакетах находящихся в файле с метаинформацией, создать скрипты оболочки позволяющие обнаруживать пакеты на компьютере с помощью ROS2 CLI команд (CLI - Command Line Interface) и т.п.

Замечание 1: *Линковкой называют процесс связывания скомпилированных динамических библиотек в исполняемый файл. Является одним из этапов создания компилятором исполняемого файла из исходного кода в компилируемых языках программирования*

Замечание 2: *Динамической библиотекой называется скомпилированный особым способом исходный код, который связывается (линкуется) с пользовательской программой во время ее сборки*

Файл с метаинформацией о пакете, является не менее важной частью в процессе создания и регистрации пакета, поэтому ниже также будет расмотрен процесс создания на следующем примере:

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd" schematypens="http://www
    .w3.org/2001/XMLSchema"?>
<package format="3">
  <name>eureka_odometry</name>
  <version>1.0.0</version>
  <description>Collect joint states and publish odometry</description>
  <license>Apache License 2.0</license>
  <maintainer email="sewa.egorov@yandex.ru">Vsevolod Egorov</maintainer>

  <author email="sewa.egorov@yandex.ru">Vsevolod Egorov</author>

  <buildtool_depend>ament_cmake</buildtool_depend>

  <build_depend>generate_parameter_library</build_depend>

  <depend>sensor_msgs</depend>
  <depend>tf2_msgs</depend>
```

```
18   <depend>tf2_geometry_msgs</depend>
19   <depend>tf2</depend>
20   <depend>nav_msgs</depend>
21   <depend>pluginlib</depend>
22   <depend>rcpputils</depend>
23   <depend>realtime_tools</depend>
24   <depend>geometry_msgs</depend>
25   <depend>rclcpp</depend>
26
27   <test_depend>ament_cmake_gmock</test_depend>
28
29   <export>
30     <build_type>ament_cmake</build_type>
31   </export>
32 </package>
```

Листинг 1.2: Файл метаинформации пакета eureka_odometry

Рассмотрим ключевую информацию, необходимую для создания файла метаинформации. Во-первых, парный тэг <name>...</name>, в котором необходимо указывать имя пакета, аналогичное имя необходимо указывать в файле сборки внутри функции project(...). Во-вторых, парные тэги типа сборки - <buildtool_depend>... и <build_type>...</build_type>, которые сообщают системе сборки об особенностях выбора подсистемы сборки, например, ament_cmake, сообщает colcon о том, что для сборки будет использоваться система CMake и файла сборки написан с использованием языка CMake. В-третьих, парные тэги семейства depend, есть небольшое множестов тэгов этого семейства, среди которых <depend>...</depend>, <build_depend>...</build_depend>, <test_depend>...</test_depend> и <exec_depend> Парный тэг <depend>...</depend> отдает управление зависимостью colcon, который должен определить к какому конкретному типу пренадлежит зависимость, другие тэги из семества определяют зависимость как зависимость сборки, тестирования, запуска соответственно. Тэги данного семейства позволяют управлять функциями подсистемы сборки, например, зависимость указанная как exec_depend не может быть найдена через find_package(...) CMake функцию в файле сборки и ,соответсвенно, не может быть слинкована с исполняемым файлом данного пакета, однако, должна быть найдена для успешной сборки пакета, так как может быть необходима для создания и запуска launch файлов (файлов запуска).

1.1.3 Использование ROS2

Создание пакета и использование ROS2 CLI, является важной частью работы с ROS2 инфраструктурой, однако, полезную нагрузку создается именно наличие вычислительной сети из узлов и топиков. В ROS2 существует два способа запуска исполняемых файлов (узлов) - нативно с помощью ROS2 CLI, или через launch файлы. Запуск узлов с помощью ROS2 CLI был рассмотрен выше, поэтому подробно остановимся на запуске узлов с помощью launch файлов.
launch файл представляет собой инструкции по запуску узлов, в котором, кроме того может быть задана очередность запуска , запуск вспомогательных утилит и много другое. launch файл может быть написан нескольких языках программиро-

вания, например на python или xml. Рекомендуемым ROS2 сообществом языком для создания launch файлов является Python[37][38], за его гибкость и хорошую интеграцию. Ниже будет представлен листинг простого launch - а на языке Python с разбором основных моментов:

```
import os

from ament_index_python.packages import get_package_share_directory

from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument, IncludeLaunchDescription
from launch.conditions import IfCondition
from launch.substitutions import LaunchConfiguration, PathJoinSubstitution
from launch.launch_description_sources import PythonLaunchDescriptionSource

from launch_ros.actions import Node, SetParameter

ARGUMENTS = [
    DeclareLaunchArgument('spawn_robot', default_value='True',
                          choices=['True', 'False'],
                          description='Spawn the eureka robot model.'),
    DeclareLaunchArgument('pose_estimator', default_value="'odometry'",
                          choices = ["'eagleye'", "'rgbd_odometry'", "'odometry'"],
                          description='Pose estimators for eureka robot'),
    DeclareLaunchArgument('model', default_value='eureka',
                          description='Model to use for simulation')
]


def generate_launch_description():

  package_name = 'eureka_simulation'

  model_type = LaunchConfiguration('model')
  spawn_robot = LaunchConfiguration('spawn_robot')
  pose_estimator = LaunchConfiguration('pose_estimator')

  # launch's path
  world_launch_path = os.path.join(get_package_share_directory(package_name), 'launch',
  'world', 'world_create.launch.py')
  localization_launch_path = os.path.join(get_package_share_directory(package_name), 'launch',
  'navigation', 'localization.launch.py')
  navigation_launch_path = os.path.join(get_package_share_directory(package_name), 'launch',
  'navigation', 'navigation.launch.py')

  # rviz config
  rviz2_config = PathJoinSubstitution([get_package_share_directory(package_name), 'rviz',
  'eureka.rviz'])


  # world initialization
  world = IncludeLaunchDescription(PythonLaunchDescriptionSource(world_launch_path),
      launch_arguments={'model': model_type,'spawn_robot': spawn_robot}.items())

  # dynamic localization initialization
  localization = IncludeLaunchDescription(PythonLaunchDescriptionSource(
      localization_launch_path),launch_arguments={'pose_estimator': pose_estimator}.items())

  # mapping initialization
  navigation = IncludeLaunchDescription(PythonLaunchDescriptionSource(navigation_launch_path))

```

```
55  # rviz initialization
56  rviz = Node(package='rviz2',
57            executable='rviz2',
58            name='rviz2',
59            arguments=['-d', rviz2_config],
60            parameters=[],
61            remappings=[
62               ('/tf', 'tf'),
63               ('/tf_static', 'tf_static')
64            ],
65            output='screen'
66  )
67
68  use_sim_time_param = SetParameter(name='use_sim_time', value=True)
69
70  ld = LaunchDescription(ARGUMENTS)
71  ld.add_action(world)
72  ld.add_action(localization)
73  ld.add_action(navigation)
74  ld.add_action(rviz)
75  ld.add_action(use_sim_time_param)
76  return ld
```

Листинг 1.3: launch файл пакета eureka_localization

Первым блоком идет определение импортируемых функции, ROS2 сообщество основательно поработало над созданием функции почти под все сценарии использования. Следующий блок обьявляет лист с аргументами, которые могут быть переданны launch - у пользователем который запускает данный launch или launch - ем запускающим данный. Далее идет определение главной функции - generate_launch_description() , в которой и происходит запуск всех необходимых узлов. С помощью функции LaunchConfiguration(...) переданные launch - у аргументы могут быть захвачены и преобразованые в переменные с некоторым типом (так как Python не строго типизированный, то до конца не ясно в какой), обычно в тип строки, и использоваться, например, для конфигурирования работы launch файла. Следующий блок захватывает программные пути к библиотекам с помощью функций библиотеки os, в качестве аргумента предается путь до папки с содержимым необходимого пакета, в котором содержится исполняемые файлы (узлы) и вспомогательная информация. Данные папки содержаться в окружении в качестве подпапки папки install. Для понимания дальнейших инструкции следует упомянуть о том, что launch файлы могут быть включены в другие launch файлы, т.е. при запуске может существовать верхнеуровневый launch, который будет запускать другие launch - и. Включение других launch файлов осуществляется с помощью команды IncludeLaunchDescription, в которую, в качестве аргументов, передается путь до другого launch файла и аргументы, которые должны быть переданы этому launch файлу, как если бы его запускали из командной строки. После инструкций добавляющих launch файлы к запуску, идет инструкция запуска узла rviz2. Запуск узла осуществляется с помощью команды Node(...), в качестве аргументов передается имя пакета, имя исполняемого файла (то имя, которое было задано в файле сборки), аргументы, как если бы запуск происходил из командной строки, параметры, в форме {имя параметра:значение параметра} или в

форме пути до .yaml файла с параметрами, переопределения топиков - ROS2 дает возможность переименовывать топики для конкретного узла, это сделано потому что, узлы ожидают определенный топик, а топики различаются, в основном, по имени. Далее идет непосредственно формирование списка из узлов, который будет передан ROS2 платформе. Список создается функцией LaunchDescription(...), которая в данном случае принимает аргументы launch - а, однако, может принимать и список узлов, после чего следует передача сущности LaunchDescription непосредственно в экосистему ROS2 и запуск.

1.1.4 FastDDS Server

FastDDS Discovery Server — это альтернативный механизм обнаружения узлов в ROS 2, заменяющий стандартный протокол Simple Discovery Protocol (SDP)[39]. Он решает ключевые проблемы масштабируемости и надежности в распределенных системах, таких как робототехнические платформы, IoT-сети и промышленные IoT-решения. В данной работе рассматриваются архитектура, функциональные возможности и практические примеры интеграции Discovery Server в ROS 2.

FastDDS Discovery Server — это альтернативный механизм обнаружения узлов в ROS 2, заменяющий стандартный протокол Simple Discovery Protocol (SDP). Данный протокол помогает решить проблему масштабируемости в распределенной компонентной системе. В основе лежит клент/серверная архитектура, в которой сервер выполняет роль клея для соединения узлов между собой, таким образом экономится трафик, так как отсутсвет механизм multicast (вид поиска устройств в сети, при котором устройства посылает специального вида сообщение в надежде на ответ) и соединение идет напрямую через сервер.
Данный механизм имеет множество возможностей, в том числе и возможность создания резервных серверов для связи, создания сохраняющих состояние во время сбоев и т.п., однако здесь будет расмотрены лишь основные возможности. Создание сервера происходит с помощью команды:

$$fastdds\ discovery\ \text{-}l\ [inteface\_ip]\ \text{-}p\ [port]$$

после этого fastdds сервер будет слушать подключение к указанному ip и порту, чтобы воспользоваться связью через данный сервер необходимо указать environment variable - переменную окружению ROS_DISCOVERY_SERVER с ip и портом сервера, например:

$$export\ ROS\_DISCOVERY\_SERVER{=}127.0.0.1{:}11811$$

после этого, все узлы публикующие сообщения в топики, будут связываться через fastdds сервер.

1.2 Docker

Docker — это программная платформа с открытым исходным кодом, предназначенная для автоматизации развертывания, масштабирования и управления приложениями в изолированных средах, называемых контейнерами. Контейнеры позволяют упаковать приложение вместе со всеми его зависимостями, обеспечивая его стабильную работу в различных средах и упрощая процессы разработки и эксплуатации. Docker был создан компанией docCloud.inc, как внутрений продукт компании, однако, после презентации проекта широкой публике, Docker быстро набрал популярность у программистов и системных инженеров, став мощным инструментом для инфраструктурных задач.

Основные компоненты Docker платформы:

- Docker Engine: основной компонент, обеспечивающий создание, запуск и управление контейнерами.
- Docker Images: шаблоны, содержащие все необходимое для запуска приложения, включая код, библиотеки и зависимости.
- Docker Containers: изолированные среды, созданные на основе образов, в которых выполняются приложения.
- Docker Compose: инструмент для определения и управления многоконтейнерными приложениями с использованием YAML-файлов.
- Docker Hub: облачный реестр для хранения и распространения Docker-образов.

Среди преимуществ использования данного решения в данной работе можно отметить:

- Портативность: контейнеры Docker могут быть запущены на любой системе, поддерживающей Docker, что обеспечивает консистентность между различными средами разработки и эксплуатации.
- Изоляция: каждое приложение работает в своем контейнере, что предотвращает конфликты между зависимостями и обеспечивает безопасность.
- Масштабируемость: Docker облегчает горизонтальное масштабирование приложений путем запуска нескольких контейнеров одновременно.
- Быстрое развертывание: контейнеры запускаются значительно быстрее по сравнению с традиционными виртуальными машинами, что ускоряет процессы разработки и тестирования.
- Эффективное использование ресурсов: контейнеры используют ресурсы хост-системы более эффективно, чем виртуальные машины, что снижает затраты на инфраструктуру

Docker будет использоватся для построения топологии компонентов на облачной платформе. Контейнера будут абстрагировать программные компоненты, а точнее, ROS 2 ноды, в которых, уже существуют механизмы для управления потоками данных, таким образом, топология компонентов будет использовать комбинацию возможностей Docker для разделения программных компонент между собой и выстраивания инфраструктуры для сборки и запуска этих компонент, а выбор в сторону ROS2 программных компонент позволит управлять потоками данных между компонентами, давая легкую настройку этой связи.

1.2.1 Основы Docker

Под названием Docker в дальнейшем будет подразумеваться Docker Engine в связке с инфраструктурой (Docker Compose, Docker Swarm и т.п.). Docker Engine - это ярдо платформы, представляющее собой сервер, который управляет контейнерами, образами и т.д., которое к тому же, предоставляет REST API и Docker CLI, для взаимодействия с сервером. Если первый способ взаимодействия, через REST API больше подходит приложениям, то вторым, через Docker CLI, обычно пользуются системные инженеры и программисты для удобной работы с Docker Engine сервером на компьютере. Использование возможностей данной платформы начинается с создания Dockerfile - ов и понимания термина образ. Dockerfile - это набор инструкций для создания образа, а образ своеобразные слепок, по которому создаются контейнера. Рассмотрим создания простейшего Dockerfile и образа из него:

```
ARG DISTRO=humble

FROM ros:humble-ros-base-jammy

RUN apt-get update && apt-get install -y \
  ros-${DISTRO}-tf2 \
  ros-${DISTRO}-tf2-ros \
  ros-${DISTRO}-angles \
  ros-${DISTRO}-nav2-amcl \
  ros-${DISTRO}-nav2-map-server \
  && rm -rf /var/lib/apt/lists/*

ARG CONTAINER_NAME
ARG USER=docker_${CONTAINER_NAME}
ARG UID=1000
ARG GID=1000

# Create a non-root user
RUN groupadd --gid $GID $USER \
  && useradd -s /bin/bash --uid $UID --gid $GID -m $USER \
  # Add sudo support for the non-root user
  && apt-get update \
  && apt-get install -y sudo \
  && echo $USER ALL=\(root\) NOPASSWD:ALL > /etc/sudoers.d/$USER\
  && chmod 0440 /etc/sudoers.d/$USER \
  && rm -rf /var/lib/apt/lists/*

WORKDIR /home/$USER
```

```
29 RUN cd /home/$USER && mkdir -p colcon_ws/src colcon_ws/log colcon_ws/install && chown -R ${UID
       }:${GID} ./
30 RUN usermod -G dialout -a $USER && newgrp dialout
```

Листинг 1.4: Простой Dockerfile

Ключевое слово ARG - позволяет принимать аргументы переданные запускающей программой, чаще всего bash оболочкой, таким образом Dockerfile становятся конфигурируемыми. Далее FROM - ключевое слово, которое позволяет использовать любой образ имеющийся в облачном регистре (при наличии интернет соединения) или на компьютере, в качестве базового для собирающегося образа. Ключевое слово RUN используется для выполенения комманд заданных пользователем. WORKDIR - просто устанавливает рабочую директорию и напрямую не влияет на сборку. Выделим главные этапы сборки данного образа:

1. Выбор базового образа для текущего собираемого. Это позволяет упростить данный Dockerfile до простейших комманд над заранее собранном образом.

2. Создание аргументов. Повзоляет делать инструкции конфигурируемые не прибегая к редактированию файла.

3. Загрузка дополнительных утилит. Опционально, однако, если образ на базе GNU/Linux[40][41], позволяет экономить время и использовать заранее скомпилированные программы.

4. Создания инфраструктуры внутри будущих контейнеров данного образа. Опционально, однако также позволяет экономить время, создавая общую для всех контейнеров инфраструктуру еще на этапе создания образа.

Запуск сборки данного образа можно выполнить использовав POST запрос к DockerEngine демону, либо использовать вместо REST API CLI запрос, через командную оболочку. Воспользуемся вторым способом, так как он легче для понимания:

$$docker\ build\ –tag\ [new\_image\_name]$$

где $[new\_image\_name]$ - название образа, по которому его можно будет идентифицировать. После запуска команды начнется процесс сборки, если на одном из этапов сборки произойдет ошибка, Docker закэширует данные о предыдущих этапах. После окончания процесса сборки, образ будет хранится в регистре Docker Engine демона, просмотреть какой идентификационный код был присвоен демоном образу и другие данные о собранном образе можно с помощью команды:

$$docker\ images$$

Создать и запустить контейнер из собранного образа можно несколькими способами, один из них это указать в Dockerfile ключевое слово CMD, которое означает команду по-умолчанию, которая будет выполненна при создании и старте контейнера, созданного из данного образа, другим способом является создание и запуск

контейнера "вручную с помощью Docker CLI команды, третьим способом является создание docker-compose файла, который подобен по смыслу launch файлам из ROS2, с его помощью можно заранее определить свойства целой группы контейнеров. Воспользуемся вторым по счету методом, и создадим контейнер с помощью Docker CLI, для этого необходимо использовать комманду:

$$docker\ run\ -it\ [tag]\ [command]$$

где $[tag]$ - имя образа, которое мы передали с ключем –tag при его создании, а $[command]$ - произвольная команда, которая будет выполененна оболочкой при старте контейнера. Просмотреть данные о запущенном контейнере можно с помощью команды:

$$docker\ ps$$

1.2.2 Преимущества Docker

Docker остается одним из наиболее популярных решений для контейнеризации, особенно в контексте микросерверной архитектуры. Далее буду рассмотрены альтернативы для контейнеризации и обоснова выбор Docker для данного исследования. Среди основных конкурентов Docker можно выделить таких решения как Podman и LXC, оба этих решения являются средами для контейнеризации, кратко рассмотрим их преимущества и недостатки.
Преимущества Podman:

1. Повышенная безопастность. Так как система не имеет единого демона владеющего привелегиями root, то при взломе, злоумышленик не сможет управлять всеми контейнерами и потенциально, всей системой, также контейнера запускаются от имени пользователей, которые создаются Podman используя пространство имен пользователя GNU/Linux, что позволяет отследить запуск до конкретного пользователя.

2. Совместимость с Docker CLI. Возможность использовать Podman в качестве ядра и управлять им с помощью команд Docker CLI.

Недостатки Podman:

1. Менее зрелая экосистема. Из-за сравнительно небольшой популярности, отсутвует необходимая поддержка для экосистемы такого масштаба, следовательно многие функции существующие в Docker, не поддерживаются Podman.

Преимущества LXC:

1. Запуск контейнеров с полной операционной системой. Полноценная операционная система, является как сильной стороной, т.к. позволяет создать собственные сетевые интерфейсы, пользователей, пространства процессов и иметь собственную файловую систему, так и недостатком, т.к. наличие не несущих пользы возможностей накладывает ограничения на масштабируемость подобных контейнеров.

2. Гибкость. Тонкий процесс настройки потребляемых ресурсов.

Недостатки LXC:

1. Сложность в настройке. Гибкость и запуск контейнеров с полноценной операционной системой требуют соответсвующей квалификации у людей работающих с настройками данной технологии.

2. Ограниченная поддержка сообщества. Сравнительно низкая популярность, вследствии чего, недостаток специалистов способных развивать экосистему.

В тоже время Docker, сочетает в себе гибкость, способность к масштабированию и сильную поддержку сообщества, которая делает данный инструмент наиболее сбалансированным и поддерживаемым решением для данного проекта. Широкая экосистема, удобство использования делают Docker оптимальным выбором для разработки и развертывания сетевых сервисов.

1.3 Алгоритмы одновременного картографирования и локализации

SLAM (Simultaneous Localization and Mapping)[42][43] — это метод, используемый автономными мобильными системами[44], позволяющий одновременно строить карту неизвестной среды и определять своё местоположение в ней. Эта технология стала ключевой в области робототехники, обеспечивая навигацию без предварительно известной карты. Алгоритмы одновременного картографирования и локализации решают задачу одновременного построения карты окружающей среды и определения положения робота на этой карте путём обработки последовательности управляющих воздействий и наблюдений, обновляя оценки положения и карты на каждом шаге времени. Основой большинства алгоритмов одновременного картографирования и локализации является использование фильтров на основе байесовского фильтра[45][46], позволяющий учитывать неопределённость и шум в данных.

Существует несколько основных видов алгоритмов одновременного картографирования и локализации по типу карты[47] - метрические, топологические и семантические. Все алгоритмы построенные по метрическому типу используют для локализации байесовский фильтр, в основе которого лежит байесовская рекурсия, которая обновляет апостериорное распределение состояния роботы и карты на каждом шаге:

$$P(x_k, m|Z_{0:k}, U_{0:k}) = \eta P(z_k|x_k, m) \int P(x_k|x_{k-1}, u_k) P(x_{k-1}, m|Z_{0:k-1}, U_{0:k-1}) dx_{k-1}$$

где:

- x_k - состояние робота в момент времени k.

- m - карта окружающей среды.
- z_k - наблюдение в момент времени k.
- u_k - управляющее воздействие в момент времени k.
- $Z_{0:k}$ и $U_{0:k}$ - последовательность наблюдение и управляющих воздействия до момента k включительно.
- η - нормализующий множитель.

Это уравнение отражает двухэтапный процесс, на котором строятся байесовские фильтры:

1. Прогноз - с помощью модели движения робота и управляющих воздейсвий предсказывается новое состояние $P(x_k|x_{k-1}, u_k)$
2. Коррекция - с помощью модели наблюдения и данных с сенсоров прогноз корректируется $P(z_k|x_k, m)$

Все фильтры подобного типа основаные на марковских свойствах, т.к. текущее состояние зависит только от предыдущего и управляющих воздействий на шаге k. Алгоритмы данного типа создают карту привязаную к некой координатной системе, в которой объекты, имеющие собственные размеры аналогичные реальным, привязаны к координатной системе.

Более продвинутым вариантом метрических алгоритмов являются топологические алгоритмы[48], которые еще называют графовыми, потому что окружающая среда в представлении алгоритма более походит на вершины, соединенные ребрами - возможными путями между вершинами. Граф, как структура данных, используемая в топологических алгоритмах более всего подходит для создания карт больших или слабо структурированных пространств. Основной задачей топологических алгоритмов является задача оптимизации[49], которая формулируется как задача оценки траекторий и карты, основанных на зашумленных измерениях одометрии и наблюдаемых ориентиров. Цель заключается в нахождении конфигурации вершин, которая бы минимизировала суммарную ошибку между предсказанием и фактическим результатом. Математически задача формулируется следующим образом:

$$x = arg\,min_x \sum_{(i,j)\in C} e_{ij}^T \Omega_{ij} e_{ij}$$

где:

- x - вектор состояния.
- e_{ij} - ошибка между измерением z_{ij} и предсказанным значением $\hat{z}_{ij}(x_i, x_j)$.
- Ω_{ij} - информационная матрица, обратная ковариации шума измерения, отражающая уверенность в измерении.

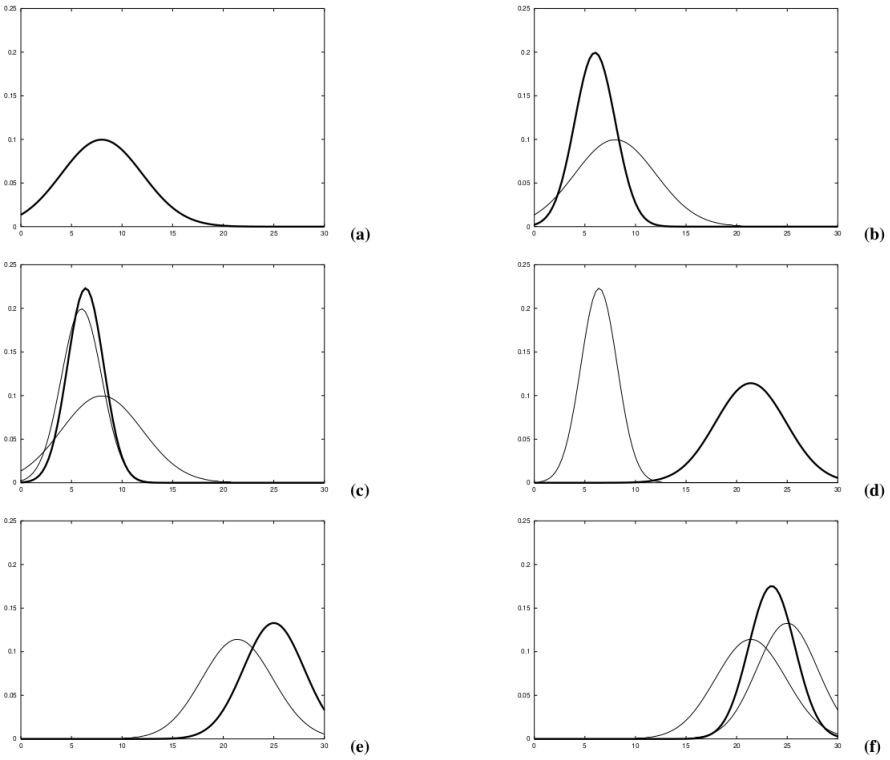


Рис. 1.4: Процесс прогноза и коррекции в байесовском фильтре. График (a) представляет начальное распределение, (b) более темный график - измерение со своим распределением, (c) распределение после интеграции измерения (коррекция), (d) распределение после некоторого управления (предсказание), (e) более темный график - новое измерение со своим распределением, (f) распределение после интеграции измерения (коррекция)

- C - множество всех пар связанных узлов (i, j) в графе.

Эта формулировака основывается на предположении о гауссовском шуме и независимости измерений, что позволяет преобразовать задачу в минимизацию отрицательного логарифма правдоподобия. Таким образом, задача оптимизации направлена на нахождение вероятной конфигурации состояний, согласующейся с полученными измерениями. Для решения часто используют иттеративные методы, такие как метод Гаусса-Ньютона или алгоритм Левенберга-Марквардта, т.к. они их легко запрограммировать под вычислительные устройства. Эти методы требуют линеаризации функции ошибок вокруг текущей оценки и последующего решения полученной линейной системы уравнений:

$$H\Delta x = -b$$

где:

- H - матрица вторых производных.
- Δx - приращение оценки состояния.

- b - градиент функции стоимости.

После вычисления Δx текущая оценка обновляется:

$$x \leftarrow x + \Delta x$$

Этот процесс будет повторяться до сходимости, т.е. до тех пор, пока изменения в оценке не станут пренебрежимо малыми. Так же, топологические алгоритмы часто интегрируют в себя функциональность метрических алгоритмов, позволяя повысить устойчивость системы.

Следующим витком развития алгоритмов одновременного картографирования и локализации являются семантические алгоритмы[50], способные не только создавать граф из особых объектов на карте и ,подобно метрических алгоритмам, знать расстояния между вершинами в этом графе, но и способны собирать информацию о контексте окружающей среды. Например, подобные решения, могут распознать кластер точек, который является стулом и дать кластеру характеристику, что этот объект является стулом. В семантическом алгоритме, задача оптимизации усложняется, т.к. кроме всего прочего, необходимо учитывать и семантику меток. Одним из подходов к такой задаче является формулировка задачи как оптимизации по фактор-графу[51], где узлы представляют состояния робота и объектов, а ребра - наблюдения и ограничения. Семантические алгоритмы строятся с использованием глубокого обучения позволяющий им осознавать контекст окружающей среды.

$$\theta = arg\,max_\theta \sum_{t=1}^{T} \sum_{k=1}^{K_i} \log \left(\sum_{j=1}^{M} \omega_{t,k,j} p(z_{t,k}|x_t, m_j) \right), \text{ где } \omega_{t,k,j} = p(a_{t,k} = j|\theta)$$

Где:

- θ - параметры, включающие позы робота x_t и семантические ориентиры m_j.
- $z_{t,k}$ - наблюдение k в момент времени t.
- $\omega_{t,k,j}$ - вероятность того, что наблюдение $z_{t,k}$ соответсвует семантическому ориентиру m_j.
- $p(z_{t,k}|x_t, m_j)$ - вероятность получения наблюдения $z_{t,k}$ при позе робота x_t и семантическом ориентире m_j.

Эта формулировка учитывает неопределенность в ассоциации данных между наблюдениями и ориентирами, а также интегрирует семантическую информацию в процесс оптимизации.

Интеграция решения из области Облачной робототехники, к сожалению, пока всего лишь тенденция. Большинство компаний используют алгоритмы одновременного картографирования и локализации, запускаемые на той же вычислительной машине, что и остальные критически важные подсистемы. Это решение

действительно обосновано, потому как алгоритмы должны быстро обрабатывать большой объем пространственных данных с датчиков, однако, с развитием лазерной техники и производством новых оптоволоконных кабелей, стала возможной передача больших объемов данных на больших расстояниях за приемлимое время, порядка 10 мс, что равно одной сотой секунды. Задержки такого порядка позволяют использовать удаленные сервера для использования в задачах одновременного картографирования и локализации, которые работают с периодчиностью подряка одной десятой секунды. То, что раньше считалось невозможным ввиду высокой задержки, вследствии чего алгоритмы использовали искаженное представление мира, не отражающее положение робота в реальности, сейчас может стать главным вектором развития робототехники и индустрии умных устройств. Важно считаться и с тем, что и алгоритмы и микросхемы становятся все более оптимизированным, вследствии чего, не исключен тот факт, что использование решения на одной вычислительно машине, в будущем, окажется более жизнеспособным.
Однако, на данный момент, в традиционном использовании алгоритмов мы все же сильно ограничены производительностью бортового компьютера робота, в то время как на сервере, возможно создать инфраструктуру из серверов любой сложности, которые бы справлялись в разы быстрее с обработкой данных с сенсоров.
Проведем сравнительный анализ существующих алгоритмов одновременного картографирования и локализации и выберем наболее оптимальный, на основе которого и будет работать сетевой сервис.
Критерием выбра алгоритмов для данного сравнительного анализа послужили количество "звезд"в репозитории проекта. Реализация алгоритмов и их последующая интеграция проводится с использованием платформы ROS2.

1.3.1 OpenSlam's Gmapping + AMCL/GMCL

Связка из двух алгоритмов была рассмотрена не случайно, дело в том, что использование одного из них, без другого в целях одновременного картографирования и локализации невозможна. Пакет OpenSlam's Gmapping представляет из себя набор библиотек и инструментов для создания алгоритма одновременного картографирования и локализации, однако разработчиками, в качестве демонстрации функциональности был создан урезанный вариант алгоритма.
slam_gmapping - главный узел в пакете OpenSlam's Gmapping, который представляет из себя реализацию алгоритма, использующего данные с плоского лазерного лидара, для картографирования и локализации на строящейся карте. Картографирование - главная цель этого програмного пакета, локализация здесь не представлена в явном виде (в виде топика). Следовательно локализацию из данного программного пакета нельзя получить, возможно лишь модифицировать его исходный код с целью получения топика с локализацией на карте. Данный алгоритм основан на фильтре частиц. Проект хорошо поддерживается, у него большое сообщество и много документации.

amcl - главный узел в пакете nav2_amcl, который предоставляет реализацию алгоритма Adaptive Monte Carlo Localization (AMCL)[52][53] , который представляет собой адаптивные метод локализации робота в известной 2D-среде (на карте). Данный алгоритм реализует вероятностный подход с использованием фильтра частиц для оценки положения робота, как и алгоритм узла slam_gmapping. Хорошо известен всему ROS сообществу и является одним из наиболее популярных алгоритмов локализации использующихся на простейших мобильных платформах.
gmcl - главный узел пакета GMCL[54], алгоритм которого является потомком AMCL и предоставляет собой более усовершенствованную версию классического алгоритма. GMCL алгоритм вводит новые режимы для фильтра частиц - optimal, intelligent и self-adaptive, которые повышают эффективность распределения частиц, что позволяет улучшить производительность при работе в реальном времени. Пакет реализующий алгоритм, не так хорошо известен, поэтому документация к нему достаточно скромная.
Работа с данной связкой возможна лишь при наличии плоского лазерного лидара, или его аналога. Так как алгоритмы реализованы с использованием платформы ROS2, то данные от сенсора к алгоритмам должные поступать строго по топикам и строго в виде sensor_msgs/msg/LaserScan структуры.
Репозитории:

- OpenSlam Gmapping на GitHub
- AMCL на GitHub
- GMCL на GitHub

1.3.2 hector_slam

Данный алгоритм одновременного картографирования и локализации[55], основанный на фильтре частиц, использует для работы плоские лазерные лидары. Главным фактором конкурентноспособности является способность быстро обрабатывать большое количество данных. Быстрое встраивание новых кадров данных, и грубый этап сопоставления методом Гаусса-Ньютона на выходе дают удовлетворительную карту окружающего мира. Чтобы воспользоваться преимуществами данного алгоритма одновременного картографирования и локализации, необходимо использовать плоский лидар, способный отправлять большое количество снимков в секунду. Хорошо зарекомендовал себя в ROS сообществе, из-за удобства и эффективности.
Основным источником данных для данного метода картографирования и локализации является плоский лидар, однако возможно использовать инерциальный датчик, который улучшит точность локализации.
Репозитории:

- Hector SLAM на GitHub

1.3.3 rtabmap_ros

Данный алгоритм одновременного картографирования и локализации основан на графе[56]. Все предыдущие методы так или иначе используют принципы цепи Маркова, в которых текущее состояние является результатом изменений только предыдущего состояния. Метод графовых алгоритм использует цепи маркова немного по другому, позволяя создавать вершины в графе привязаном к карте. Данные вершины позволяют оптимизировать карту привязаную к ним прямо во время работы алгоритма. Rtabmap - является графовым алгоритмом не в прямом понимании, он лишь объединяет различные методы локализации и картографирования и строит граф, привязаный к карте, чтобы оптимизировать его. Алгоритм способен работать сразу с большим числом источников данных, таких как, RGB камеры, RGBD камеры, плоские лазерные лидары, объемные лазерные лидары, сонары, дально меры, энкодеры, инерциальные датчики ускорения, гироскопы и т.п. , причем количество датчиков одного типа ограничено лишь возможностями процессора.

Репозитории:

- Rtabmap ROS на GitHub

1.3.4 cartographer_ros

Данный алгоритм одновременного картографирования и локализации, как и предыдущий, основан на графах[57]. Работает с сенсорами, который способны выдавать данные в виде облак точек. Поддерживает функциональность картографирования и локализации сразу у нескольких роботов одновременно, что хорошо интегрируется в задачи облачных вычислений алгоритмов одновременного картографирования и локализации.

Репозитории:

- Cartographer ROS на GitHub

Компоненте amcl для корректной работы, нужна другая компонента - map_server, которая является вспомогательной утилитой для конвертации png, pgm и других форматов в сообщение типа nav_msgs::OccupancyGrid для ROS системы и дальнейшей публикацией данного сообщения в системе на топике map. После запуска вспомогательной компоненты мы должны передать топик, в котором публикуются данные с сенсора, в данном случае лидара. Тип таких сообщений sensor_msgs::LaserScan. Семантику других параметров для компоненты можно изучить в документации для данного пакета.

1.4 Технологии и инструменты разработки сетевых сервисов

Прежде чем переходить к инструментам для разработки необходимо упомянуть об общепринятой модели сетевого взаимодействия - Open System Interconnection (OSI)[58][59]. Модель разеляет сетевое взаимодействие на 7 уровне, каждый из которых выполняет определенные функции и поддерживается определенным протоколом. Перечень уровней модели OSI:

- Уровень 1: Физический - отвечает за передачу потока битов по кабелям. Протоколы: Ethernet, USB, DSL.
- Уровень 2: Канальный - отвечает за надежную передачу данных между узлами. Протоколы: Ethernet, Point-to-Point Protocol (PPP) Wi-Fi.
- Уровень 3: Сетевой - отвечает за маршрутизацию данных между сетями. Протоколы: Internet Protocol (IP), Internet Control Message Protocol (ICMP), Address Resolution Protocol (ARP).
- Уровень 4: Транспортный - обеспечивает надежную связь между узлами. Протоколы: Transmission Control Protocol (TCP), User Datagram Protocol (UDP).
- Уровень 5: Сеансовый - отвечает за установку, поддержание и завершение сеансов связи. Протоколы: Remote Procedure Call (RPC).
- Уровень 6: Представление - отвечает за преобразование данных между форматами. Протоколы: SSL/TSL, MIME.
- Уровень 7: Прикладной - отвечает за предоставление сетевых сервисов конечным пользователям. Протоколы: HTTP/HTTPS, FTP, DNS, DHCP.

Это модель работы любого сетевого узла, именно благодаря созданию данной модели удалось распределить и стандартизировать механизм сетевого взаимодействия. Увеличение номера уровня отражает более абстрактные структуры данных, если на первом уровне модели протоколы работают лишь с потоком битов, то на последнем, приложения работают со сложной структурой, заданной в протоколе. Поддержка всех 7 уровней позволяет разработчикам не думать о нижележащих уровнях модели, работая лишь с протоколом текущего уровня.

Для разработки сетевых приложений применяются множество фреймворков и библиотек, однако, не все из них подходят для задач создания всех механик сетевого сервиса с чистого листа. Часть библиотек используют протоколы верхнего уровня модели OSI, которые позволяют абстрагировать сложные структурных данных и сериализовать их при пересылке до потока битов, однако, для создания сетевого сервиса подобные решения не подходят, т.к. абстракции использующиеся при этом не способные охватить необходимую информацию. Поэтому

при создании сетевого взаимодействия сервиса с чистого листа необходим собственный протокол связи, т.е. необходимо создать набор структур, которые будут понимать и клиенты и сервера. Создание протокола - непростая задача, т.к. необходимо отказаться от заранее отлаженых механизмов верхнего уровня и создать их используя механизмы нижнего уровня. Так же, следует принять во внимание тот факт, что реализация сервиса планируется на языке C++, что накладывает ограничение на использование других языков, т.к. лишь небольшое количество из других языков программирования имеют возможность управлять инструкциями на языке C++.Ниже представлен перечень библиотек и фреймворки способных удовлетворить требованиям данного решения:

- Boost.Asio - библиотека с открытом исходным кодом для синхронного/асинхронного ввода-вывода, поддерживающая протоколы TCP/IP уровня. Используется для создания высокопроизводительных сетевых приложений. Среди преимуществ - поддержка многих протоколов, обработка ошибок, асинхронность.

- POrtable COmponents (POCO)[60] - набор библиотек с открытым исходным кодом, позволяющая создавать кросплатформенные приложения, так же поддерживает работу с протоколами TCP/IP уровня и другими. Среди ключевых возможностей - модульная архитектура, поддержка большого количества протоколов, интеграция с C++ стандартом.

- Adaptive Communication Environment (ACE)[61] - объектно-ориентированный фреймворк на C++, предоставляющий набора классов для быстрого создания высокопроизводительных приложений. Среди ключевых возможностей - реализация шаблонов проектирования, поддержка структурами многопоточности, кросплатформенность.

Появление первых версий межпроцессорного взаимодействия (IPC)[62] и сетевых коммуникация случилось после появления операционной системы UNIX версии 4.2BSD в 1983 году. Туда были добавлены новые программные сущности названные сокетами. Сокеты (от англ. - разъем) - програмнные интерфейсы для обеспечения обмена данными между процессами, процессы не обазтельно должны исполняться на одном компьютере, возможна и связь между разными компьютерами. К сокетам так же был добавлен соответсвующий API на языке C, функции которого можно было вызвать из программы. Это существующая база на основе которой работают все современные приложения и операционные системы. Дело в том, что при создании сетевого сервиса, можно было бы использовать только сокеты для создания сетевого взаимодействия и пересылке данных между устройствами в сети, однако, для создания сложной логики выполнения необходимы большие затраты времени. Текущие библиотеки, используют сокеты в своей основе, при этом достаточно абстрагируя их и предоставляя пользователям новые механизмы для их использования. Например, библиотека Boost.Asio способна создавать асинхронные задачи связи, которые пермещают буффер данных в хранилище и отправлют в тот момент, когда для выполнения есть мощности CPU.

В установлении сетевой коммуникации и пересылке данных многие моменты понятны, однако возникает вопрос с собственным протоколом связи. Существует множество протоколов связи, например, HTTP/HTTPS - протокол для передачи гипертекста или FTP - протокол для передачи файлов и т.п., однако пользоваться таким протоколом в данном решении - излишне. Необходимо опуститься ниже в модели OSI до протоколов транспортного уровня и создать простой протокол, для перессылки собственной структуры данных. Создание сетевого протокола и задача выполнения абстракций лежащих ниже по уровням в модели OSI ложится на библиотеки из списка выше. Рассмотрим их и проведем сравнительный анализ, который поможет выявить решение, которое ляжет в основу сетвого взаимодействия сервиса. Boost.Asio, POCO и ACE - три известные библиотеки для сетевого программирования написанных на языке C++ и нативно поддерживающих его. Каждая из них имеет уникальные решения, однако среди всех выделяется библиотека Boost.Asio, с использованием которой напрямую или косвенно, связаны подовляющее большенство сетевых приложений и сервисов, выделяет ее не только большое сообщество, но и гибкость, эффективность и соотвествие современным стандартам C++. Boost.Asio предоставляет модель асинхронного ввода-вывода, которая позволяет эффективно управлять сетевыми операциями и ресурсами вычислительной машины. Ключевым в асинхронном механизме является io_service, котороый координирует все асинхронные операции и управляет ресурсами, такими как сокеты и потоки, что обеспечивает высокую масштабируемость, прин низких затратах на использование абстракций вводимых Boost.Asio. В отличии от Boost.Asio, набор библиотек POCO ориентирован на пользовательские приложения работающие с существующими протоколами верхнего уровня - HTTP/HTTPS, SSL и т.п., что делает ее использование в качестве основы для создания протокола затруднительной, однако, нельзя не учесть ее удобсто в быстром создании сетевых сервисов на базе высокоуровневых протоколов. Кроме того, POCO имеет меньшее сообщество энтузиастов и ограниченную документацию, которая делает создания и поддержку сетевого сервиса с чистого листа еще более сложной задачей. Переходя к рассмотрению достоинств и недостатков фреймворка ACE, нельзя не учесть широкий набор инструментов для разработки, включая поддержку шаблонов проектирования[63][64] и компонентов безопастных к многопоточности[65][66]. Однако широкий набор инструментов делает этот фреймворк сложным в освоении. Использование данного решения приведет к работе с большим объемом кода, что может привести к непозволительному увеличению времени разработки и усложнит поддержку сетевого кода на основе данного решения. Таким образом, при создании сетевого сервиса с чистого листа библиотека Boost.Asio является единственно верным решением, на базе которого будет построен гибкий и производительный компонент системы предназначенный для сетевого взаимодействия.

Глава 2

Проектирование архитектуры сетевого сервиса

Эта глава будет полность посвящена архитектуре создаваемого сетевого сервиса для алгоритмов одновременного картографирования и локализации. В ней будут сформулированы основные требования к системе, рассмотрена диаграма классов из UML[67] для данного решения, выстроена модель сетевого взаимодействия и показаны основные моменты проектирования графического интерфейса пользователя, которым будет обладать клиентская часть сервиса. Начнем с анализа требований к системе.

2.1 Анализ требований к системе

Исходя из особенностей архитектуры системы ниже представлен струтурный подходи к анализу требования, которым система должна следовать чтобы обеспечить надежность, масштабируемость и эффективность.
Функциональные требования:

- Микросервисная архитектура - каждый сервис должен выполнять одну конкретную функцию и быть независимым от других. Масштабируемость и низкие накладные расходы на создание каждого сервиса являются приоритетом. Docker обеспечит изоляцию и портируемость таких сервисов, позволяя легко управлять их жизненным циклом.

- Поддержка обмена сообщениями с помощью ROS2 - ROS2 предоставляет механизм топиков на основе Data Distribution Service, для обмена сообщения между узлами в микросервисах и между микросервисами и роботами, пользующимися сетевым сервисом. Это также обеспечит высокую степень гибкости и масштабируемости, без особых накладных расходов.

- Асинхронная клиент-серверная связь через Boost.Asio - Boost.Asio предоставляет средства для реализации асинхронной сетевой коммуникации между микросервисами и супер-клиентом, имеющим возможность настройки

микросервисной архитектуры. Асинхронная связь позволяет эффективно обрабатывать множество одновременных соединений, что позволит использовать целые кластеры микросервисов.

- Взаимодействие с Docker через Web API[68] - Сервер управляющий микросервисами должен имет возможность управлять контейнерами через RESTful API Docker демона, обеспечивая автоматизацию процессов развертывания и мониторинга.

Нефункциональные требования:

- Производительность - система должна обеспечивать низкую задержку и высокую пропускную способность, особенно при обмене сообщениями между микросервисами и клиентом.
- Масштабируемость - ключевой фактор, архитектура должна поддерживать горизонтальное масштабирование, позволяя добавлять новые экземпляры сервисов без накладных расходов на программирование.
- Надежность и отказоустойчивость - система должна продолжать работу даже при отказе отдельных компонентов.

Таким образом, исходя из требований к системе можно перейти к этапу проектирования системы. Архитектуру системы состоит из двух основных частей, первая часть это супер-клиент с графическим интерфейсом пользователя, вторая часть - сервер взаимодействующий с микросервисами, которые в свою очередь взаимодействуют с роботом. Под супер-клиентом здесь подразумевается устройство, не являющееся роботом, в следствии чего не нуждающейся в большей части информации с микросервисов, служащее для настройки серверов по средством графического интерфейса пользователя. Сервер же, устройство, напрямую взаимодействующее с микросервисами и управляющее ими. Микросервисы - контейнера Docker, способные передовать информацию серверу через Docker Web Api и способные передавать информацию роботу по средствам FastDDS Discovery Server - а запущенного на каждом из доступных серверов. Рассмотрим структуру компонентов и их взаимодействие на рисунке:

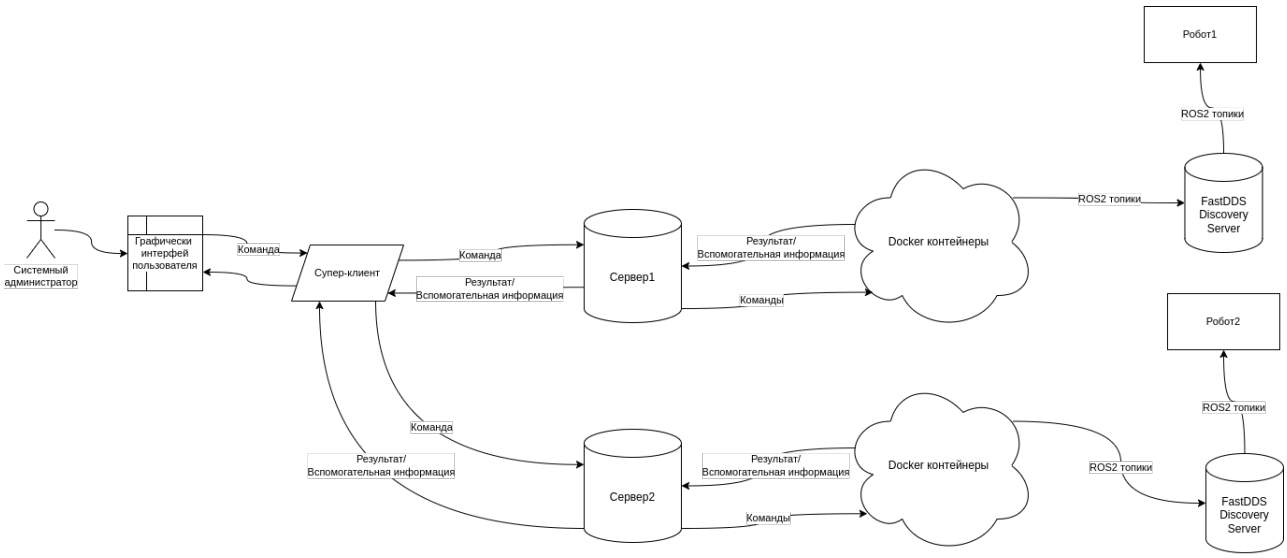


Рис. 2.1: Схема взаимодействия компонент

Из рисунка видно, что клиентская часть не берет на себя главные функции в сетевом сервисе, а лишь является инфраструктурным компонентом позволяющий пользователям легко настраивать необходимые узлы реализующие функциональность алгоритмов одновременного картографирования и локализации. Из схемы также становится понятна расписанная выше архитектура, здесь супер-клиент и сервера, реализуют клиент-серверную архитектуру, а Docker контейнера реализую микросервисную архитектуру, в которой связь между микросервисами происходит по средствам механизма топиков ROS2. Так же супер-клиентом поддержан функционал взаимодействия с несколькими серверами. Данная схема раскрывает лишь общие моменты архитектуры сетевого сервиса, передейдем к более подробному рассмотрению отдельных компонентов.

2.2 Архитектура системы

Проектирование архитектуры играет ключевую роль при построении системы, т.к. оно обеспечивает четкое понимание структуры и взаимодействия всех компонентов. Проектирование архитектуры позовляет выявить возможные проблемы и оценить возможные риски, что способствует более эффективному управлению ресурсами и сроками разработки, кроме того, хорошо продуманная архитектура облегчает масштабирование и модификации системы в будущем, что особенно важно в условиях быстро меняющегося технологического мира.

2.2.1 Архитектура сетевого взаимодействия

Вопрос архитектуры системы хотелось бы начать с клиент серверного взаимодействия между супер-клиентом и сервером. Основой является библиотека Boost.Asio способная поддерживать сеанс связи между клиентом и сервером и создавать

асинхронные задачи связи для пересылки данных. Для того чтобы пересылать данные между клиентом и сервером мало одной возможности, необходимо иметь собственный протокол связи, вариант использования готового протокола не подходит ввиду того, что такой протокол не сможет вместить в себя данные либо будет избыточен. Проектирование протокола связи необходимо начать с определения структур данных, которые будут пересылаться в ходе сетевого взаимодействия. Такого рода структуры обычно состоят из двух частей - "головы"и "тела термины "голова"употребляется не в прямом значении, а в переносном, обозначая первые несколько битов в которых кодируются вспомогательные данные. Вспомогательными данными обычно являются длины, коды и т.п. величины, которые характеризуют данное сообщение и помогают в распозновании основные данные в "теле". В нашем случае, удобно было бы реализовать подобную структуру в виде следующего класса:

```
class SerializedMessage {
public:
        enum { HeaderLength = 7 };

private:
        char header_[HeaderLength];
        char* data_;
        uint32_t body_length_;
};
```

Листинг 2.1: Структура данных "сообщение"

Так как передача структуры напрямую между двумя устройствами невозможна, то для передачи необходима процедура сериализации, которая превратит структуру данных в последовательный набор байтов. После приема данных необходима обратная сериализации процедура - десериализация, которая позволяет из потока битов восстановить структуру данных. C++ класс продемонстрированный выше служит промежуточной структурой данных в процессе сериализации/дессериализации, его структура позволяет легко поддерживать этот процесс. Во-первых, "голова"структуры - поле header_ имеет тип char* и является массивом из битов длины HeaderLength. Неизменяемость длины "головы"позволяет в процессе сериализации и дессериализации однозначно закодировать необходимые вспомогательные данные. Однозначность кодирования заключается в том, что данные закодированные внутри находяться в строго определенном порядке, задокументированном в протоколе. В процессе сетевого взаимодействия, когда поток битов проходит по сети и доходит до серверной/клиентской части сетевого сервиса, в процессе десериализации будут прочитаны именно первы HeaderLength байт сообщения, что позволит декодировать и остальную часть сообщения, следующего после "головы". Для большей наглядности, продемонструем процесс в виде рисунка:

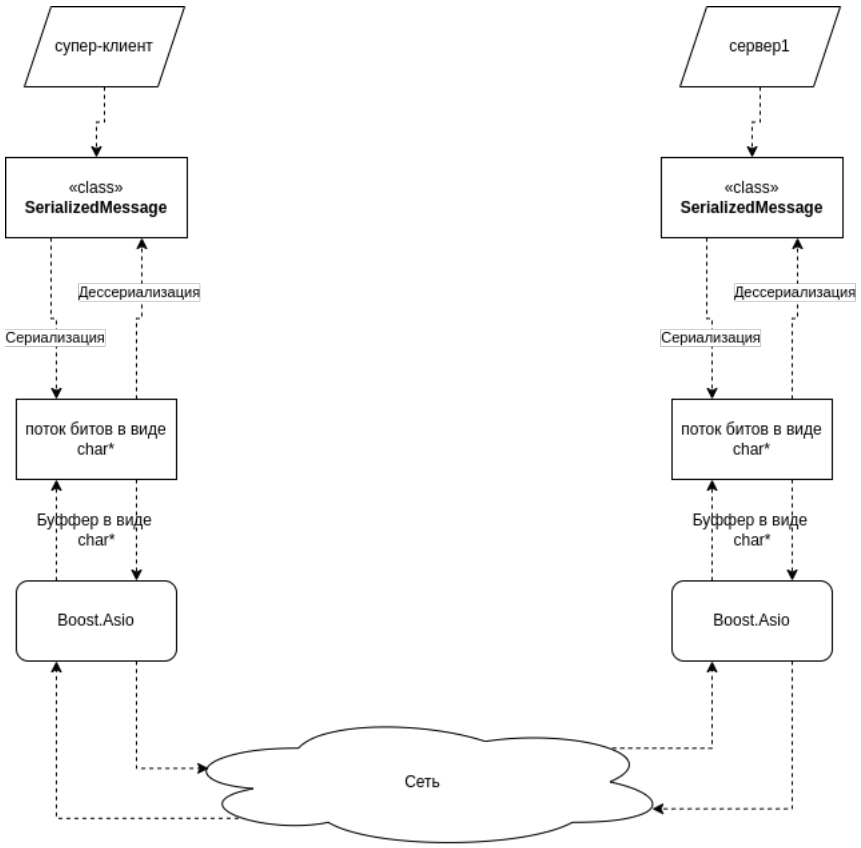


Рис. 2.2: Схема взаимодействия компонент

Модель взаимодействия между супер-клиентом раскрыта, однако, это не единственное сетевое взаимодействие реализованное в данной работе. Следующий момент касается взаимодействия сервера с микросервисами на основе Docker контейнеров. Как было раскрыто в первой главе, Docker стостоит из нескольких частей, самой главной из которых является Docker Engine, который в свою очередь состоит из демона и утилит для взаимодействия с ним. Утилит всего 3-и - командный интерфейс, web API и Docker SDK. Удобнее пользоваться командным интерфейсом, однако его использования из другого исполняемого файла затруднительно, т.к. командный интерфейс обычно уже является конечным продуктом, не допускающим постройку над ним дополнительной архитектуры, что приводит к тому, что из исполняемого файла можно вызвать процедуру командного интерфейса, однако результат работы и процесс выполнения проконтролировать невозможно. Например, при попытке вызвать команду *dockerps* из исполняемого файла, возможен вариант того, что демон Docker отключен, по той или иной причине, что приведет к тому что команда закончится неудачно, однако исполняемая программа никак об этом не узнает, т.к. у командного интерфейса остуствует возможности интеграции с другими программами. Вариант использования Docker SDK в данном случае невозможен, т.к. компания создала SDK лишь для нескольких языков, C++ к которым не принадлежит, остается использование web API, который предоставляет RESTful API по протоколу HTTP/HTTPS, которым может воспользоваться абсолютно любой клиент демона, который способен отправлять ему HTTP запросы с правильной структурой.

Замечание 1: *Software Development Kit (SDK) - набор инструментов разработки программного обеспечения, позволяющий создавать собственные решения на своей основе.*
Замечание 2: *RESTful API (полные по REST API) - стандарт поддерживающий набор правил для web API, что позволяет создавать удобные, хорошо поддерживаемые веб интерфейсы.*

Для работы с веб-интерфейсом Docker - а необходима библиотека способная отправлять HTTP запросы определенной структуры, такой библиотекой является ASL. В анализе она не упомянута, т.к. напрямую не относится к реализации сетевого сервиса, а лишь является компонентом другой библиотеки, реализующей простой SDK на основе веб-интерфейса Docker - docker_cpp. Часть функционала данной библиотеки была также реализована в этой работе. У Docker Web API являющейся веб-интерфейсом существует документация, благодаря которой и был реализован данный SDK. Для большей наглядности продемонстрируем работу сервиса с Docker демоном через веб-интерфейс ввиде рисунка:

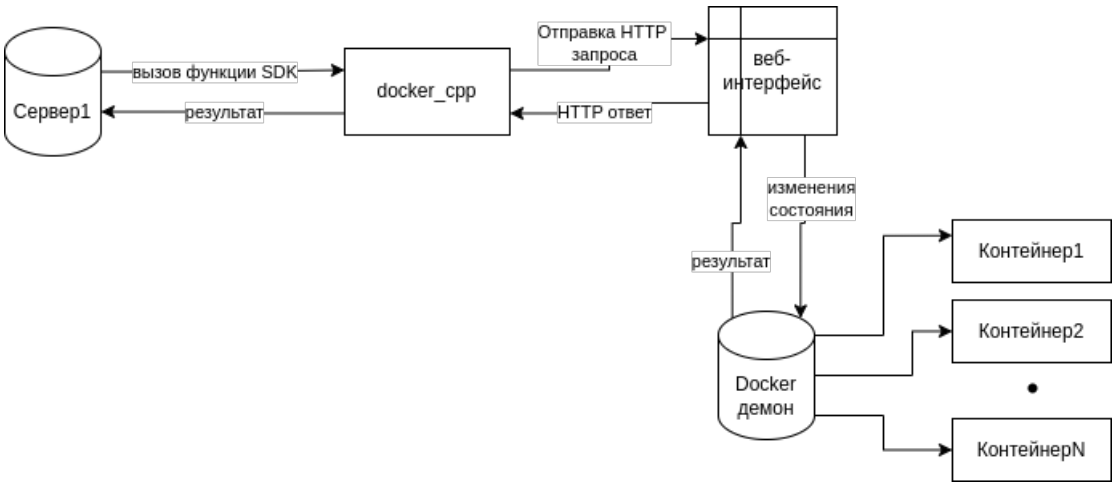


Рис. 2.3: Схема взаимодействия компонент

Последней частью сетевого взаимодействия является пересылка сообщений ROS2 между микросервисами, реализованными с использованием Docker, и пересылка сообщений между микросервисами и роботом, который является конечным потребителем обработанных сервисом данных. ROS2 благодаря использованию DDS, может создавать связь не только между узлами на одном компьютере, но может быть использован в качестве сервера для пересылки сообщений между компьютерами, разберем механизм связи по порядку. Для того, чтобы Docker контейнеры были способы пересылать сообщения между собой необходима соответствующая настройка. Самый простой вариант, настраивать каждый из создающихся контейнеров на использование сети "хоста". Таким образом, каждый контейнер запущенный на "хосте"для другого устройства будет виден как сам "хост что очень удобно, т.к. дает нам доступ в сеть без предварительной настройки сети контейнеров на самом "хосте". Следующая проблема, возникающая на

пути к запуску микросервисов с выходом в сеть и межконтейнерной связью, является механизм распределенной памяти[69] в ROS2. Дело в том, что по умолчанию, DDS платформа являющаяся основной ROS2 снижая нагрузку на сеть использует для взаимодействия между узлами распределенную память. Распределенная память является абстракцией, которая позволяет нескольким процессам в операционной системе иметь общую область памяти, по-умолчанию это запрещено. Такая общая память позволяет при пересылке не тратить процессорное время на процесс сериализации/дессериализации каждого сообщения посылаемого одним узлом другому, а просто переместить всю структуру данных в другую облась в памяти. Проблема заключается не в самой распределенной памяти, а в том, как Docker использует ресурсы операционной системы для поддержания контейнеров. По-умолчанию, Docker контейнер не может взаимодействовать с распределенной памятью основной операционной системы, что не дает узлу запущенному внутри Docker контейнера возможность пересылать сообщения, однако, в Docker существует настройка, которая дает доступ. И наконец, последняя проблема, пересылка сообщений между разными устройствами в сети. Существует два способа создания такого взаимодействия, первый - с помощью механизма подсетей в ROS2, второй - с помощью механизма Discovery Server - ов. Первый механизм эффективен, когда устройств в сети немного, тогда механизм подсетей в ROS2, с помощью мультикастинга, помогает системе находить новые узлы объединяя подсети, однако, мультикастинг - сильно нагружает сеть и зачастую выходит из строя, поэтому наиболее надежным методом пересылки сообщений между микросервисами и роботом, является механизм Discovery Server - ов. По аналогии с подобным механизмом в ROS первой версии, механизм Discovery Server создает сервер координирующий и осуществляющий перессылку сообщений между устройствами подключенными к нему.

Замечание 2: *Хост - в контексте компьютерных сетей и интернета означает любое уникальное устройство в сети имеющее собственный IP-адресс.*

2.3 Архитектура графического интерфейса пользователя

Архитектура графического интерфейса пользователя основывается на использовании функции одного из самых развитых кросплатформенных фреймоврка - Qt. У Qt есть версии, в данной работе будет использован Qt пятой версии, или сокращенно - Qt5. Рассмотрение процесса проектирования и основных моментов архитектуры графического интерфейса пользователя необходимо начать с базового понимания работы самого Qt фреймвора. В первую очередь, стоить заметить, что Qt5 - Объекто-Ориентированный фреймворк, что говорит нам о том, что вся основная функциональность в нем заключена в классах и что Qt5 - написан на C++ и большая часть его функциональности работает для этого языка программирования. Основные классы, который используют программисты при работе с

Qt5 называются виджетами. Виджеты работают по принципу матрешки, позволяя вкладывать в себя другие виджеты и т.д. Это свойство виджетов позволяет строить графический интерфейс пользователя по принципу конструктора. Виджеты являются классами в языке C++ и у них существуют определенные ограничения на взаимодействия с другими классами-виджетами, поэтому создатели Qt5 предусмотрели возможные проблемы и искусственно создали рефлексию[70] для собственного фреймворка. Рефлексия - механизм, позволяющий классам получать информацию о своей структуре и поведению во время выполнения. Рефлексия в Qt5 необходим для создания собственного механизма связи - слотов и сигналов. Механизм слотов и сигналов схож с механизмом топиков у ROS2, суть его в следующем, виджет создает слоты, которые используются в качестве реакции на действие сигналов. Например, представим класс клавиша, у класса клавиши есть слот - OnPress, допустим мы запрограммировали нашу клавиатуру таким образом, что связали сигнал Press и слот OnPress, при компиляции программы и нажатии на кнопку, сработает сигнал Press, а после него и слот клавиатуры OnPress, который выполнит некие действия, допустим выведет на экран букву соответсвующую этой клавише. Таким образом, работает механизм слотов и сигналов, связывая несколько виджетов сетями передачи сигналов. Стоит оговориться, что слоты и сигналы в C++ принимают форму методов и перечислений (enum).
Замечание 2: *Кроссплатформенность - свойство Программного Обеспечения (ПО) работать на нескольких программных платформах, так называемых, операционных системах (ОС).*

Центральным виджетом будет специальный вид виджетов, помогающий в объединений функциональности небольших приложений - MainWindow виджет, который является более абстрактным виджетом по сравнению с другими. Он берет на себя часть процессов по управлению другими виджетами, он создает верхнеуровневое меню, пустое окно приложений и имеет пустой главный виджет, который можно заполнить другими. Проще всего заполнить главный виджет с помощью менеджеров компоновки, так называемых Layout - ов, которые размещают вспомогательный виджеты в заданном порядке, обычно горизонтально или вертикально, еще есть вариант размещение в сетку. В данном решении, менеджер компоновки будет вертикальным, всего виджетов главного менеджера компоновки будет два, первый виджет, находящийся сверху, будет отображать либо информацию о текущих серверах, к которым подключен данный супер-клиент, либо информацию о микросервисах запущенных, на выбранном из перечня, сервере, второй виджет будет отображать контекстную текстовую информацию, например о записях собранных с терминала микросервиса.

Для создания первого главного виджета необходимо использовать изменение внешнего вида виджета, т.к. смена вид необходима нам для смены отображающейся информацию о серверах и информации о микросерверах. Изменять облик может специальный тип виджета, называемых stack (от англ. - куча) виджетом, подобно анимированным рекламным банерам на улицах городов, он способен менять свой внешний вид в зависимости от действий пользователя. С помощью данного виджета, можно будет реализовать мехнику "проваливания"внуть сервера.

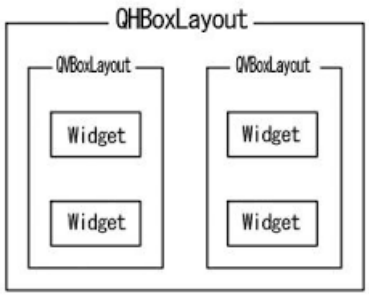


Рис. 2.4: Различные менеджеры компоновки

При установлении соединения супер-клиента и сервера, наш графический интерфейс пользователя отобразит имя сервера и его порядковый номер на одном из видов stack виджета, однако, если пользователь решит нажать на надпись, обозначающую имя сервера, то stack widget изменит свой вид и станет виджетом отображающим состояния микросервисов на данном сервере. Это и есть механика "проваливания".

Как говорилось раннее, центральный виджет, будет создавать также меню верхнего уровня. Так как графический интерфейс пользователя связан с супер-клиентом, то логичным будет добавить в меню верхнего уровня возможность подключения к серверам.

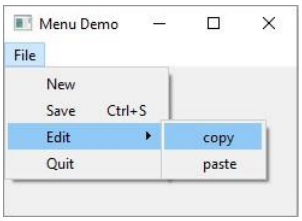


Рис. 2.5: Пример меню верхнего уровня

При инициировании создании сервера через меню верхнего уровня, необходимым будет создания диалогового окна, позволяющего пользователю супер-клиента ввести необходимый ip-адрес и порт сервера чтобы установить с ним соединение.

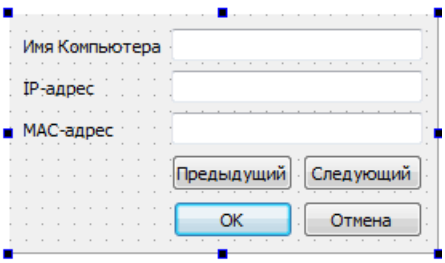


Рис. 2.6: Пример диалогового окна

После завершения процедуры подключения супер-клиента к серверу, как говорилось выше, необходимо отразить состояние микросервисов, предназначенныех для запуска алгоритмов одновременного картографирования и локализации. Так как микросервисы в работе будут построены на основе Docker, необходимо учитывать аспекты связанные с этим. Поскольку сетевой сервис предполагает легкую настройку сервера для работы с алгоритмами, то необходимо добавить возможность работать с еще не собранными контейнерами в виде Dockerfile. Таким образом на сервере, для создания всей инфраструктуры из микросервисов потребуется хранить лишь один Dockerfile с инструкциями по сборке. Кроме того, после сборки необходимо отображать уже собранные контейнеры, которые готовы к запуску. Само собой необходимо учитывать и информацию о уже запущенных контейнерах. Это подводит нас к тому что, мы должны создать 3 виджета, отображающих полную информацию о состояниях готовности микросервисов - не собран, собран, запущен, также нам необходимо построить структуру работы с Dockerfile и launch файлами, поскольку именно они буду создавать и запускаться в микросервисах.

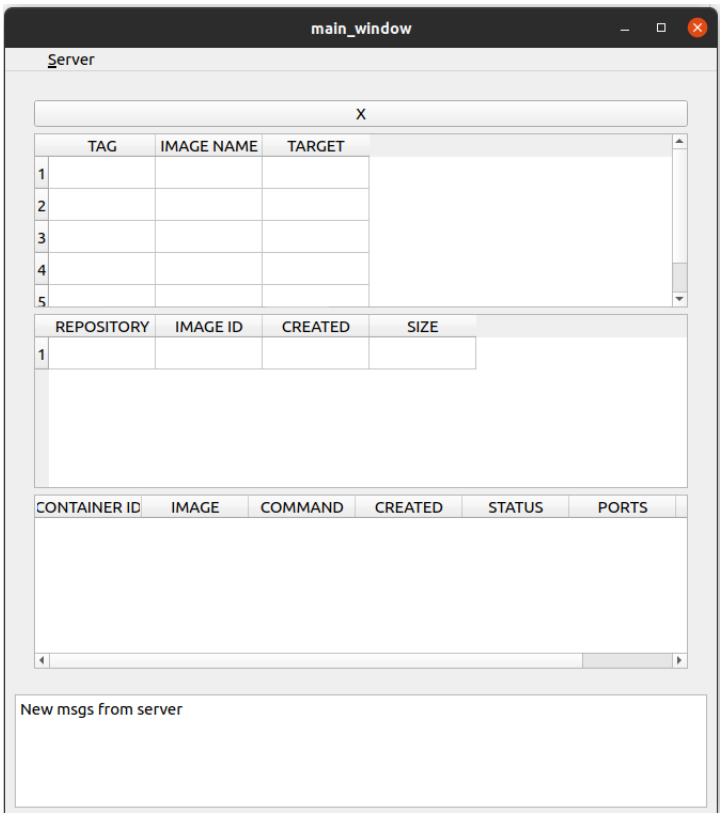


Рис. 2.7: Пример работы с возможностями микросервисов

В первом из трех виджетов, призванных отражать информацию о микросервисах, будет хранится информация из некоего текстового файла, хранящегося на сервере. В данном файле, должна быть отражена информация о микросервисах, которые можно собирать и использовать, кроме того для создания микросервисов необходимо вручную формировать .tar.gz архив в который необходимо добавлять Dockerfile. Это необходимо, т.к. создание контейнеров в Docker с использованием Web API невозможно напрямую с Dockerfile, необходимо посылать демону Docker

через API специализированный запрос вместе с сериализованным .tar.gz архивом, только в этом случае он способен собрать образ контейнера. Кроме .tar.gz, файла с перечнем образов, Dockerfile необходимых в процессе создания микросервисов, нам необходимо создать инфраструктуру для запуска этих микросервисов, т.к. они используют ROS2 возможности, то и запускать в автоматическом режиме их проще всего с помощью launch файлов. Каждый из контейнеров будет запускать launch файлы, которые, в свою очередь, будут запускать определенные узлы необходимые для корректной работы алгоритмов одновременного картографирования и локализации. Это достаточно распространненая практика среди компании строяющих свои робототехнические продукты, облегчающая их масштабирование. Для реализации подобной структуры лучше всего подходят директории компьютера. Примерная структура директорий показана на рисунке ниже:

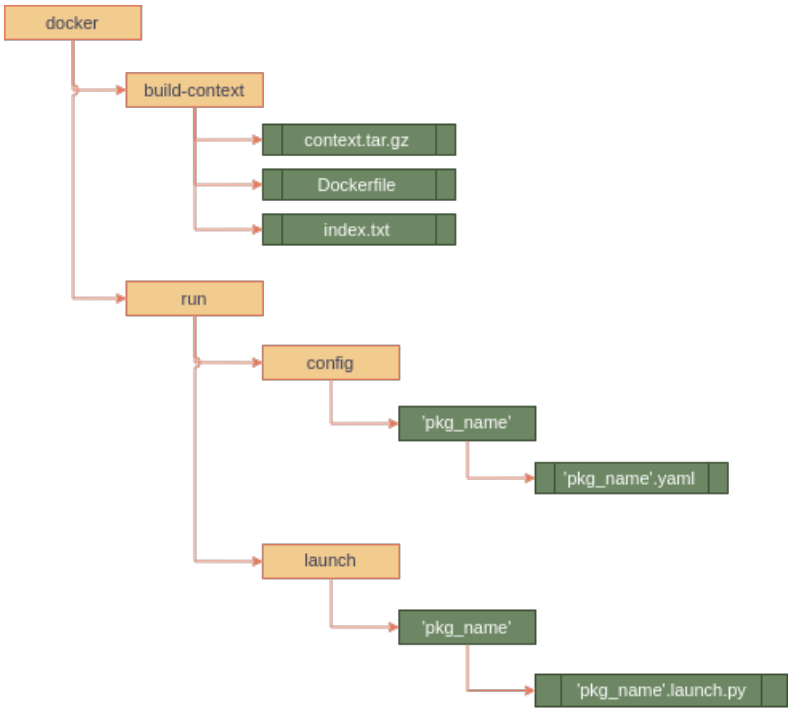


Рис. 2.8: Структура файлов для обслуживания микросервисов

2.3.1 Диаграмма классов UML

Unified Modeling Language (UML) - стандартный язык моделирования, используемый для визуализации, спецификации и документирования программных систем. В языке есть набор графические диаграмм, которые позволяют отразить различные стороны архитектуры приложений. Для создания архитектуры сетевого сервиса, в процессе работы была создана диаграмма классов UML, которая представляет собой структуру системы, отображающая классы и их внутренее устройство, а также связи между ними. Диаграмма классов для данного решения проблематично разместить в работе, поэтому необходимо разбиение общей диаграммы на несколько слабозависимых частей. Первая часть диаграммы затраги-

вает устройство обработки запросов в клиент-серверной связи, она представлена ниже:

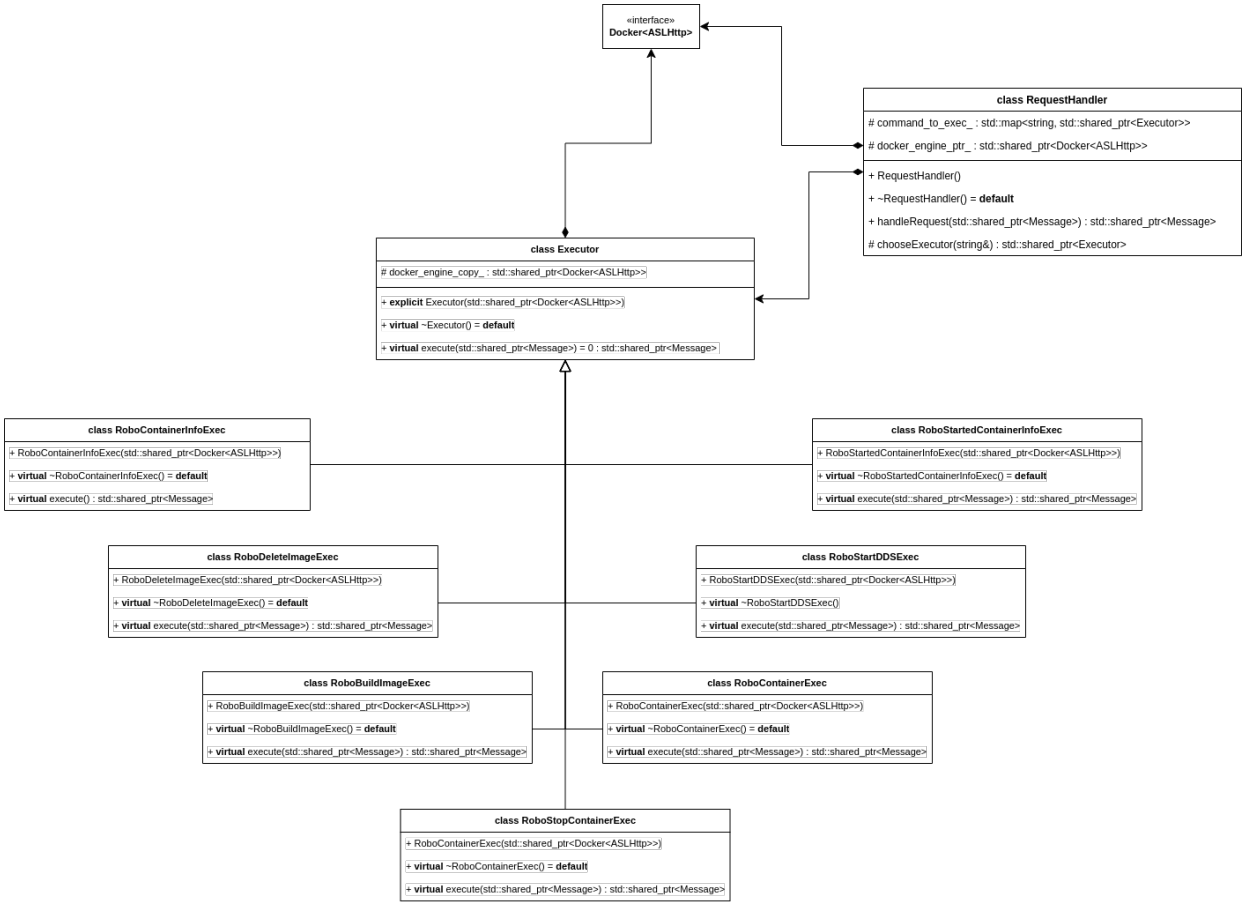


Рис. 2.9: Обработка запросов сервера

Данная часть отражает работу функционала ответсвенного за обработку десериализованных запросов от супер-клиента к серверу. Здесь класс RequestHandler получает запрос и переедадрессует его одному из классов-наследников класса Executor, ответственного за исполнения запросов. Внутри классов-насследников Executor - а, находятся вызов функции docker_cpp библиотеки, которая реализует C++ API для вызова функции Web API Docker. Результат выполнения функций валидируется, в случае успешного выполнения обратно формируется сообщение-ответ на запрос, в котором находятся необходимые сведения. В случае неудачи, классы-наследники Executor так же формируют сообщение, отражающее причину ошибки и отправляют ее супер-клиенту, который обрабатывает ее и выдает результат. Вторая часть данной диаграммы отражает устройство сетевого взаимодействие серверной и клиенской части сетевого сервиса:

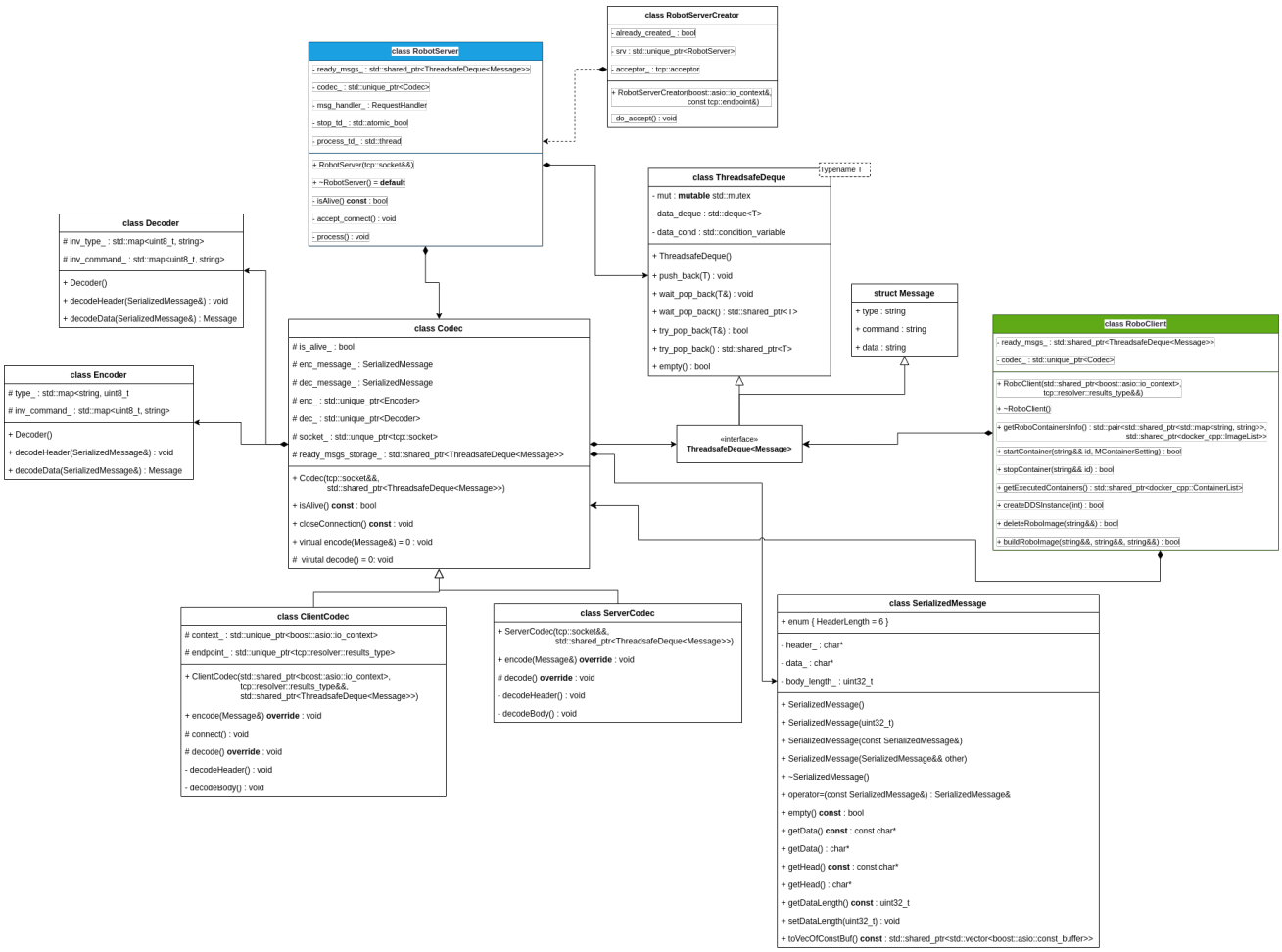


Рис. 2.10: Сетевое взаимодействия сервера и супер-клиента

Как видно из диаграммы классы реализующие абстракции сервера и супер-клиента построенны на базе класса Codec, который реализует само сетевое взаимодействие на основе библиотеки Boost.Asio. Класс Codec - базовый класс для ClientCodec и ServerCodec, каждый из которых имеет свои отличительные особенности, помогающие им в реализации функционала связи. Так же есть классы SerializedMessage, о которых было кратко написано выше, и класс Message, эти классы реализуют абстракции сериализованного и дессериализованного сообщения, а сам процесс происходит внутри классов Decoder и Encoder, которые переводят информацию между двумя классами сообщений. Процесс обработки запроса и подготовки ответа требует многопоточности. Многопоточность здесь разделена между классами RobotServer, реализующего сервер, и классом Codec равно как и между классом RoboClient и Codec. Для отправки сообщения, сервер или клиент формирует его внутри методов своего класса, однако для того, чтобы принять и обработать сообщение, ему необходимо ждать неопределенное время. Для работы с неопределенным временем существует класс ThreadsafeDeque, являющийся потокобезопастной структурой данных, которая синхронизирует классы сервера или клиента и классы-наследники класса Codec. Во время передачи сервером или клиентом запроса в виде сообщения в метод encode класса Codec, они должны использовать метод push_back класса ThreadsafeDeque, в котором, спустя время будет

сформирован ответ, таким образом, во время вызова метода push_back из класса RobotServer или RoboClient поток выполнения в них блокируется внутренним мьютексом принадлежащем ThreadsafeDeque. Как только ответ придет, классом Codec к экземпляру класса ThreadsafeDeque, который она разделяет с RobotServer или RoboClient, будет добавлено дессериализованное сообщение и поток выполнения вновь вернется в классы сервера или клиента. Третья и последняя часть диаграммы отражает структуру графического интерфейса пользователя принадлежащего клиентской части сетевого сервиса:

Замечание 2: *Потокобезопастность - свойство структур данных, позволяющее им работать с несколькими программными потоками, не вызывая опасных состояний гонки или взаимоблокировки.*

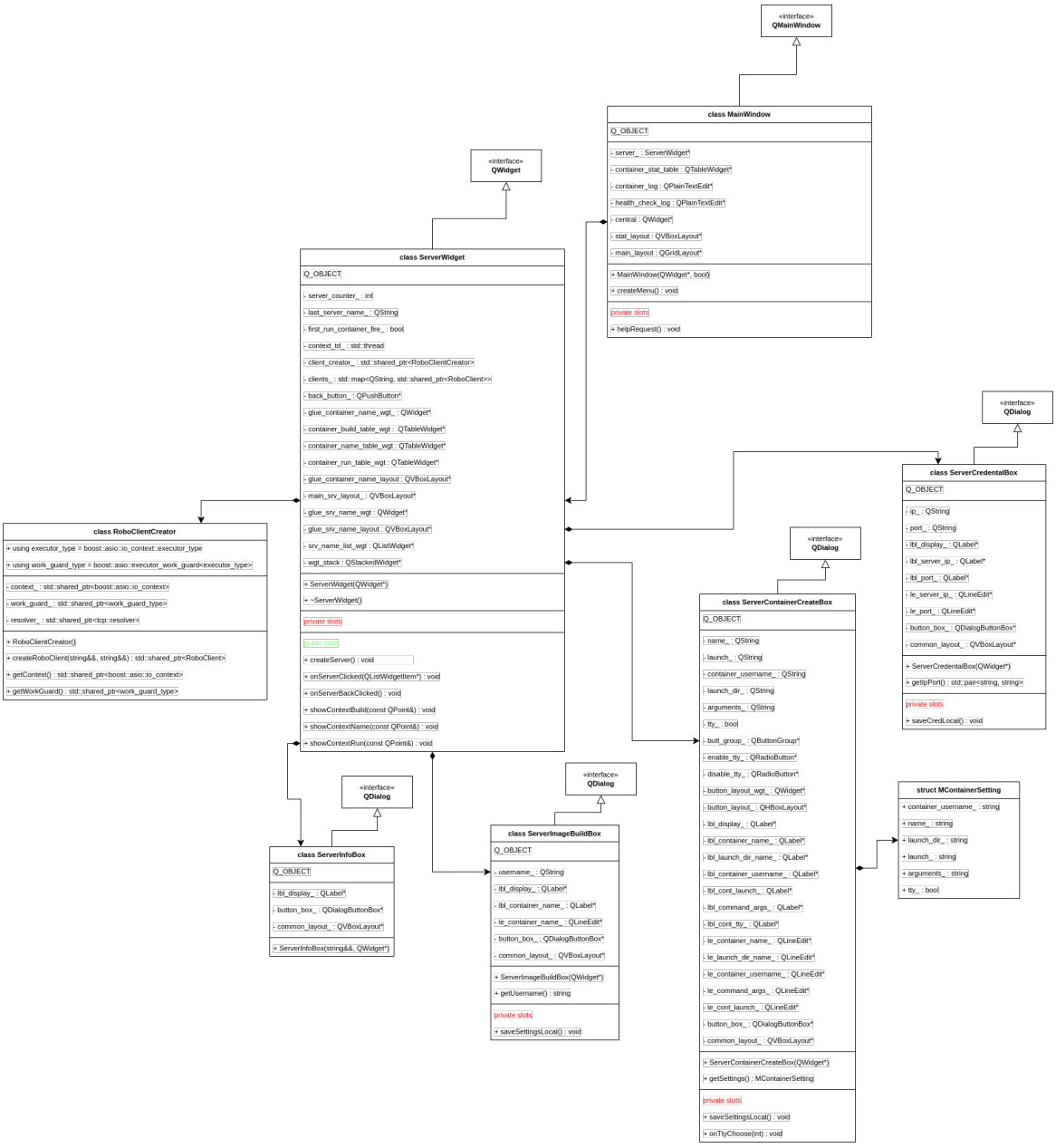


Рис. 2.11: Графический интерфейс пользователя

В данной схеме находится класс MainWidget, о котором говорилось выше, в ча-

сти про архитектуру графического интерфейса. Он содержит в себе главного менеджера компоновки, который добавляет в себя классы-виджеты ServerWidget использующий stack виджет для реализации функциональности управления серверами и микросервисами, а также QPlainTextEdit виджет, который способен отображать текстовую информацию. ServerWidget в свою очередь содержит управление созданием класса супер-клиента реализованого в виде класса RoboClientCreator, а также класы ServerInfoBox, ServerImageBuildBox, ServerContainerCreateBox и ServerCredentialBox, которые управляют всплывающими окнами, информирующими пользователя о различных произошедших ситуациях или требующих от пользователя ввода данных. Например, класс ServerCredentialBox является всплывающим окном, которое отображается в случае добавления пользователем нового сервера и требующем от пользователя данных необходимых для подключение - ip-адресса и порта.

Глава 3

Реализация сетевого сервиса

В данной главе будут показаны основные моменты реализации сетевого сервиса с примерами из программного кода решения. Также будут затронуты моменты касающиеся особенностей использования функции библиотек для сетевого взаимодействия и создания графического интерфейса.

3.1 Разработка сетевого взаимодействия

Сетевое взаимодействие, как было написано ранее, является ключевой механикой данной работы. Практически все взаимодействие между компонентами сетевого сервиса происходит с использованием сетевых технологий. Вопрос выбора библиотек для реализации сетевого взаимодействия, был рассмотрен в первой главе данной работы, в ходе которого стало очевидным преимущество библиотеки Boost.Asio для создания подобного рода системы. Хотелось бы начать с рассмотрения протокола для сетевого взаимодействия, процесса сериализации/дессериализации и создания асинхронных задач связи.

3.1.1 Создания инфраструктуры связи

Прежде чем начать обмен данными между устройствами в сети, необходимо наладить инфраструктур, которая будет отвечать за установку и поддержку этого взаимодействия. Рассмотрим класс RobotServerCreator ответсвенный за начало сетевого взаимодействия. Он также реализует функции создания сервера:

```
1  class RobotServerCreator {
2          public:
3                  RobotServerCreator(boost::asio::io_context& io_context, const tcp::endpoint&
                       endpoint);
4                  ~RobotServerCreator();
5
6          private:
7                  void do_accept();
8
9          private:
10                 bool already_created_;
11
```

```
12              std::unique_ptr<RobotServer> srv;
13              tcp::acceptor acceptor_;
14      };
```

Листинг 3.1: Заголовочный файл класса RobotServerCreator

```
1  RobotServerCreator::RobotServerCreator(boost::asio::io_context& io_context, const tcp::
       endpoint& endpoint)
2  : already_created_(false),
3          acceptor_(io_context, endpoint)
4  {
5          do_accept();
6  }
7
8  RobotServerCreator::~RobotServerCreator()
9  {
10         if (srv != nullptr)
11         srv->shutdown();
12         srv.reset();
13 }
14
15 void RobotServerCreator::do_accept() {
16         acceptor_.async_accept([this](boost::system::error_code ec, tcp::socket socket) {
17         if(!ec && !already_created_)
18         {
19         srv = std::make_unique<RobotServer>(std::move(socket));
20         already_created_ = true;
21         }
22
23         do_accept();
24         });
25 }
```

Листинг 3.2: Определение методов класса RobotServerCreator

Из заголовочного файла видно, что класс использует std::unique_ptr<RobotServer> для хранения экземпляра класса, реализующего сервер. Это необходимо для того, чтобы недопустить создание нового объекта этого класса, т.к. должен поддерживаться инвариант существования только одного сервера на компьютере. Конструктор класса принимает ссылки на объекты классов boost::asio::io_context и boost::asio::ip::tcp::endpoint. Объект класса boost::asio::io_context реализует программную сущность позволяющую осуществлять перессылку сообщений, являясь основным классом библиотеки Boost.Asio для поддержания асинхронной передачи данных. Класс boost::asio::ip::tcp::endpoint является реализацией структуры данных, ответсвенной за хранения учетных данных сервера, таких как ip-адресс и порт. В конструкторе эти данные используются для создания сущности boost::asio::ip::tcp::acceptor, которая принимает входящие подключения к серверу. Далее идет вызов метода do_accept, внутри которого используется метод объекта accetor_, являющегося менеджером подключений, функция которого заключается в установлении соедниения с клиентом. В случае успеха, создается объект boost::asio::ip::tcp::socket, который является абстракцией сетевого соединения. В дальнейшем, с помощью данного сетевого сокета станет возможным передача данных между устройствами в сети. Также, в случае успешной установки сетевого соединения, вызывается лябмда-функция, переданная методу async_accept, внутри

которой реализована проверка факта создания сервера, если проверка пройдена, то создается новый сервер.

Почти идентичным, за исключением пары моментов, является создание инфраструктуры связи с клиентской стороны сервиса.

```
class RoboClientCreator {
	public:
		using executor_type = boost::asio::io_context::executor_type;
		using work_guard_type = boost::asio::executor_work_guard<executor_type>;
	public:
		RoboClientCreator();

		std::shared_ptr<RoboClient> createRoboClient(std::string& ip,std::string& port);
		std::shared_ptr<boost::asio::io_context> getContext();
		std::shared_ptr<work_guard_type> getWorkGuard();

	private:
		std::shared_ptr<boost::asio::io_context> context_;
		std::shared_ptr<work_guard_type> work_guard_;
		std::shared_ptr<tcp::resolver> resolver_;
	};
```

Листинг 3.3: Заголовочный файл класса RoboClientCreator

```
RoboClientCreator::RoboClientCreator()
: context_(std::make_shared<boost::asio::io_context>()),
  work_guard_(std::make_shared<work_guard_type>(boost::asio::make_work_guard(*context_))),
  resolver_(std::make_shared<tcp::resolver>(*context_))
{ }

std::shared_ptr<RoboClient>
RoboClientCreator::createRoboClient(std::string& ip, std::string& port)
{
	auto&& endpoint = resolver_->resolve(ip.c_str(), port.c_str());

	return std::make_shared<RoboClient>(context_, std::move(endpoint));
}

std::shared_ptr<boost::asio::io_context>
RoboClientCreator::getContext()
{
	return context_;
}

std::shared_ptr<RoboClientCreator::work_guard_type>
RoboClientCreator::getWorkGuard()
{
	return work_guard_;
}
```

Листинг 3.4: Определение методов класса RoboClientCreator

Поговорим про различия, для клиентов разрешено создание нескольких экземпляров класса RoboClient реализующего функциональность клиента, т.к. мы должны поддерживать инвариант, позволяющий одному супер-клиенту взаимодействовать с несколькими серверами сразу. Для создания одного из клиентов, необходимо использовать учетные данные, такие как ip-адресс и порт сервера, к которому подключается данный клиент, эти данные в методе createRoboClient мы

передаем объекту resolver_, являющимся объектом класса boost::asio::ip::tcp::resolver, суть которого преобразование адресса сервера в ip-адресс используя DNS сервер. Грубо говоря, это перевод адреса, заданного с помощью строковых литералов в ip-адрес, заданный с помощью цифр и точек. Результатом работы метода данного объекта, является tcp::endpoint, который, как говорилось ранее, хранит учетные данные подключения. Также стоит отметить тот факт, что пересылка данных возможна лишь после вызова метод run() объекта класса boost::asio::io_context. Данная операция является блокирующей, она останавливает поток управления, что приводит к невозможности выполнения программных инструкций следующих ниже по коду. Это создает проблему в графическом интерфейсе пользователя, т.к. именно в классе ServerWidget, ответственным за реализацию графических отображений управления, идет работа с группой созданных объектов класса RoboClient. Такая проблема вынуждает создавать отдельный поток для использования метода run() объекта класса boost::asio::io_context.

Данный классы являются лишь "фабрикой"для создания объектов клиента и сервера. Они выступают в качестве объектов хранящих разделяемое клиентами и серверами состояния. В данном случае, они хранят boost::asio::io_context и т.п. вещи, которые являются общими для всех объектов клиентов и серверов. Манипуляцией потоками данных, в данном случае, занимаются классы реализующие клиент и сервер.

Рассмотрим реализацию сервера:

```
class RobotServer {
	public:
		RobotServer(tcp::socket&& socket);
		~RobotServer();

		void shutdown();

	private:
		void isAlive();
		void accept_connect();
		void process();

	private:
		std::shared_ptr<ThreadsafeDeque<Message>> ready_msgs_;
		std::unique_ptr<Codec> codec_;

		RequestHandler msg_handler_;

		std::atomic_bool stop_td_;
		std::thread process_td_;
};
```

Листинг 3.5: Заголовочный файл класса RobotServer

```
RobotServer::RobotServer(tcp::socket&& socket)
: ready_msgs_(std::make_shared<ThreadsafeDeque<Message>>()),
  codec_(std::make_unique<ServerCodec>(std::move(socket), ready_msgs_)),
  stop_td_(false)
{
	process_td_ = std::thread(&RobotServer::process, this);
}

```

```
void RobotServer::shutdown() {
	stop_td_ = true;
	ready_msgs_->shutdown();

	if (process_td_.joinable())
	{
		process_td_.join();
	}

	codec_->closeConnection();
}

RobotServer::~RobotServer() {
	ready_msgs_.reset();
	codec_.reset();
}

void RobotServer::process() {
	while(!stop_td_)
	{
		std::shared_ptr<Message> request = ready_msgs_->wait_pop_back();

		if (request != nullptr)
		{
			std::shared_ptr<Message> resp = msg_handler_.handleRequest(request);
			codec_->encode(*resp);
		}
	}
}

void RobotServer::isAlive() {
	if (!codec_->isAlive())
	{
		stop_td_ = true;
		throw std::runtime_error("Our connection to client is die,
		 terminate RobotServer!");
	}
}
```

Листинг 3.6: Определение методов класса RobotServer

В конструкторе, абстракция сервера, реализованного в виде класса RobotSever, создает объект класса std::shared_ptr<ThreadsafeDeque<Message», являющегося одновременно хранилищем для сообщений-ответов и структурой, синхронизирующей процесс запрос-ответ. Так же в конструкторе создается объект класса std::shared_ptr<ServerCodec>, принимающий сокет и хранилище, необходимый для управления сетевым взаимодействием. Внутри конструктора создается объект std::thread, которому передается метод класса и указатель на этот класс, в котором происходит основная логика процесса запрос-ответ. В цикле проверяющем условие остановки сервиса, реализованном на атомарном типе bool, с помощью объека stop_td_, приосходит ожидание сервером запроса, реализованого в форме вызова метода wait_pop_back() хранилища. Как уже было упомянуто, данное хранилище выступает в роли структуры данных синхронизирующей процесс запрос-ответ, путем приостановки потока выполнения на методе wait_pop_back(), который является блокирующим. Синхронизация принимает форму блокировки потока, т.к. состояния гонки в функции невозможны, однако возможны большие

накладные расходы ресурсов на работу цикла сервера в "холостую". Таким образом сервер "дожидается"запроса на действия от супер-клиента. После отправки супер-клиентом запроса, происходит разблокировка потока выполнения и переход к следующим инструкциям, которые формируют "ответ"сервера. Формирование "ответа"происходит с использованием функций класса RequestHandler, однако, это выходит за рамки реализации задачи сетевого взаимодейсвия и будет рассмотрено в дальнейшем. Рассмотрим реализацию клиента:

```
class RoboClient {
       public:
              RoboClient(std::shared_ptr<boost::asio::io_context> context,
                         tcp::resolver::results_type&& endpoint);
              ~RoboClient();

              std::pair<std::shared_ptr<std::map<std::string, std::string>>,
                        std::shared_ptr<docker_cpp::ImageList>>
              getRoboContainersInfo();
              bool startContainer(std::string&& id, MContainerSetting settings);
              bool stopContainer(std::string&& id);
              std::shared_ptr<docker_cpp::ContainerList> getExecutedContainers();
              bool createDDSInstance(int id);
              bool deleteRoboImage(std::string&& id);
              bool buildRoboImage(std::string&& name, std::string&& target, std::string&&
                  username);
              bool getRoboLog(std::string&& id, std::string& log);

       private:
              std::shared_ptr<ThreadsafeDeque<Message>> ready_msgs_;

              std::unique_ptr<Codec> codec_;
};
```

Листинг 3.7: Заголовочный файл класса RoboClient

```
RoboClient::RoboClient(std::shared_ptr<boost::asio::io_context> context,
                       tcp::resolver::results_type&& endpoint)
: ready_msgs_(std::make_shared<ThreadsafeDeque<Message>>()),
  codec_(std::make_unique<ClientCodec>(context, std::move(endpoint), ready_msgs_))
{ }

RoboClient::~RoboClient()
{
       codec_->closeConnection();
}

std::pair<std::shared_ptr<std::map<std::string, std::string>>,
         std::shared_ptr<docker_cpp::ImageList>>
RoboClient::getRoboContainersInfo()
{
       Message msg;
       msg.type = "Request";
       msg.command = "getRoboContainerInfo";
       msg.data = "";

       codec_->encode(msg);

       std::shared_ptr<Message> resp = ready_msgs_->wait_pop_back();

       if (resp->result)
       {
```

```
27                size_t delim = resp->data.find(" ");
28                std::string tag_map = resp->data.substr(0, delim);
29                std::string image_info_list = resp->data.substr(delim + 1);
30
31                json m = json::parse(tag_map);
32                json im = json::parse(image_info_list);
33                return {std::make_shared<std::map<std::string, std::string>>(
34                                m.get<std::map<std::string, std::string>>()),
35                        std::make_shared<docker_cpp::ImageList>(
36                                im.get<std::vector<docker_cpp::ImageInfo>>())};
37        }
38        else
39        {
40                std::cerr << resp->data << std::endl;
41                return {nullptr, nullptr};
42        }
43}
44
45bool RoboClient::startContainer(std::string&& id, MContainerSetting settings)
46{
47        Message msg;
48        msg.type = "Request";
49        msg.command = "execRoboContainer";
50
51        docker_cpp::CreateConfig conf;
52
53        conf.Image = id.substr(7);
54        conf.Tty = settings.tty_;
55        conf.OpenStdin = true;
56        conf.NetworkDisabled = false;
57        conf.Entrypoint.push_back("bash");
58        conf.Entrypoint.push_back("-c");
59        std::string command = settings.launch_dir_ + ".launch.py " + settings.arguments_;
60        conf.Cmd.push_back(command);
61        conf.hostConfig.networkMode = "host";
62        conf.hostConfig.autoRemove = false;
63        conf.hostConfig.privileged = true;
64        conf.hostConfig.ipcMode = "host";
65        conf.hostConfig.readonlyRootfs = false;
66
67        json j = conf;
68
69        msg.data = settings.launch_dir_ + " " + settings.name_ + " " + j.dump();
70
71        codec_->encode(msg);
72
73        std::shared_ptr<Message> resp = ready_msgs_->wait_pop_back();
74
75        if (!resp->result)
76        {
77                std::cerr << resp->data << std::endl;
78                return false;
79        }
80
81        return true;
82}
83
84bool RoboClient::stopContainer(std::string&& id)
85{
86        Message msg;
87        msg.type = "Request";
```

```
88          msg.command = "stopRoboContainer";
89          msg.data = std::move(id);
90
91          codec_->encode(msg);
92
93          std::shared_ptr<Message> resp = ready_msgs_->wait_pop_back();
94
95          if (!resp->result)
96          {
97                  std::cerr << resp->data << std::endl;
98                  return false;
99          }
100
101         return true;
102 }
103
104 std::shared_ptr<docker_cpp::ContainerList> RoboClient::getExecutedContainers()
105 {
106         Message msg;
107         msg.type = "Request";
108         msg.command = "getExecutedRoboContainer";
109         msg.data = "";
110
111         codec_->encode(msg);
112
113         std::shared_ptr<Message> resp = ready_msgs_->wait_pop_back();
114
115         if (resp->result)
116         {
117                 json j = json::parse(resp->data);
118                 return std::make_shared<docker_cpp::ContainerList>(
119                         j.get<std::vector<docker_cpp::ContainerInfo>>());
120         }
121         else
122         {
123                 std::cerr << resp->data << std::endl;
124                 return nullptr;
125         }
126 }
127
128 bool RoboClient::createDDSInstance(int id)
129 {
130         Message msg;
131         msg.type = "Request";
132         msg.command = "execDDSServer";
133         msg.data = std::to_string(id);
134
135         codec_->encode(msg);
136
137         std::shared_ptr<Message> resp = ready_msgs_->wait_pop_back();
138
139         if (!resp->result)
140         {
141                 std::cerr << resp->data << std::endl;
142                 return false;
143         }
144
145         return true;
146 }
147
148 bool RoboClient::deleteRoboImage(std::string&& id)
```

```
{
	Message msg;
	msg.type = "Request";
	msg.command = "deleteRoboImage";
	msg.data = id.substr(7);

	codec_->encode(msg);

	std::shared_ptr<Message> resp = ready_msgs_->wait_pop_back();

	if (!resp->result)
	{
		std::cerr << resp->data << std::endl;
		return false;
	}

	return true;
}

bool RoboClient::buildRoboImage(std::string&& name, std::string&& target, std::string&&
    username)
{
	Message msg;
	msg.type = "Request";
	msg.command = "buildRoboImage";
	msg.data = std::move(name);
	msg.data += " ";
	msg.data += std::move(target);
	msg.data += " ";
	msg.data += std::move(username);

	codec_->encode(msg);

	std::shared_ptr<Message> resp = ready_msgs_->wait_pop_back();

	if (!resp->result)
	{
		std::cerr << resp->data << std::endl;
		return false;
	}

	return true;
}

bool RoboClient::getRoboLog(std::string&& id, std::string& log)
{
	Message msg;
	msg.type = "Request";
	msg.command = "getRoboLog";
	msg.data = std::move(id);

	codec_->encode(msg);

	std::shared_ptr<Message> resp = ready_msgs_->wait_pop_back();

	size_t delim = resp->data.find(" ");
	std::string result = resp->data.substr(0, delim);
	log = resp->data.substr(delim + 1);

	if (!resp->result)
	{
```

```
209                 std::cerr << resp->data << std::endl;
210                 return false;
211         }
212         else
213         {
214                 log = resp->data;
215                 return true;
216         }
217 }
```

Листинг 3.8: Определение методов класса RoboClient

Реализация класса клиента, ощутимо сложнее чем реализация класса сервера. Все дело в том, что над классом сервера проводится минимальное количество манипуляций, в то время, как методы класса клиента будет постоянно использовать классы связанные с графическим интерфейсом пользователя, поэтому выгодно иметь развитый класс клиента и закрытый класс сервера. Клиента также реализует модель процесса запрос-ответ, что заставляет его создавать и хранить класс std::shared_ptr<ThreadsafeDeque<Message», который он создает в конструкторе и объект класса std::shared_ptr<Codec>, который также создается в конструкторе. В качестве аргументов для создания объекта класса ClientCodec, необходимо передавать контекст и учетные данные сервера к которому идет подключение, из этих данных в конструкторе класса будет создан сокет, который, как говорилось выше, является абстракцие соединения устройств по сети. Так же, в конструктор класса ClientCodec передается хранилище. Формирование запроса происходит в методах класса клиента, каждый метод ответсвеннен за определенный запрос. Рассмотрим структуру на примере метода getRoboContainerInfo(). Первым идет создание запроса, принимающий форму объекта класса Message, структура которого будет рассмотрена ниже. Затем идет отправка запроса серверу и вызов блокирующей операции, которая ожидает ответа от сервера. В зависимости от успеха выполнения, идет либо выдача результата графическому интерфейсу, либо передача ошибки в стандартный поток вывода ошибок. Обработка результата происходит непосредственно в методах класса клиента, в отличии от класса сервера, в котором обработки результата как такого не требуется.

3.1.2 Пересылка данных между устройствами

Для пересылки библиотека Boost.Asio использует, так называемую, асинхронную задачу связи которая создается с помощью C++ функций boost::asio::async_write и boost::asio::async_read:

```
1       boost::asio::async_write(*socket_, *enc_message_.toVecOfConstBuf(),
2       [this](boost::system::error_code ec, size_t)
3       {
4         if(ec)
5         {
6           is_alive_ = false;
7         }
8         else
9           std::cout << "encode error: " << ec.message() << std::endl;
10      }
```

```
11      );
```

Листинг 3.9: Вызов функции async_write

```
1       boost::asio::async_read(*socket_, boost::asio::buffer(dec_message_.getHead(),
            SerializedMessage::HeaderLength),
2       [this](boost::system::error_code ec, size_t)
3       {
4         if(!ec)
5         {
6           std::cout << "In decodeHeader() func" << std::endl;
7           dec_->decodeHeader(dec_message_);
8           std::cout << "temp message size: " << dec_message_.getDataLength() << std::endl;
9           decodeBody();
10        }
11        else
12        {
13          is_alive_ = false;
14          std::cout << ec.message() << std::endl;
15upquote           upquote//upquote upquotesocket_upquote->upquotecloseupquote()upquote;
16        }
17      }
18    );
```

Листинг 3.10: Вызов функции async_read

Функции нужны для записи в сетевой сокет и чтения из сетевого сокета соответсвенно. Обе функции принимают в качестве аргумента, сетевой сокет. Функция async_write принимает вторым аргументом буффер данных в виде структуры boost::asio::const_buffer, которая хранит в себе последовательный массив данных, третьим же аргументом передается лямбда-функция, принмающая на себя роль функции, вызывающейся после завершения выполнения передачи данных. Внутри лямбда-функции становится доступным объект ec, класса boost::system::error_code, который хранит в себе возможные ошибки, произошедшие во время передачи данных, путем обработки объекта ec, можно реализовать в лямбда- функции подобие проверки на ошибки. Функция async_read принимает вторым аргументов объект boost::asio::buffer, в который будет записан переданный поток данных, в виде потока байт, третьим аргументом предается лямбда-функция, аналогичная по структуре лябмда-функции использующейся в async_write функции, однако в случае функции async_read, обязательно должен происходить рекурсивный вызов, т.к. без этого не произойдет процесса "ожидания"следующего потока данных. Более подробно рассмотрено про функцию async_read будет рассказано ниже.

Как было замечено ранее, класс сервера и класс клиента, хранят и взаимодействуют лишь с верхнеуровневой логикой Boost.Asio, основные операции вынесены в отдельную группу классов - Codec. Для клиентской и серверной части приложения имеется собственный класс-наследник класса Codec, реализующий функции необходимые клиентской и серверной части. Рассмотрим определения этих классов, а также их класса-прародителя и разберем их реализацию:

```
1 class ClientCodec : public Codec {
2       public:
3               ClientCodec(std::shared_ptr<boost::asio::io_context> context,
4                           tcp::resolver::results_type&& endpoint,
```

```
                        std::shared_ptr<ThreadsafeDeque<Message>> ready_msg_storage);

        void encode(Message& msg) override;

    protected:
        void connect();
        void decode() override;

    private:
        void decodeHeader();
        void decodeBody();

    protected:
        std::shared_ptr<boost::asio::io_context> context_;
        std::unique_ptr<tcp::resolver::results_type> endpoint_;
};
```

Листинг 3.11: Заголовочный файл класса ClientCodec

```
ClientCodec::ClientCodec(std::shared_ptr<boost::asio::io_context> context,
                         tcp::resolver::results_type&& endpoint,
                         std::shared_ptr<ThreadsafeDeque<Message>> ready_msg_storage)
: Codec(std::move(tcp::socket(*context)), ready_msg_storage),
  context_(context),
  endpoint_(std::make_unique<tcp::resolver::results_type>(std::move(endpoint)))
{
    connect();
}

void ClientCodec::encode(Message& msg)
{
    std::cout << "I'm in write" << std::endl;
    enc_message_ = enc_->encode(msg);

    boost::asio::async_write(*socket_, *(enc_message_.toVecOfConstBuf()),
    [this](boost::system::error_code ec, size_t)
    {
        if(ec)
        {
            is_alive_ = false;
            std::cout << ec.message() << std::endl;
        }
    }
    );

    std::cout << "I'm after write" << std::endl;
}

void ClientCodec::connect()
{
    std::cout << "Im in connect" << std::endl;
    boost::asio::async_connect(*socket_, *endpoint_,
    [this](boost::system::error_code ec, tcp::endpoint)
    {
        if(!ec)
            decode();
        else
        {
            is_alive_ = false;
            std::cout << ec.message() << std::endl;
        }
    }
```

```
44        );
45
46        std::cout << "Im after connect" << std::endl;
47  }
48
49  void ClientCodec::decode()
50  {
51        decodeHeader();
52  }
53
54  void ClientCodec::decodeHeader()
55  {
56        boost::asio::async_read(*socket_, boost::asio::buffer(dec_message_.getHead(),
              SerializedMessage::HeaderLength),
57        [this](boost::system::error_code ec, size_t)
58        {
59              if(!ec)
60              {
61                    std::cout << "In decodeHeader() func" << std::endl;
62                    dec_->decodeHeader(dec_message_);
63                    std::cout << "temp message size: " << dec_message_.getDataLength()
64                              << std::endl;
65                    decodeBody();
66              }
67              else
68              {
69                    is_alive_ = false;
70                    std::cout << ec.message() << std::endl;
71              }
72        }
73        );
74  }
75
76  void ClientCodec::decodeBody()
77  {
78        std::cout << "temp message size in decodeBody: " << dec_message_.getDataLength()
79                  << std::endl;
80
81        boost::asio::async_read(*socket_, boost::asio::buffer(dec_message_.getData(),
              dec_message_.getDataLength() - 1),
82        [this](boost::system::error_code ec, size_t)
83        {
84              if(!ec)
85              {
86                    std::cout << "In decodeBody() func" << std::endl;
87                    Message decs = dec_->decodeData(dec_message_);
88                    std::cout << decs.data << std::endl;
89                    std::cout << "In decodeBody() func end" << std::endl;
90                    ready_msgs_storage_->push_back(decs);
91                    decodeHeader();
92              }
93              else
94              {
95                    std::cout << ec.message() << std::endl;
96                    is_alive_ = false;
97              }
98        }
99        );
100 }
```

Листинг 3.12: Определение методов класса ClientCodec

Класс ClientCodec создается в конструкторе класса RoboClient в котором ClientCodec предаются объектс контекста, поддерживающего асинхронные возможности, учетные данные сервера и хранилище. В конструкторе самого ClientCodec происходит создание сокета, пока еще пустого, из контекста. В функции connect происходит подключение к серверу, по учетным данным и заполнение сокета необходимой для поддержания связи с сервером информации, с помощью функции boost::asio::async_connect. В случае успешного подключения, объект ec преобразуется к типу bool и в зависимости от значения полученного в результате преобразования к bool происходит вызов функции decode , либо вывод сообщения об ошибки выполнения подключения. В случае, если ec равно значению true, вызывается функция decode. Если внимательно взглянуть на функции decode(), decodeHeader() и decodeBody(), можно заметить, что они вызывают друг друга, образуя своеобразную рекурсию. Это связано с особенностями асинхронного чтения данных из сетевого сокета, с помощью функции boost::asio::async_read. Данная функция находится в неактивном режиме, в момент ее вызова, до тех пор, пока через сетевой сокет не возобновится предача данных. Это означает, что вызов данной функции не означает начало ее выполнения, а лишь гарантирует ее исполнения, в момент начала или возобновления передачи данных, для этого и нужна рекурсия в этих функциях. После того как чтение закончится, нужно рекурсивно вызвать функцию чтения еще раз, чтобы она стала "запланированной если этого не сделать, то процесс чтения оборвется после прихода первого сообщения. Это касалось процесса чтения из сокета, однако клиент должен уметь и отправлять данные в сокет. Для этого используется функция encode, которая сериализует сообщение до потока байтиов с помощью объекта enc_ класса Encoder, о котором будет идти речь ниже, и отправляет их через сокет с помощью функции boost::asio::async_write прямо к серверу. Рассмотрим структуру серверного кодека:

```
class ServerCodec : public Codec {
	public:
		ServerCodec(tcp::socket&& socket,
				std::shared_ptr<ThreadsafeDeque<Message>> storage);

		void encode(Message& msg) override;

	protected:
		void decode() override;

	private:
		void decodeHeader();
		void decodeBody();
};
```

Листинг 3.13: Заголовочный файл класса ServerCodec

```
ServerCodec::ServerCodec(tcp::socket&& socket,
				std::shared_ptr<ThreadsafeDeque<Message>> storage)
: Codec(std::move(socket), storage)
{
	decode();
}
```

```
7
8  void ServerCodec::encode(Message& msg)
9  {
10         enc_message_ = enc_->encode(msg);
11
12         std::cout << msg.data << std::endl;
13
14         boost::asio::async_write(*socket_, *enc_message_.toVecOfConstBuf(),
15         [this](boost::system::error_code ec, size_t)
16         {
17                 if(ec)
18                 {
19                         is_alive_ = false;
20                 }
21                 else
22                         std::cout << "encode error: " << ec.message() << std::endl;
23         }
24         );
25 }
26
27 void ServerCodec::decode()
28 {
29         decodeHeader();
30 }
31
32 void ServerCodec::decodeHeader()
33 {
34         boost::asio::async_read(*socket_, boost::asio::buffer(dec_message_.getHead(),
               SerializedMessage::HeaderLength),
35         [this](boost::system::error_code ec, size_t)
36         {
37                 if(!ec)
38                 {
39                         std::cout << "Decode header function" << std::endl;
40                         dec_->decodeHeader(dec_message_);
41                         decodeBody();
42                 }
43                 else
44                 {
45                         std::cout << "decode header error: " << ec.message() << std::endl;
46                         is_alive_ = false;
47                 }
48         }
49         );
50 }
51
52 void ServerCodec::decodeBody()
53 {
54         boost::asio::async_read(*socket_, boost::asio::buffer(dec_message_.getData(),
               dec_message_.getDataLength() - 1),
55         [this](boost::system::error_code ec, size_t)
56         {
57                 if(!ec)
58                 {
59                         ready_msgs_storage_->push_back(dec_->decodeData(dec_message_));
60                         decodeHeader();
61                 }
62                 else
63                 {
64                         std::cout << "decode body error: " << ec.message() << std::endl;
65                         is_alive_ = false;
```

```
66                  }
67          }
68          );
69  }
```

Листинг 3.14: Определение методов класса ServerCodec

Реализации кодеков для клиента и сервера почти неотличимы друг от друга, за исключением нескольких моментов. Первый момент - конструктор, т.к. сервер принимает входящие подключения, то объект сокет уже создается функцией async_accept, находящейся в классе RobotServerCreator. Все что остается кодеку, это просто использовать уже созданный сокет для целей передачи данных. Второй момент - отсутсвие использования функции async_connect, т.к. это сервер, ему нет нужды подключаться к клиентам, он способен лишь принимать подключения. Во всем остально функции практически неотличимы.

Именно таким образом реализованы функции по принятию данных и их передаче. Далее же будет рассмотрен процесс создания протокола, согласно которому идет передача и соответствующие ему структуры.

3.1.3 Протокол связи

Так как Boost.Asio реализует низкоуровневые функции сетевого взаимодействия прикладного уровня OSI, то при перессылки данных пользоваться разрешено только этими функциями. Прикладной уровень обеспечивает устойчивость соединения и гарантирует перессылку данных при правильно подключений. То какую форму эти данные будут иметь и как их правильно трактовать будет определено в протоколе связи. Функции библиотеки Boost.Asio способны работать лишь с потоком битов, поэтому нужно предусмотреть реализацию процесса сериализации/дессериализации. Так же необходимо учитывать то, какие структуры будут реализовывать сериализованное и дессериализованное состояние сообщения. В качестве реализации сериализованного сообщения выступает класс SerializedMessage, о котором кратко шла речь в главе про архитектуру. Рассмотрим его методы и устройство:

```
1  class SerializedMessage {
2          public:
3                  enum { HeaderLength = 7 };
4
5          public:
6                  SerializedMessage();
7                  SerializedMessage(uint32_t body_length);
8                  SerializedMessage(const SerializedMessage& other);
9                  SerializedMessage(SerializedMessage&& other);
10
11                 SerializedMessage& operator=(const SerializedMessage& other);
12
13                 ~SerializedMessage();
14
15                 bool empty() const;
16
17                 const char* getData() const;
18                 char* getData();
```

```
19              const char* getHead() const;
20              char* getHead();
21              uint32_t getDataLength() const;
22
23              void setDataLength(uint32_t body_length);
24
25              std::shared_ptr<std::vector<boost::asio::const_buffer>> toVecOfConstBuf() const;
26
27      private:
28              char header_[HeaderLength];
29              char* data_;
30              uint32_t body_length_;
31 };
```

Листинг 3.15: Заголовочный файл класса SerializedMessage

```
1  SerializedMessage::SerializedMessage()
2  : data_(nullptr),
3    body_length_(0)
4  { }
5
6  SerializedMessage::SerializedMessage(uint32_t body_length)
7  {
8          body_length_ = body_length;
9          data_ = new char[body_length];
10 }
11
12 SerializedMessage::SerializedMessage(const SerializedMessage& other)
13 {
14         memcpy(header_, other.header_, HeaderLength);
15
16         body_length_ = other.body_length_;
17
18         data_ = new char[body_length_];
19
20         memcpy(data_, other.data_, body_length_);
21 }
22
23 SerializedMessage::SerializedMessage(SerializedMessage&& other)
24 {
25         memcpy(header_, other.header_, HeaderLength);
26
27         std::swap(data_, other.data_);
28
29         body_length_ = other.body_length_;
30 }
31
32 SerializedMessage& SerializedMessage::operator=(const SerializedMessage& other)
33 {
34         memcpy(header_, other.header_, HeaderLength);
35
36         body_length_ = other.body_length_;
37
38         if(data_ != nullptr)
39         delete[] data_;
40
41         data_ = new char[body_length_];
42
43         memcpy(data_, other.data_, body_length_);
44
45         return *this;
46 }
```

```
47
48 SerializedMessage::~SerializedMessage()
49 {
50         if (data_ != nullptr)
51         delete[] data_;
52 }
53
54 bool SerializedMessage::empty() const {
55         return (body_length_ == 0);
56 }
57
58 const char* SerializedMessage::getData() const {
59         return data_;
60 }
61
62 char* SerializedMessage::getData() {
63         return data_;
64 }
65
66 const char* SerializedMessage::getHead() const {
67         return header_;
68 }
69
70 char* SerializedMessage::getHead() {
71         return header_;
72 }
73
74 void SerializedMessage::setDataLength(uint32_t body_length) {
75         if(data_ != nullptr)
76         {
77                 body_length_ = body_length;
78
79                 delete[] data_;
80
81                 data_ = new char[body_length_];
82         }
83         else
84         {
85                 body_length_ = body_length;
86
87                 data_ = new char[body_length_];
88         }
89 }
90
91 uint32_t SerializedMessage::getDataLength() const {
92         return body_length_;
93 }
94
95 std::shared_ptr<std::vector<boost::asio::const_buffer>>
96 SerializedMessage::toVecOfConstBuf() const
97 {
98         std::shared_ptr<std::vector<boost::asio::const_buffer>> res;
99         res = std::make_shared<std::vector<boost::asio::const_buffer>>();
100
101         res->push_back(boost::asio::buffer(header_));
102         res->push_back(boost::asio::buffer(data_, body_length_));
103
104         return res;
105 }
```

Листинг 3.16: Определение методов класса SerializedMessage

Класс абстрагирует сериализованное сообщение поэтому его поля содержат только массивы и их характеристики. Эти массивы и являются инвариантом данного класса, их передача и осуществляется через функции async_write и async_read. Можно заметить, что массива в классе два, один из них константного размера HeaderLength, а другой - динамический. Это сделано для разделения сериализованного сообщения на две части - "голова"и "тулвище". В голове записывают метаданные необходимые для определения размера тулвища и некоторые специальные константы, обозначающие тип и команду, которую в себе несет данное сообщение, подробнее об этом будет рассказано ниже. Тулвище - массив произвольного размера, который хранит в себе данные необходимые для полного определения операций, например, аргументы команд и т.п. Как видно из опредления методов, большинство из них вспомогательные, предназначенные для более удобного создания экземпляра данного класса. Явно выделяется лишь метод toVecOfConstBuf(), который нужен для создания константного буффера boost::asio. Это часто необходимо для более простой конвертации массивов хранящихся внутри класса в подходящий для передачи вид объектов класса boost::asio::const_buffer. Теперь необходимо рассмотреть класс Message, который является реализацией абстракции дессериализованного сообщения:

```
1 struct Message {
2         std::string type;
3         std::string command;
4         std::string data;
5         bool result;
6 };
```

Листинг 3.17: Заголовочный файл класса Message

Определенных методов у класса Message, дессериализованного сообщения нет, т.к. он является лишь структурой, которая заполняется в процессе дессериализации, служащей для передачи различным обработчикам сообщений. Однако он хорошо отражает структуру протокола. Поле type тип string, хранит в себе один из следующих типов сообщения:

1. "Hello"
2. "RespHello"
3. "Response"
4. "Request"

Это все типы сообщений, которые могут быть однозначно закодированны и декодированны в сообщении. Каждому из типов соответствует набор разрешенных command. Пока в протоколе активно используются тольо Request и Response типы сообщений, им соответсвуют комманды:

1. "No command"
2. "getRoboContainerInfo"

3. "execRoboContainer"
4. "getExecutedRoboContainer"
5. "execDDSServer"
6. "deleteRoboImage"
7. "buildRoboImage"
8. "stopRoboContainer"
9. "getRoboLog"

Типу Response, соответсвует только команда "No command т.к. в ответе не должно быть никаких комманд, все остальные команды соответствуют типу Request сообщения. Поле data, хранит в себе информацию необходимую для настройки запроса (различные аргументы, json структур и т.п.), либо, в случае Response, различные данные для формирования результата (различные сообщения об ошибка, если запрос провален и т.п.). Поле result, кратко сообщает об успехе запроса, если в Response сообщении поле установлено в true - значит выполнение запроса прошло успешно и наоборот. Классы SerializedMessage и Message являются различными состояниями сообщения, поэтому их можно соотнести. Так поле type и command серализуются в массив "головы"в SerializedMessage, а поле data и result находятся в массиве "тело"в SerializedMessage.

Сам процесс перехода между различными состояниями сообщения происходит в классах Encoder и Decoder. Именно они реализуют процесс дессериализации и сериализации. Рассмотрим их:

```
class Encoder {
       public:
              Encoder();

              SerializedMessage encode(Message& msg);

       protected:
              std::map<std::string, uint8_t> type_;
              std::map<std::string, uint8_t> command_;
};
```

Листинг 3.18: Заголовочный файл класса Encoder

```
Encoder::Encoder()
{
       type_.insert(std::make_pair("Hello", 1));
       type_.insert(std::make_pair("RespHello", 2));
       type_.insert(std::make_pair("Response", 3));
       type_.insert(std::make_pair("Request", 4));

       command_.insert(std::make_pair("No command", 0));
       command_.insert(std::make_pair("getRoboContainerInfo", 1));
       command_.insert(std::make_pair("execRoboContainer", 2));
       command_.insert(std::make_pair("getExecutedRoboContainer", 3));
       command_.insert(std::make_pair("execDDSServer", 4));
```

```
13          command_.insert(std::make_pair("deleteRoboImage", 5));
14          command_.insert(std::make_pair("buildRoboImage", 6));
15          command_.insert(std::make_pair("stopRoboContainer", 7));
16          command_.insert(std::make_pair("getRoboLog", 8));
17  }
18  SerializedMessage Encoder::encode(Message& msg)
19  {
20          if (type_.find(msg.type) == type_.end())
21                  throw std::runtime_error("Can't encode message without propriate type!");
22
23          std::optional<uint8_t> command;
24
25          if (command_.find(msg.command) == command_.end())
26          {
27                  if (type_[msg.type] == 4)
28                  throw std::runtime_error("Can't encode request without command!");
29          }
30          else
31          {
32                  if(type_[msg.type] < 3)
33                          throw std::runtime_error("Can't encode this type of message
34                                                    with command!");
35                  else
36                          command.emplace(command_[msg.command]);
37          }
38
39          uint32_t data_size = msg.data.size();
40          std::cout << "Enc data size in Encoder: " << static_cast<int>(data_size) << std::endl;
41
42          SerializedMessage m(data_size);
43
44          char type_c[1];
45          type_c[0] = static_cast<char>(type_[msg.type]);
46
47          char command_c[1];
48          if (command.has_value())
49                  command_c[0] = static_cast<char>(command_[msg.command]);
50          else
51                  command_c[0] = static_cast<char>(command_["No command"]);
52
53          char* size_b = new char[4];
54
55          size_b[0] = (data_size >> 24) & 0xFF;
56          size_b[1] = (data_size >> 16) & 0xFF;
57          size_b[2] = (data_size >> 8) & 0xFF;
58          size_b[3] = (data_size & 0xFF);
59
60          char result_c[1];
61          result_c[0] = static_cast<char>(static_cast<int>(msg.result));
62
63          memcpy(m.getHead(), type_c, 1);
64          memcpy(m.getHead() + 1, command_c, 1);
65          memcpy(m.getHead() + 2, size_b, 4);
66          memcpy(m.getHead() + 6, result_c, 1);
67          memcpy(m.getData(), msg.data.c_str(), data_size);
68
69          return m;
70  }
```

Листинг 3.19: Определение методов класса Encoder

Как видно из конструктора класса Encoder, в нем создается отображение

из строковых литералов в числа (из типа string в int), заключенное в объекте command_ и такое же отображение в объекте type_. Назначение этих объектов хранить возможные множества типов и комманд, которые могут быть переданны в сообщении. Они отображают название комманд в числа, для их более простого перевода в поток битов. Сам процесс сериализации заключен в методе encode(...), который принимает не сериализованное сообщение и, в качестве результата работы, отдает сериализованное сообщение. Внутри метода, на строках с 21 по 45 идет процесс подготовки к сериализации, проверяется структура Message на возможные логические ошибки, например, на передачу команды в сообщении типа Response и т.п. В строках с 56 по 59 идет сериализация размера поля data сообщения. Это необходимо, чтобы точно понимать сколько битов нужно принимать в функции async_read. Процесс сериализации поля data идет побайтово, в массив из четырех char, т.к. тип поля data uint32 и его размер строго 4 байта, а один объект типа char занимает 8 бит или один байт. Далее на строке 62 идет преобразования поля result, имеющего тип bool, в тип char с помощью конструкции языка static_cast, после чего, на строках с 64 по 68 идет прямое копирование в участки памяти массива "головы"серализованного сообщения с помощью функции memcpy. В "голову"перемещаются сведения о типе сообщения, команде, размере поля data, результат работы. В "тело"копируется содержимое поле data. Рассмотрим процесс дессериализации и класс ее реализующий.

```
1  class Decoder {
2          public:
3                  Decoder();
4
5                  void decodeHeader(SerializedMessage& msg);
6                  Message decodeData(SerializedMessage& msg);
7
8          protected:
9                  std::map<uint8_t, std::string> inv_type_;
10                 std::map<uint8_t, std::string> inv_command_;
11 };
```

Листинг 3.20: Заголовочный файл класса Decoder

```
1  Decoder::Decoder()
2  {
3          inv_type_.insert(std::make_pair(1, "Hello"));
4          inv_type_.insert(std::make_pair(2, "RespHello"));
5          inv_type_.insert(std::make_pair(3, "Response"));
6          inv_type_.insert(std::make_pair(4, "Request"));
7
8          inv_command_.insert(std::make_pair(0, "No command"));
9          inv_command_.insert(std::make_pair(1, "getRoboContainerInfo"));
10         inv_command_.insert(std::make_pair(2, "execRoboContainer"));
11         inv_command_.insert(std::make_pair(3, "getExecutedRoboContainer"));
12         inv_command_.insert(std::make_pair(4, "execDDSServer"));
13         inv_command_.insert(std::make_pair(5, "deleteRoboImage"));
14         inv_command_.insert(std::make_pair(6, "buildRoboImage"));
15         inv_command_.insert(std::make_pair(7, "stopRoboContainer"));
16         inv_command_.insert(std::make_pair(8, "getRoboLog"));
17 }
18
19 void Decoder::decodeHeader(SerializedMessage& msg)
```

```
{
        char data_size_c[4];
        memcpy(data_size_c, msg.getHead() + 2, 4);

        uint32_t data_size = static_cast<uint32_t>(static_cast<uint8_t>(data_size_c[0]) << 24 |
                                                   static_cast<uint8_t>(data_size_c[1]) << 16 |
                                                   static_cast<uint8_t>(data_size_c[2]) << 8 |
                                                   static_cast<uint8_t>(data_size_c[3]));

        msg.setDataLength(data_size + 1);
}

Message Decoder::decodeData(SerializedMessage& msg)
{
        char type_c[1];
        char command_c[1];
        char result_c[1];

        memcpy(type_c, msg.getHead(), 1);
        memcpy(command_c, msg.getHead() + 1, 1);
        memcpy(result_c, msg.getHead() + 6, 1);

        Message m;

        m.type = inv_type_[static_cast<uint8_t>(type_c[0])];
        m.command = inv_command_[static_cast<uint8_t>(command_c[0])];
        msg.getData()[msg.getDataLength() - 1] = '\0';
        m.data = std::string(msg.getData());
        m.result = static_cast<uint8_t>(result_c[0]) == 0 ? false : true;

        return m;
}
```

Листинг 3.21: Определение методов класса Decoder

В конструкторе класс опредлены отражения, обратные отражениям в классе Encoder. Это нужно, что из чисел вновь получить текстовые команды в удобочитаемом виде. Сам процесс дессериализации распределен между методами decodeHeader(...) и decodeData(...), т.к. без дессериализации "головы"невозможно узнать, сколько еще нужно прочитать данных, чтобы сформировать "тело". По этой же причине и происходит разделение методов декодирования в классах-наследниках Codec. После передачи сериализованного сообщения в качестве аргумента в метод decodeHeader, происходит обратный процесс, данные из "головы"перемещаются во временный массив char, после чего собираются воедино и преобразовываются к временном объекту data_size, имеющему тип uint32_t. Далее для корректного заполнения "тела"присходит выделение памяти в сообщении с помощью метода setDataLength(...). В методе decodeData(...) происходит формирование выходного сообщения с типом Message и процесс копирования битов из массива и их последующего преобразования в соответсвующие типы с использованием конструкций static_cast.

Процесс сериализации/дессериализации строго задан в методах классов Encoder/Decc поэтому порядок заполнения так же строго соблюдается. На это процесс реализации протокола можно считать закрытым.

Обработка результатов

Раздел посвящен выполнению инструкций, переданных в форме сообщений, их обработке и структуре классов ответсвенных за обработку. Также здесь будет рассмотрена структура потокобезопастного хранилища, синхронизирующего сетевые функции и обработку.

3.1.4 Обработка данных клиента

Обработкой ответов и формированием запросов, занимается сам класс RoboClient, т.к. это удобно интегрируется в функции графического интерфейса пользователя. Рассмотрим на примере одной из функции формирующих запрос в RoboClient, т.к. остальные функции работают по схожему принципу:

```
bool RoboClient::startContainer(std::string&& id, MContainerSetting settings)
{
        Message msg;
        msg.type = "Request";
        msg.command = "execRoboContainer";

        docker_cpp::CreateConfig conf;

        conf.Image = id.substr(7);
        conf.Tty = settings.tty_;
        conf.OpenStdin = true;
        conf.NetworkDisabled = false;
        conf.Entrypoint.push_back("bash");
        conf.Entrypoint.push_back("-c");
        std::string command = settings.launch_dir_ + ".launch.py " + settings.arguments_;
        conf.Cmd.push_back(command);
        conf.hostConfig.networkMode = "host";
        conf.hostConfig.autoRemove = false;
        conf.hostConfig.privileged = true;
        conf.hostConfig.ipcMode = "host";
        conf.hostConfig.readonlyRootfs = false;

        json j = conf;

        msg.data = settings.launch_dir_ + " " + settings.name_ + " " + j.dump();

        codec_->encode(msg);

        std::shared_ptr<Message> resp = ready_msgs_->wait_pop_back();

        if (!resp->result)
        {
                std::cerr << resp->data << std::endl;
                return false;
        }

        return true;
}
```

Листинг 3.22: Определение метода startContainer класса RoboClient

Данная команда формирует запрос на создания микросервиса, который будет запускать определенный компонент алгоритма одновременного картографирования и локализации. Формирование запроса начинается с создания экземпляра класса Message. Объекту будет присвоен тип "Request означающий, что дан-

но сообщение является запросом на действие. Далее сообщению будет передана команда "execRoboContainer которая означает запрос на создание и запуск микросервиса. После идет создание объекта класса docker_cpp::CreateConfig, которая является частью библиотеки, реализующей C++ docker API. Объект класса docker_cpp::CreateConfig реализует конфигурацию, которую необходимо передать Docker демону, для создания и запуска контейнера. Далее идет заполнение этой структуры, на 9 строке идет передача id образа из которого должен быть запущен данный микросервис. На 12 строке идет включение функций использующих сеть. На 13 и 14 строке происходит передача названия командной оболочки, котороая будет использоваться для запуска команды, запускаемой при старте микросервиса, в данном случае запускать команду будет оболочка bash. На 15 и 16 строке происходит формирование самой команды, которая будет запущена. В нее передается название директории, в которой будет хранится нужный для запуска launch файл и его аргументы, а затем идет передача в конфигурацию. Это не полное определение команды, т.к. команда контекстно-зависима, то необходимый для второй части контекст находится в реализации обработчика запросов на серверной стороне сетевого сервиса, который будет рассмотрен позже. На 17 строке идет передача конфигурации сети в микросервисе, в данном случае это "host означающее что контейнер будет пользоваться сетью машины на которой запущен. На строках 18-21 идет настрока привелигерованного пользователя контейнера, использование контейнером распределенной памяти машины на которой он запущен и разрешение на доступ к системным папкам обычных (не root) пользователей в микросервисе. После заполнения конфигурации, она передается в объект типа json, который является частью библиотеки nlohmann::json, реализующий API для взаимодействия с json файлами из программы на C++. Передача конфигурации в объект типа json, запускает процесс его конвертации в json-структуру, в дальнейшем эту структуру можно будет преобразовывать к типу std::string, что облегчает процесс формирования сообщения. В поле data сообщения мы передаем название директории, в которой хранится launch файл необходимый для запуска микросервиса, название микросервиса и объект std::string типа, полученный путем преобразования структуры json с помощью метода dump(). Таким образом, мы легко можем заполнять произвольные сообщения, а метод encode класса ClientCodec, легко их сериализует до потока битов, с помощью класса Encoder, и отправит их серверу. После отправки запроса, поток управления простанавливается на вызове метода wait_pop_back(...) хранилища ready_msgs_, реализуя своеобразный механизм ожидание ответа от сервера. После принятия хранилищем сообщения полученного от сервера, происходит возобновление работы потока управления и мы переходим на конструкцию if, которая проверяет результат выполнения запроса, если запрос выполнен успешно, то функция startContainer, возвращает в вызывающую функцию true, которая означает, что процесс завершился успешно и наоборот.

3.1.5 Обработка данных сервером

Обработка данных серверной частью сетевого сервиса представляет собой более сложный процесс, т.к. идет взаимодействие с библиотекой docker_cpp. Для того чтобы рассмотреть процесс, нам необходимо вернуться к рассмотрению метода process класса RobotServer.

```
1  void RobotServer::process()
2  {
3          while(!stop_td_)
4          {
5                  std::shared_ptr<Message> request = ready_msgs_->wait_pop_back();
6
7                  if (request != nullptr)
8                  {
9                          std::shared_ptr<Message> resp = msg_handler_.handleRequest(request);
10                         codec_->encode(*resp);
11                 }
12         }
13 }
```

Листинг 3.23: Определение метода process класса RobotServer

Это основной цикл работы сервера, в нем по условию stop_td_ идет работа с запросами. На строке 5 поток выполнения блокируется, ожидая прихода дессериализованного сообщения в хранилище. После того, как запрос принят и обработан ServerCodec - ом, поток возобновляет свою работу и передает в объект request сообщение типа Message. Далее идет проверка на пустоту объекта request, т.к. возможна ситуация, когда серверу будет необходимо завершить свою работу и свернуть сетевые функции, даже если выполнение заблокировано на функции wait_pop_back(). Осуществляется это с помощью одного из методов класса хранилища, о котором будет рассказано позже. На строке 9 идет вызов метода handleRequest у объекта msg_handler_ типа RequestHandler, который и осуществляет работу запрошеную в сообщении. Рассмотрим класс RequestHandler подробнее.

```
1  class RequestHandler {
2          public:
3                  RequestHandler();
4                  ~RequestHandler();
5
6                  std::shared_ptr<Message> handleRequest(std::shared_ptr<Message> reciv_msg);
7
8          protected:
9                  std::shared_ptr<Executor> chooseExecutor(std::string& command);
10
11         protected:
12                 std::map<std::string, std::shared_ptr<Executor>> command_to_exec_;
13                 std::shared_ptr<Docker<ASLHttp>> docker_engine_ptr_;
14                 std::shared_ptr<SystemManipulator> system_manip_;
15                 std::shared_ptr<ImageSettings> image_settings_;
16 };
```

Листинг 3.24: Заголовочный файл класса RequestHandler

```
1  RequestHandler::RequestHandler()
```

```
: docker_engine_ptr_(std::make_shared<Docker<ASLHttp>>("http://127.0.0.1:2375")),
  system_manip_(std::make_shared<SystemManipulator>()),
  image_settings_(std::make_shared<ImageSettings>())
{
	if (!docker_engine_ptr_->checkConnection())
		throw std::runtime_error("Can't connect to docker engine web api on server!");

	command_to_exec_.insert(std::make_pair(
	"getRoboContainerInfo", std::make_shared<RoboContainerInfoExec>(docker_engine_ptr_,
                                                                    system_manip_)));
	command_to_exec_.insert(std::make_pair("getExecutedRoboContainer", std::make_shared<
	    RoboStartedContainerInfoExec>(docker_engine_ptr_)));
	command_to_exec_.insert(std::make_pair("getRoboLog", std::make_shared<
	    RoboContainerLogExec>(docker_engine_ptr_)));
	command_to_exec_.insert(std::make_pair("execDDSServer", std::make_shared<
	    RoboStartDDSExec>(docker_engine_ptr_)));
	command_to_exec_.insert(std::make_pair(
	"execRoboContainer", std::make_shared<RoboContainerExec>(docker_engine_ptr_,
                                                             system_manip_,
                                                             image_settings_)));
	command_to_exec_.insert(std::make_pair("stopRoboContainer", std::make_shared<
	    RoboStopContainerExec>(docker_engine_ptr_)));
	command_to_exec_.insert(std::make_pair(
	"deleteRoboImage", std::make_shared<RoboDeleteImageExec>(docker_engine_ptr_,
                                                             image_settings_)));
	command_to_exec_.insert(std::make_pair(
	"buildRoboImage", std::make_shared<RoboBuildImageExec>(docker_engine_ptr_,
                                                           system_manip_,
                                                           image_settings_)));
}

RequestHandler::~RequestHandler()
{
	docker_engine_ptr_.reset();
	system_manip_.reset();
	image_settings_.reset();
}

std::shared_ptr<Message> RequestHandler::handleRequest(std::shared_ptr<Message> reciv_msg)
{
	std::shared_ptr<Executor> exec = chooseExecutor(reciv_msg->command);

	if (exec == nullptr)
	{
		std::shared_ptr<Message> m = std::make_shared<Message>();
		m->type = "Response";
		m->result = false;
		m->data = "No such command, please provide class to handle this type of message
		    in message handler area";

		return m;
	}
	else
		return exec->execute(reciv_msg);
}

std::shared_ptr<Executor> RequestHandler::chooseExecutor(std::string& command)
{
	auto&& exec_it = command_to_exec_.find(command);


```

```
58          if(exec_it != command_to_exec_.end())
59                  return exec_it->second;
60          else
61                  return nullptr;
62  }
```

Листинг 3.25: Определение методов класса RequestHandler

Данный класс реализует паттер программирования "Команда". Управляя группой классов, наследованных от класса Executor, он осуществляет реакцию на запросы клиента. В конструкторе класса создаются объекты класса Docker<ASLHttp>, реализующие методы для доступа к web API Docker, а также вспомогательные классы, которые помогают группе классов наследованных от Executor, реализовывать запрос, не перегружая при этом пользователя графического интерфейса различными вопросами настройки. При создании объекта класса Docker<ASLHttp> необходимо, в качестве аргумента передать учетные данные web API Docker сервера в виде std::string объекта, после проверки подключения с помощью метода checkConnection(...), можно считать настройку объекта законченной. Внутри конструктора класса RequestHandler создается объект отображения command_to_exec, в котором командам добавлены соответсвующие им классы исполнители, которым в свою очередь передаются необходимые для работы объекты. Это необходимо для того, чтобы по полю command класса Message определять необходимого исполнителя запроса. Далее в методе handleRequest, который вызывается из метода process класса RobotServer, происходит непосредственный выбор исполнителя, путем вызова метода chooseExecutor. В случае отсутствия соответсвующего исполнителя, формируется сообщение, в котором передается информация о произошедшей ситуации. Все классы исполнители наследованы от базового Executor, поэтому их устройство схоже, рассмотрим один из них в качестве демонстрации обработки данных.

```
1  class RoboContainerExec : public Executor {
2          public:
3                  RoboContainerExec(std::shared_ptr<Docker<ASLHttp>> docker_engine,
4                                  std::shared_ptr<SystemManipulator> system_manip,
5                                  std::shared_ptr<ImageSettings> image_settings)
6                  : Executor(docker_engine),
7                    system_manip_(system_manip),
8                    image_settings_(image_settings)
9                  { }
10
11                 virtual ~RoboContainerExec() = default;
12
13                 virtual std::shared_ptr<Message> execute(std::shared_ptr<Message> msg) override;
14
15         private:
16                 std::shared_ptr<SystemManipulator> system_manip_;
17                 std::shared_ptr<ImageSettings> image_settings_;
18 };
```

Листинг 3.26: Заголовочный файл класса RoboContainerExec

```
1  std::shared_ptr<Message> RoboContainerExec::execute(std::shared_ptr<Message> msg)
2  {
3          size_t delim = msg->data.find(" ");
```

```
4         std::string launch_dir = msg->data.substr(0, delim);
5         std::string temp_str = msg->data.substr(delim + 1);
6         delim = temp_str.find(" ");
7         std::string name = temp_str.substr(0, delim);
8         std::string json_str = temp_str.substr(delim+1);
9
10        json j = json::parse(json_str);
11
12        std::shared_ptr<docker_cpp::CreateConfig> conf_ptr = std::make_shared<docker_cpp::
              CreateConfig>(j.get<docker_cpp::CreateConfig>());
13
14        std::shared_ptr<Message> m = std::make_shared<Message>();
15
16        m->type = "Response";
17
18  #ifdef DEBUG
19        std::cout << "---------------------------" << std::endl;
20        std::cout << "Current POST command in container: " << j.dump(' ') << std::endl;
21        std::cout << "---------------------------" << std::endl;
22  #endif
23
24        std::string pkg_launch_path = system_manip_->getAbsoluteLaunchPath(launch_dir);
25        std::string pkg_config_path = system_manip_->getAbsoluteConfigPath(launch_dir);
26
27  #ifdef DEBUG
28        std::cout << "-------------------------" << std::endl;
29        std::cout << "Pkg Config path: " << pkg_config_path << std::endl;
30        std::cout << "Pkg Launch path: " << pkg_launch_path << std::endl;
31        std::cout << "-------------------------" << std::endl;
32  #endif
33
34        if (pkg_config_path.empty() || pkg_launch_path.empty())
35        {
36                m->result = false;
37                std::string error_msg = "Can't find config or launch folder with " + launch_dir
                      + " name!";
38                m->data = error_msg;
39  #ifdef DEBUG
40                std::cerr << error_msg << std::endl;
41  #endif
42
43                return m;
44        }
45
46        std::string username = image_settings_->getUsernameById(conf_ptr->Image);
47
48        if (username.size() == 0)
49        {
50                m->result = false;
51                std::string error_msg = "Can't find username for " + conf_ptr->Image
52                                        + "image id";
53                m->data = error_msg;
54  #ifdef DEBUG
55                std::cerr << error_msg << std::endl;
56  #endif
57
58                return m;
59        }
60
61        conf_ptr->User = "docker_" + username;
62
```

```
63          std::string subcommand1 = "source /opt/ros/humble/setup.bash";
64          std::string subcommand2 = " && set -a && source /home/docker_" + username + "/launch/up
                .env && set +a";
65          std::string subcommand3 = " && ros2 launch /home/docker_" + username + "/launch/";
66
67          std::string command = subcommand1 + subcommand2 + subcommand3;
68
69          conf_ptr->Cmd[0].insert(0, command);
70
71          conf_ptr->hostConfig.binds.push_back(pkg_config_path + ":" + "/home/docker_" + username
                + "/config");
72          conf_ptr->hostConfig.binds.push_back(pkg_launch_path + ":" + "/home/docker_" + username
                + "/launch");
73
74          docker_cpp::DockerError err_create = docker_engine_copy_->createContainer(*conf_ptr,
                name);
75          docker_cpp::DockerError err_start = docker_engine_copy_->containerStart(name);
76
77          if (err_create.isError())
78          {
79                  std::stringstream ss;
80                  ss << err_start;
81                  std::string error_msg = "Error while create container! Server response: " + ss.
                        str();
82                  m->result = false;
83                  m->data = error_msg;
84  #ifdef DEBUG
85                  std::cerr << error_msg << std::endl;
86  #endif
87
88                  return m;
89          }
90          else if (err_start.isError())
91          {
92                  std::stringstream ss;
93                  ss << err_start;
94                  std::string error_msg = "Error while start container! Server response: " + ss.
                        str();
95                  m->result = false;
96                  m->data = error_msg;
97  #ifdef DEBUG
98                  std::cerr << error_msg << std::endl;
99  #endif
100
101                 return m;
102         }
103
104         m->result = true;
105         return m;
106 }
```

Листинг 3.27: Определение методов класса RoboContainerExec

Обработка начинается с декодировки данных сокрытых в поле data объекта msg. Первым из поля извлекается объект launch_dir, который несет в себе информацию о названии директории, в которой находится launch файл необходимый для запуска микросервиса, затем извлекается поле name, которое сообщает о названии данного микросервиса. Названия необходимо для быстрой идентификации микросервиса среди других, оно должно кратко и емко отра-

жать суть компонента запущенного в системе. Далее извлекается сериализованная json структура, которая на строках с 10 по 12 дессериализуется до объекта типа std::shared_ptr<docker_cpp::CreateConfig>, являющейся заполненной на клиентской стороне конфигурацией. Далее идет создание сообщения, в которое будет передан ответ сервера. На строках с 24 по 25 идет получение абсолютных путей к директориям в которых хранится launch файл и файл конфигурации для launch файла. Оба этих файла ищутся по названию переданному с клиентской стороны. Это накладывает важное ограничение на структур директории хранящей в себе инфраструктур для создания и запуска микросервисов на серверной стороне, а именно, название директории для конфигурации должно быть идентичным названию директории в которой хранятся launch файлы. Далее идет первая проверка, на наличие этих директорий на сервере. В случае их отсуствия формируется ответ с ошибкой и поток выполнения немедлено возвратит управление вызывающей функции, чтобы та отправил ответ клиенту. По данному принципу работает большинство проверок в данном сетевом сервисе. На строке 46 идет получение имени пользователя, созданного в контенере. Это необходимо для корректной установки директорий с launch файлом и директорий с конфигурацией в контейнер. Получение происходит путем вызова метода, который ищет нужное имя сопоставленное с переданным id образа, из которого будет запускаться контейнер. Поиск происходит сначала в памяти процесса, а если сервер был перезапущен или закрыт в нормальном режиме, ищет в сохраненном на сервере специального вида файле, хранящем в себе все множество пар имя-id. После получения имени, приходит расширение конфигурации переданной с клиентской стороны. Это вторая часть процесса формирования конфигурации, о которой говорилось выше. К конфигурации добавляется имя пользователя контейнера. Со строки 53 по 59 идет добавление комманд к существующим. Устройство директорий хранящей инфраструктур для запуска унифицирует процесс создания команды до нескольких переменных, поэтому команду удалось жестко закрепить в коде. На строках с 71 по 72 идет процесс связывания директорий находящихся на сервере, с директориями контейнера. Это необходимо, чтобы папки для запуска launch файла и папка хранящая в себе конфигурацию добавились в микросервис и стали доступными для чтения и изменения. После чего идет вызов методов C++ Docker API. Метод createContainer принимает объект конфигурации и имя контейнера, после чего возвращает объект ошибки, который проверяется на строках с 77 по 89. Этот метод создает микросервис на сервере. Далее в методе containerStart идет запуск созданного микросервиса по его имени. Если все проверки пройдены, то в поле result сообщения устанавливается значение true, сигнализирующее об успехе выполненной операции.

3.1.6 Синхронизирующее хранилище

Хранилище играет особую роль в устройстве передачи и обработки данных в сетевом сервисе. Эту абстракцию реализует класс ThreadsafeDeque, особые свойства которого помогают в синхронизации процесса запрос-ответ. Рассмотрим его

структуру.

```
template<typename T>
class ThreadsafeDeque {
public:
	ThreadsafeDeque();

	void push_back(T new_value);

	void shutdown();

	void wait_pop_back(T& value);
	std::shared_ptr<T> wait_pop_back();

	bool try_pop_back(T& value);
	std::shared_ptr<T> try_pop_back();

	bool empty();

private:
	mutable std::mutex mut;
	std::atomic_bool stop_deq;
	std::deque<T> data_deque;
	std::condition_variable data_cond;
};
```

Листинг 3.28: Заголовочный файл класса ThreadsafeDeque<T>

```
template <typename T>
ThreadsafeDeque<T>::ThreadsafeDeque()
{ }

template <typename T>
void ThreadsafeDeque<T>::push_back(T new_value)
{
	std::lock_guard<std::mutex> lk(mut);

	data_deque.push_back(std::move(new_value));
	data_cond.notify_one();
}

template <typename T>
void ThreadsafeDeque<T>::shutdown()
{
	stop_deq = true;
	data_cond.notify_one();
}

template <typename T>
void ThreadsafeDeque<T>::wait_pop_back(T& value)
{
	std::unique_lock<std::mutex> lk(mut);
	data_cond.wait(lk,
	[this]
	{
		return !data_deque.empty() || stop_deq;
	}
	);

	if (!data_deque.empty())
	{
		value = std::move(data_deque.back());
```

```
35                data_deque.pop_back();
36        }
37 }
38
39 template <typename T>
40 std::shared_ptr<T> ThreadsafeDeque<T>::wait_pop_back()
41 {
42        std::unique_lock<std::mutex> lk(mut);
43        data_cond.wait(lk,
44        [this]
45        {
46                return !data_deque.empty() || stop_deq;
47        }
48        );
49
50        if (!data_deque.empty())
51        {
52                std::shared_ptr<T> res = std::make_shared<T>(std::move(data_deque.back()));
53                data_deque.pop_back();
54                return res;
55        }
56        else
57                return nullptr;
58 }
59
60 template <typename T>
61 bool ThreadsafeDeque<T>::try_pop_back(T& value)
62 {
63        std::lock_guard<std::mutex> lk(mut);
64
65        if(data_deque.empty())
66                return false;
67
68        value = std::move(data_deque.back());
69        data_deque.pop_back();
70
71        return true;
72 }
73
74 template <typename T>
75 std::shared_ptr<T> ThreadsafeDeque<T>::try_pop_back()
76 {
77        std::lock_guard<std::mutex> lk(mut);
78
79        if(data_deque.empty())
80                return std::shared_ptr<T>();
81
82        std::shared_ptr<T> res = std::make_shared<T>(std::move(data_deque.back()));
83        data_deque.pop_back();
84
85        return res;
86 }
87
88 template <typename T>
89 bool ThreadsafeDeque<T>::empty() {
90        std::lock_guard<std::mutex> lk(mut);
91        return data_deque.empty();
92 }
```

Листинг 3.29: Определение методов класса ThreadsafeDeque<T>

Реализация структуры построена на шаблонном классе, т.к. хорошим тоном

является реализация хранилища способного хранить различные типы данных. Принцип работы данной структуры очень прост. В структуре есть общий мьютех mut, который блокирует все операции кроме единственной, которая выполняется в данный момент, однако, это не вся синхронизация. В классе присуствуется объект типа condition_variable, которая срабатывает только при вызове метода push_back(...) класса. Это означает, что сколько бы потоков не пыталось воспользоваться методами данной структуры, они будут переведены в режим ожидания до того момента пока один из потоков не захватит мьютекс в методе push_back() и не добавит в структур элемент. После добавления элемента, conditional_variable объект уведомит все потоки, которые пытались получить данные используя методы хранилища о том, что элемент добавлен. После уведомления, самый быстрый из потоков захватит освобожденный, после конца вызова метода push_back(...), мьютекс и выполнит получение данных через метод. В этом и состоит механизм синхронизации. В серверной части, в методе process(...) поток управления пытается захватить мьютекс структуры, но т.к. изначально в хранилище нет элементов, то поток "засыпает". После получения сервером запроса от клиента, он дессериализуется до структуры Message в классе ServerCodec и передается в разделенное между RobotServer и ServerCodec общее хранилище. То есть, ServerCodec передает Message в тоже хранилище, с которого ожидает этих данных класс RobotServer.

На этом процесс рассмотрения сетевого взаимодействия и обработки можно считать законченным.

3.2 Разработка графического интерфейса пользователя

Графический интерфейс пользователя состоит из нескольких виджетов, реализующих функции всплывающих банеров, палитры окон для управления микросервисом и т.п. Начнем с рассмотрения главного окна и его устроства, реализует эту абстракцию класс MainWindow

3.2.1 Реализация главного окна

Главное окно в данной программе является абстракцией, которая размещает все другие виджеты внутри себя и отображает их на экран компьютера.

```
1  class MainWindow : public QMainWindow
2  {
3          Q_OBJECT
4  public:
5          MainWindow(QWidget* parent = nullptr,
6                     bool show_splash_screen = true);
7
8          void createMenu();
9
10 private slots:
11         void helpRequest();
12
```

```
private:
        ServerWidget* server_;
        QPlainTextEdit* container_log;
        QWidget* central;
        QVBoxLayout* main_layout;
};
```

Листинг 3.30: Заголовочный файл класса MainWidget

```
MainWindow::MainWindow(QWidget* parent,
bool show_splash_screen)
: QMainWindow(parent) {
        central = new QWidget(this);

        container_log = new QPlainTextEdit();

        server_ = new ServerWidget(container_log);

        createMenu();

        container_log->setReadOnly(true);

        container_log->appendPlainText("New msgs from server");

        main_layout = new QVBoxLayout(central);
        main_layout->addWidget(server_);
        main_layout->addWidget(container_log);

        this->setCentralWidget(central);
}

void MainWindow::createMenu() {
        QAction* logo = new QAction(QIcon(":/images/Android.png"), QString(), this);
        logo->setEnabled(false);

        QAction* add_new_action = new QAction("&Add new", this);
        connect(add_new_action, &QAction::triggered, server_, &ServerWidget::createServer);

        menuBar()->addAction(logo);

        menuBar()->addMenu("&Server")->addAction(add_new_action);
}

void MainWindow::helpRequest() {
        QMessageBox::information(this, "About app",
                                 "This is a app for master degree based on Qt5.");
}
```

Листинг 3.31: Определение методов класса MainWidget

Если посмотреть на заголовочный файл класса, можно увидеть разницу в структуре между обычным C++ классом и классом-виджетом для Qt5. Здесь присуствуют новые ключевые слова, такие как Q_OBJECT и slots, которые специфичны только для классов-виджетов. Эти ключевые слова обрабатываются Meta Object Compilator (MOC) и из них формируются заголовочные файлы с метаинформацией о виджете. Эта информация в дальнейшем используется для рефлексии. Рефлексия позволяет реализовать механику сигналов и слотов, как говорилось выше. В конструкторе создается центральный виджет, объект central класса QWidget. QWidget класс построен таким образом, что теоретически способен

вмещать в себя сколько угодно других виджетов, это работает благодаря рефлексии. Далее создается объект типа QPlainTextEdit, который является виджетом способным отражать текст. После чего создается виджет ServerWidget, который является основным виджетом по управлению состоянием микросервисов, о нем будет рассказано чуть позже. Далее добавляется текстовая надпись и выставляется флаг для объекта типа QPlainTextEdit, который запрещает пользователям редактирование текста в данном виджете. На строке 16 идет создания главного менеджера компоновки, в конструкторе ему передан центральный виджет, что означает что данный менеджер будет подчиняться центральному виджету и будет в нем отображен. После виджет передается в метод setCentralWidget(...), что означает использования данного виджета, как центрального в окне. В функции createMenu(...) идет оформление верхнеуровнего меню. В этой функции происходит процесс создания объектов QAction, который представляют собой класс, который реализует передачу сигнала и простое отображение (например, текстовое). С помощью функции menuBar() являющейся методом класса QMainWindow, программа получает доступ к верхнеуровневому меню, которое уже является частью класса QMainWindow. В это меню можно добавлять различные объекты класса QAction, отображаться в конечном приложении эти объекты будут как простые прямоугольники с надписями. Процесс происходящий на строке 28 представляет большой интерес, т.к. именно там происходит соединение сигнала и слота. В качестве аргументов данной функции слева на право передаются: объект, которой реализует сигнал, сам сигнал (принимает форму указателя на метод объекта), объект, который реализует слот и сам слот (также принимает форму указателя на метод). После соединения сигнала и слота, в случае нажатия на отображение в графическом интерфейсе принадлежащее add_new_action (т.е. на надпись "Add new"), происходит отправка сигнала нажатия к объекту server_, который обработает его в слоте createServer(...). С помощью функции addAction(...), принадлежащей верхнеуровнему меню, к нему добавляются объекты QAction.

3.2.2 Реализация виджета клиента

Виджет клиента, реализуется в классе ServerWidget. Он служит в качестве связующего звена между классом реализующим клиент - RoboClient и между отображением функций класса клиента в форме виджетов.

```
1  class ServerWidget : public QWidget
2  {
3          Q_OBJECT
4
5  public:
6          ServerWidget(QPlainTextEdit* cont_log,
7                  QWidget* parent = nullptr);
8
9          ~ServerWidget();
10
11 public slots:
12         void createServer();
13         void onServerClicked(QListWidgetItem* item);
```

```
14          void onServerBackClicked();
15
16          void showContextBuild(const QPoint& pos);
17          void showContextName(const QPoint& pos);
18          void showContextRun(const QPoint& pos);
19
20  private:
21          int server_counter_;
22          QString last_server_name_;
23          bool first_run_container_fire_;
24          std::thread context_td_;
25          std::shared_ptr<RoboClientCreator> client_creator_;
26          std::map<QString, std::shared_ptr<RoboClient>> clients_;
27          QPlainTextEdit* container_log;
28          QPushButton* back_button_;
29          QWidget* glue_container_name_wgt_;
30          QTableWidget* container_build_table_wgt;
31          QTableWidget* container_name_table_wgt;
32          QTableWidget* container_run_table_wgt;
33          QVBoxLayout* glue_container_name_layout;
34          QVBoxLayout* main_srv_layout_;
35          QWidget* glue_srv_name_wgt;
36          QVBoxLayout* glue_srv_name_layout;
37          QListWidget* srv_name_list_wgt;
38          QStackedWidget* wgt_stack;
39  };
```

Листинг 3.32: Заголовочный файл класса ServerWidget

Определение методов ServerWidget занимает около 500 строк кода, поэтому для экономии места, здесь будут приведен лишь заголовочный файл класса. Создание трех виджетов описывающих управление микросервисами происходит в конструкторе класса ServerWidget:

```
1           srv_name_list_wgt = new QListWidget();
2
3           glue_container_name_wgt_ = new QWidget();
4
5           glue_container_name_layout = new QVBoxLayout(glue_container_name_wgt_);
6
7           container_build_table_wgt = new QTableWidget();
8           container_name_table_wgt = new QTableWidget();
9           container_run_table_wgt = new QTableWidget();
10
11          ...
12
13          back_button_ = new QPushButton("X");
14
15          glue_container_name_layout->addWidget(back_button_);
16          glue_container_name_layout->addWidget(container_build_table_wgt);
17          glue_container_name_layout->addWidget(container_name_table_wgt);
18          glue_container_name_layout->addWidget(container_run_table_wgt);
19
20          wgt_stack->addWidget(srv_name_list_wgt);
21          wgt_stack->addWidget(glue_container_name_wgt_);
22
23          main_srv_layout_->addWidget(wgt_stack);
```

Листинг 3.33: Создание виджетов ServerWidget

Здесь идет создание объект glue_container_name_wgt_, который будет управлять менеджером компоновки glue_container_name_layout. Внуть менеджера переданы три виджета container_build_table_wgt, который создает виджет таблицы отображающий в себе образа микросервисов, которые можно собрать, container_name который создает виджет таблицы отображающий в себе собранные образа микросервисов, которые можно запустить, container_run_table_wgt, который создает виджет таблица отображающий в себе запущенные микросервисы. После добавления всех виджетов в менеджер компоновки, их можно передать объекту wgt_stack, который является объектом stack виджет класса. Stack виджет реализует механику смену вида, о которой говорилось в главе про архитектуру. Вторым объектом, переданным в stack виджет является объект srv_name_list_wgt класса QListWidget, который реализует отображения перечня серверов, к котором подключен супер-клиент. Также, вместе с тремя виджетам, реализующими отображения управления микросервисами, в менеджер компоновки передается виджет QPushButton, который помогает реализовать механику переключения между видами в stack виджете. Данный процесс проще продемонстрировать с помощью схемы:

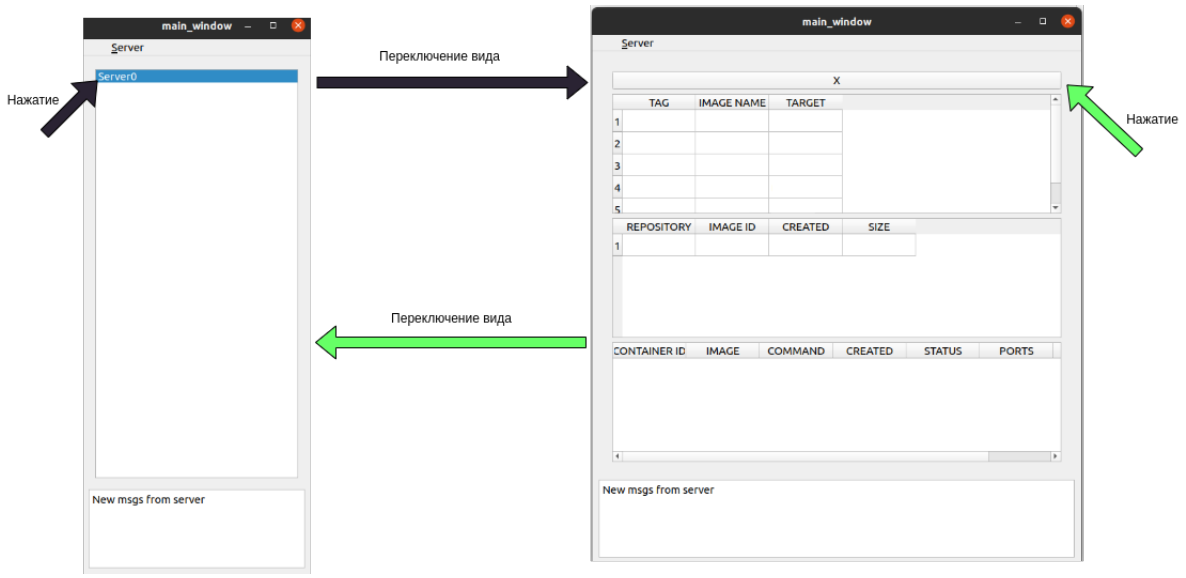


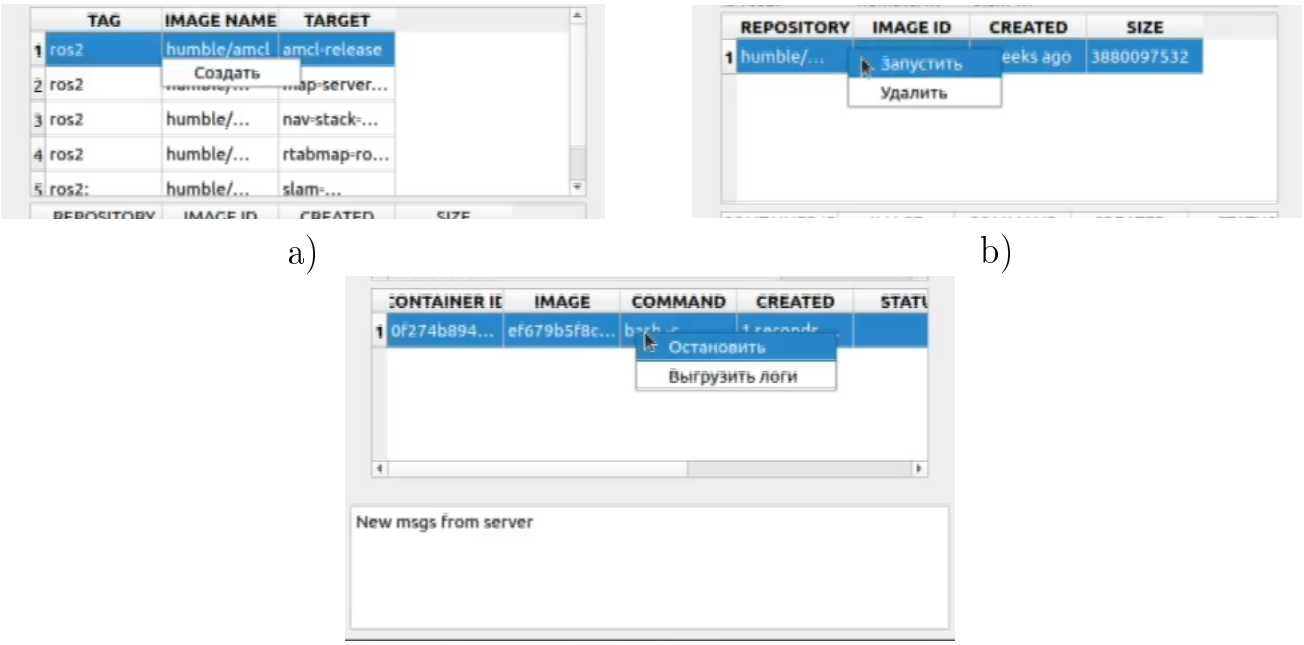
Рис. 3.1: Схема переключения вида в ServerWidget

После создания необходимых виджетов идет процесс связывания сигналов этих виджетов со слотами класса ServerWidget.

```
1        connect(srv_name_list_wgt, &QListWidget::itemClicked, this, &ServerWidget::
             onServerClicked);
2        connect(back_button_, &QPushButton::clicked, this, &ServerWidget::onServerBackClicked);
3        connect(container_build_table_wgt, &QTableWidget::customContextMenuRequested, this, &
             ServerWidget::showContextBuild);
4        connect(container_name_table_wgt, &QTableWidget::customContextMenuRequested, this, &
             ServerWidget::showContextName);
5        connect(container_run_table_wgt, &QTableWidget::customContextMenuRequested, this, &
             ServerWidget::showContextRun);
```

Листинг 3.34: Связывания сигналов и слотов в ServerWidget

Здесь содержится информация о реагировании каждого из членов графического интерфейса на действия пользователя, например, при нажатии на один из серверов в списке srv_name_list_wgt, отправляется сигнал itemClicked, который обрабатывается слотом onServerClicked. Внутри метода переключается вид в stack виджете и отрисовываются виджеты ответственные за микросервисы. Интерес представляет связь виджетов таблиц. Здесь происходит запрос контекстного меню, которое открывается при нажатии на правую кнопку мыши поверх строки таблицы. Для каждой из таблиц создается собственный вид контекстного меню и собственная реакция на предложенные действия.



a) b) c)

Рис. 3.2: Контекстное меню виджетов. a) - образа микросервисы, которые можно собрать, b) - образа микросервисов, которые можно запустить, c) - образа микросервисов, запущенных на сервере

Рассмотрим один из методов класса ServerWidget, для того чтобы понять устройство взаимодействия между графическим интерфейсом и классом клиента. Например, метод createServer(...)

```
1       ServerCredentalBox* server_cred_ = new ServerCredentalBox();
2
3       if (server_cred_->exec() == QDialog::Accepted)
4       {
5               wgt_stack->setCurrentIndex(0);
6
7               QString server_name("Server");
8
9               auto [ip, port] = server_cred_->getIpPort();
10              server_name += QString::number(server_counter_);
11
12              clients_[server_name] = client_creator_->createRoboClient(ip, port);
13
14              if(!clients_[server_name]->createDDSInstance(server_counter_))
15              {
16                      auto&& erase_it = clients_.find(server_name);
```

```
17                      clients_.erase(erase_it);
18              }
19              else
20              {
21                      QListWidgetItem* server_item_ = new QListWidgetItem(server_name);
22                      server_item_->setIcon(QPixmap(":/images/Android.png"));
23
24                      srv_name_list_wgt->addItem(server_item_);
25
26                      server_counter_++;
27              }
28      }
```

Листинг 3.35: Определение слота createServer

Это опредление слота класса ServerWidget, который вызывается в ответ на сигнал от верхнеуровнего меню главного виджета, который отправляется при нажатии кнопки создания нового подключения к серверу. На 1 строке идет запуск всплывающего окна диалога, который запрашивает у пользователя учетные данные сервера.

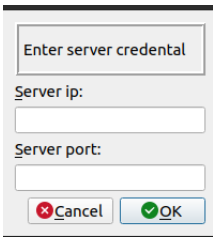


Рис. 3.3: Запрос данных сервера

Класс ServerCredentalBox реализует функции всплывающего окна и сохраняет учетные данные, как поле класса. Поток выполнения останавливается на вызове метода exec(...) объекта server_cred_, после ввода данных и нажатие на кнопку "OK поток выполнения снова возобновляет свою работу. Вызов метода exec(...) возвращает объект типа enum, который сравнивается с константой Accepted. Это проверка необходима для того, чтобы весь дальнейший код внутри if выполнился только в том случае, если пользователь нажал на кнопку "OK". Также, сохранения учетных данных в поле класса ServerCredentalBox реализовано на сигнале от нажатия кнопки "OK". На 5 строке идет переключение вида stack виджета, после чего идет формирования надписи "Server"с соотвествующим порядковым номером. На строке 9 идет вызов метода, класса диалога, который получает учетные данные сервера из полей класса диалога. На 12 строке идет создание класса клиента, с помощью вызова метода класса создателя клиентов. Класс клиентов, копируется в отображение расположенное в классе, в нем каждому имени сервера, соответсвует его клиент. На строке 14 идет вызова метода класса клиента, которая формирует запрос к серверу с требованием о запуске FastDDS Discovery Server, который осуществляет связь между микросервисами и роботом. В случае успеха, сервер добавляется к виджету и отображается на экране. В противном случае, такой сервер удаляется.

Глава 4

Заключение

В заключении хотелось бы подвести результаты выполненной работы. В исследовании получилось соединить несколько технологий для достижения общей цели - запуска алгоритмов одновременного картографирования и локализации. У данного решения еще есть много проблем с безопастносью и функционалом, однако, оно уже является продуктом, которым можно пользоваться в работе. В дальнейшем можно рассмотреть расширение функционала управления кластером микросервисов, добавить новые полезные сведения в графических интерфейс пользователя и включить инициализацию FastDDS Discovery Server через DDS API, однако, в рамках ограниченного времени, решение получилось локаничным и функциональным.

4.1 Основные результаты исследования

В качестве цели данной работы, стояла задача создания сетевого сервиса, обеспечивающее управление кластером микросервисов, реализующих алгоритм одновременного картографирования и локализации. Ознакомиться с демонстрационным видео можно по ссылке. В ходе работы был создан прототип сетевого сервиса для управления алгоритмами одновременного картографирования и локализации, реализованный с использованием передовых подходов к созданию подобных сервисов. Сервис использует Docker для изоляции и масштабирования своих микросервисов. Для реализации сетевого взаимодействия используется библиотека Boost.Asio, на основе которой построен протокол для связи клиентской и серверной части сервиса, что позволилос обеспечить высокую производительность и кросплатформенность. Для визуализации и управления разработа графический интерфей пользователя на базе Qt5, обеспечивающий интуитивный доступ к функционалу системы. В результате, сервис демонстрирует стабильную работу при работе с алгоритмами одновременного картографирования и локализации, подтверждя эффективность выбора технологий и архитектуру системы.

4.2 Дальнейшее направление исследований

Дальнейшее развитие системы, можно разделить на 3 основных направления:

- Расширение функции графического интерфейса. Добавление новых виджетов, позволяющих более полно отображать состояние микросервисов. Например, создания виджета, отображающего граф с микросервисами в вершинах и топиками в качестве ребер. По этому графу будет удобно визуализировать состояние микросервиса и возможные ошибки в его работе.

- Расширение протокола связи. Протокол связи построенный на Boost.Asio не позволяет пересылать непрерывные потоки данных, которые бы расширили функционал других подсистем сервиса. Например, потоковая передача данных о состоянии сервиса, позволило бы отображать его состояние в реальном времени.

- Обработка данных. Обработка механизма обработки нуждается в серьезной доработке, т.к. она не позволяет дать пользователю полный контроль над микросервисами.

Данный прототип системы позволяет оценить потенциал решений в области Облачной робототехники, однако, для серьезной продуктовой разработки роботов он не подходит. Системе нужна дальнейшая поддержка, которая позволила бы реализовать недостающий функционал. Однако, прототип уже обладает всеми необходимыми функциями, которые требуются для базовой настройки микросервисов, протестировать работу можно перейдя по ссылке и скачав репозитории с исходным кодом данного проекта.

Литература

[1] Gubbi J. и др. Internet of Things (IoT): A vision, architectural elements, and future directions // Future Generation Computer Systems. 2013. Т. 29. № 7. С. 1645–1660.

[2] Zhang J., Tao D. Empowering Things with Intelligence: A Survey of the Progress, Challenges, and Opportunities in Artificial Intelligence of Things // 2020.

[3] Hu G., Tay W., Wen Y. Cloud robotics: architecture, challenges and applications // IEEE Network. 2012. Т. 26. № 3. С. 21–28.

[4] Miratabzadeh S. A. и др. Cloud robotics: A software architecture: For heterogeneous large-scale autonomous robots // 2016 World Automation Congress (WAC). Rio Grande, PR, USA: IEEE, 2016. С. 1–6.

[5] Muzumdar P. и др. Navigating the Docker Ecosystem: A Comprehensive Taxonomy and Survey // 2024.

[6] Mullinix S. P. и др. On Security Measures for Containerized Applications Imaged with Docker // 2020.

[7] Eng L. Z. Qt5 C++ GUI programming cookbook: practical recipes for building cross-platform GUI applications, widgets, and animations with Qt 5. Birmingham, UK: Packt Publishing, 2019. Вып. Second edition. 1 с.

[8] Qt 5.15 [Электронный ресурс]. URL: https://doc.qt.io/qt-5/.

[9] Torjo J., Anggoro W. Boost.Asio C++ network programming: learn effective C++ network programmingwith Boost.Asio and become a proficient C++ network programmer, second edition. Birmingham: Packt Publishing, 2015. Вып. 2nd ed. 1 с.

[10] Kadusic E. и др. The transitional phase of Boost.Asio and POCO C++ networking libraries towards IPv6 and IoT networking security // 2022 IEEE International Conference on Smart Internet of Things (SmartIoT). Suzhou, China: IEEE, 2022. С. 80–85.

[11] Newman S. Building microservices: designing fine-grained systems. Beijing Sebastopol, CA: O'Reilly Media, 2015. Вып. First Edition. 259 с.

[12] Nadareishvili I. Microservice architecture: aligning principles, practices, and culture. Sebastopol, CA: O'Reilly Media, Inc, 2016. Вып. First edition. 128 с.

[13] Erl T. Service-oriented architecture: concepts, technology, and design. Upper Saddle River, NJ Munich: Prentice Hall PTR, 2009. Вып. 9. print. 760 с.

[14] Lawler J. P., Howell-Barber H. Service-oriented architecture: SOA strategy, methodology, and technology. Boca Raton: Auerbach, 2019.

[15] Gilbert J. Software Architecture Patterns for Serverless Systems: Architecting for innovation with event-driven microservices and micro frontends. Birmingham: Packt Publishing Limited, 2024. Вып. 1. 1 с.

[16] Sbarski P. Serverless architectures on AWS. Shelter Island: Manning, 2017. 354 с.

[17] Saji J., Kumar A. A Review Paper on Serverless Architecture of Web Applications // SSRN Journal. 2024.

[18] Percival H. J. W., Gregory B. Architecture patterns with Python: enabling test-driven development, domain-driven design, and event-driven microservices. Beijing Boston Farnham Sebastopol Tokyo: O'Reilly, 2020. Вып. First edition. 1 с.

[19] Олифер В. Компьютерные сети. Принципы, технологии, протоколы. Санкт-Петербург: Питер, 2021. Вып. Юбилейное издание. 1008 с.

[20] Gamal A. E., Kim Y.-H. Lecture Notes on Network Information Theory // 2010.

[21] Martinez W. L. Graphical user interfaces // WIREs Computational Stats. 2011. Т. 3. № 2. С. 119–133.

[22] Martinez W. L. Graphical user interfaces // WIREs Computational Stats. 2011. Т. 3. № 2. С. 119–133.

[23] Ernest A., Mensah E., Gilbert A. Qualitative Assessment of Compiled, Interpreted and Hybrid Programming Languages // CAE. 2017. Т. 7. № 7. С. 8–13.

[24] Aho A. V., Sethi R., Ullman J. D. Compilers, principles, techniques, and tools. Reading, Mass: Addison-Wesley Pub. Co, 1986. 796 с.

[25] Stroustrup B. The C++ programming language: C++ 11. Upper Saddle River, NJ: Addison-Wesley, 2015. Вып. 4. ed., 4. print. 1347 с.

[26] Rassokhin D. The C++ programming language in cheminformatics and computational chemistry // J Cheminform. 2020. Т. 12. № 1. С. 10.

[27] Macenski S. и др. Robot Operating System 2: Design, architecture, and uses in the wild // Sci. Robot. 2022. Т. 7. № 66. С. eabm6074.

[28] Macenski S. и др. Impact of ROS 2 Node Composition in Robotic Systems // 2023.

[29] Corsaro A., C. D. The Data Distribution Service – The Communication Middleware Fabric for Scalable and Extensible Systems-of-Systems // System of Systems / под ред. A. V. Gheorghe. : InTech, 2012.

[30] Mahmood Z. Connected Environments for the Internet of Things: Challenges and Solutions. Cham: Springer, 2018. 269–285 с.

[31] Martin K., Hoffman B. Mastering CMake: Version 3.1. Clifton Park, NY.: Kitware Inc, 2015. 700 с.

[32] Crandall Z. и др. CMakePPLang: An object-oriented extension toCMake // JOSS. 2023. Т. 8. № 89. С. 5 с.

[33] Smith P. Software build systems: principles and experience. Upper Saddle River, N.J: Addison Wesley, 2011. 583 с.

[34] Mokhov A., Mitchell N., Peyton Jones S. Build systems à la carte // Proc. ACM Program. Lang. 2018. Т. 2. № ICFP. С. 1–29.

[35] Shotts W. E. The Linux command line: a complete introduction. San Francisco: No Starch Press, 2012. 504 с.

[36] Voronkov A., Martucci L. A., Lindskog S. System Administrators Prefer Command Line Interfaces, Don't They? An Exploratory Study of Firewall Interfaces // 2019.

[37] Zelle J. M. Python programming: an introduction to computer science. Portland: Franklin, Beedle & Associates Inc, 2024. Вып. Fourth edition. 574 с.

[38] Urban M., Murach J. Murachs Python programming: beginner to pro. Fresno, CA: Mike Murach & Associates, Inc., 2021. Вып. 2nd edition.

[39] Simple Participant Discovery [Электронный ресурс]. URL: https://community.rti.com/static/documentation/connext-dds/current/doc/manuals/connext_dds_professional/users_manual/users_manual/Simple_Participant_Discovery.htm.

[40] Negus C. Linux Bible, 10th Edition. Erscheinungsort nicht ermittelbar: Wiley, 2020. Вып. 10th edition. 1556 с.

[41] Stallman R., Lessig L. Free software, free society: selected essays of Richard Stallman. Boston, MA: Free Software Foundation, 2010. Вып. 2nd ed.

[42] Scaradozzi D., Zingaretti S., Ferrari A. Simultaneous localization and mapping (SLAM) robotics techniques: a possible application in surgery // Shanghai Chest. 2018. Т. 2. С. 5–5.

[43] Kudriashov A. SLAM Techniques Application for Mobile Robot in Rough Terrain. Cham: Springer International Publishing AG, 2020. 131 с.

[44] Kala R. Autonomous mobile robots: planning, navigation and simulation. London: Academic Press, 2024. 1064 с.

[45] Särkkä S., Svensson L. Bayesian Filtering and Smoothing. : Cambridge University Press, 2023. Вып. 2.

[46] Kutschireiter A. и др. Nonlinear Bayesian filtering and learning: a neuronal dynamics for perception // Sci Rep. 2017. Т. 7. № 1. С. 8722.

[47] Placed J. A. и др. A Survey on Active Simultaneous Localization and Mapping: State of the Art and New Frontiers // IEEE Transactions on Robotics (T-RO). 2022. С. 1–20.

[48] Boal J., Sánchez-Miralles Á., Arranz Á. Topological simultaneous localization and mapping: a survey // Robotica. 2014. Т. 32. № 5. С. 803–821.

[49] Kitanov A., Indelman V. Topological belief space planning for active SLAM with pairwise Gaussian potentials and performance guarantees // The International Journal of Robotics Research. 2024. Т. 43. № 1. С. 69–97.

[50] Chen K. и др. Semantic Visual Simultaneous Localization and Mapping: A Survey // 2022.

[51] Bowman S. L. и др. Probabilistic data association for semantic SLAM // 2017 IEEE International Conference on Robotics and Automation (ICRA). Singapore, Singapore: IEEE, 2017. С. 1722–1729.

[52] Das S. Robot localization in a mapped environment using Adaptive Monte Carlo algorithm // 2025.

[53] Zuo C. и др. An Improved Adaptive Monte Carlo Localization Algorithm Integrated with a Virtual Motion Model // Sensors. 2025. Т. 25. № 8. С. 2471.

[54] Alshikh K., Mhd A., Lyad H. gmcl As a Proposed Replacement to amcl in ROS for Mobile Robots Localization in Known-Based 2D Environments. // 2021.

[55] Kohlbrecher S. и др. A flexible and scalable SLAM system with full 3D motion estimation // 2011 IEEE International Symposium on Safety, Security, and Rescue Robotics. Kyoto, Japan: IEEE, 2011. С. 155–160.

[56] Labbe M., Michaud F. RTAB-Map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation // Journal of Field Robotics. 2019. Т. 36. № 2. С. 416–446.

[57] Hess W. и др. Real-time loop closure in 2D LIDAR SLAM // 2016 IEEE International Conference on Robotics and Automation (ICRA). Stockholm, Sweden: IEEE, 2016. С. 1271–1278.

[58] Zimmermann H. OSI Reference Model–The ISO Model of Architecture for Open Systems Interconnection // IEEE Trans. Commun. 1980. Т. 28. № 4. С. 425–432.

[59] Day J. D., Zimmermann H. The OSI reference model // Proc. IEEE. 1983. Т. 71. № 12. С. 1334–1340.

[60] POCO C++ Libraries - Reference Library [Электронный ресурс]. URL: https://docs.pocoproject.org/current/.

[61] Schmidt D. C., Huston S. D. C++ network programming. Boston: Addison-Wesley, 2002.

[62] Silberschatz A., Galvin P. B., Gagne G. Operating system concepts. Hoboken, NJ: Wiley, 2018. Вып. Tenth edition.

[63] Design patterns: elements of reusable object-oriented software / под ред. E. Gamma. Boston, Mass. Munich: Addison-Wesley, 2011. Вып. 39. printing. 395 с.

[64] Schmidt D. C. Using design patterns to develop reusable object-oriented communication software // Commun. ACM. 1995. Т. 38. № 10. С. 65–74.

[65] Williams A. C++ concurrency in action. Shelter Island, NY: Manning Publications Co, 2019. Вып. Second edition. 568 с.

[66] Herlihy M. The art of multiprocessor programming // Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing. Denver Colorado USA: ACM, 2006. С. 1–2.

[67] UML distilled: a brief guide to the standard object modeling language / под ред. M. Fowler. Boston, Mass.: Addison-Wesley, 2010. Вып. 3. ed, 16. printing. 175 с.

[68] Docker Engine API v1.48 reference [Электронный ресурс]. URL: https://docs.docker.com/reference/api/engine/version/v1.48/.

[69] Diener M. и др. Communication in Shared Memory: Concepts, Definitions, and Efficient Detection // 2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP). Heraklion, Crete, Greece: IEEE, 2016. С. 151–158.

[70] Maes P. Concepts and experiments in computational reflection // SIGPLAN Not. 1987. Т. 22. № 12. С. 147–155.