РЕФЕРАТ

Пояснительная записка 00 стр., 00 рис., 00 табл., 00 ист., 00 прил.

КЛЮЧЕВЫЕ СЛОВА И СЛОВОСОЧЕТАНИЯ, НЕ БОЛЕЕ ПЯТНАДЦАТИ, ЧЕРЕЗ ЗАПЯТУЮ

КЛЮЧЕВЫЕ СЛОВА И СЛОВОСОЧЕТАНИЯ, НЕ БОЛЕЕ ПЯТНАДЦАТИ, ЧЕРЕЗ ЗАПЯТУЮ

Объектом исследования (разработки) являются указать объект исследования или разработки.

Цель работы – кратко (в 2-3 строки) указать цель работы.

Кратко (в 10-12) строк описать основное содержание работы, методы исследования (разработки), полученные результаты.

ABSTRACT

Briefly (10-15 lines) the content of graduating work is specified

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

В настоящей пояснительной записке применяют следующие термины с соответствующими определениями:

ОС — операционная система.

ПО — программное обеспечение.

ППО — промежуточное программное обеспечение.

МАРППО — многоагентное робототехническое программное обеспечение.

API — Application Programming Interface, интерфейс прикладного программирования.

P2P — Peer-To-Peer, одноранговая сеть.

URL — Uniform Resource Locator, единый указатель ресурса.

RTT — Real-Time Toolkit.

OCL — ORoCoS Component Library.

GPU — Graphics Processing Unit, графический процессор.

CPU — Central Processing Unit, центральный процессор.

ORB — Object Request Brokers, брокеры объектных запросов.

QoS — Quality of Service, предоставление различному трафику различный приоритет в обслуживании.

XML — eXtensible Markup Language, язык разметки.

JSON — JavaScript Object Notation, текстовый формат обмена данными.

DNS — Domain Name Service, система для получения информации о доменах в сети.

GUI — Graphical User Interface, графический пользовательский интерфейс.

—

—

ВВЕДЕНИЕ

Для программирования роботов доступно множество различных версий фреймворков с различными принципами работы, написанные на разных языках программирования и под разные платформы. В связи с тем, что появляются новые разработки, возникают новые задачи для автономных роботов, а так же технологии разработки ПО для них - возникает желание рассмотреть доступные и развивающиеся в данный момент решения и проанализировать с целью установки характеристик производительности и сравнения по полученным параметрам, чтобы разработчики могли обосновывать свой выбор при разработки приложений для автономных роботов. При этом необходимо учитывать как соответствие фреймворков возможным общим критериям (лицензия, статус разработки), так и важным для конкретной области: разработки ПО (программного обеспечения) для роботов. Для выполнения тестирования, следует определиться с тем, какие задачи, выполняемые фреймворком, являются значимыми для производительности системы в целом, какие компоненты системы требуется протестировать.

Для выполнения тестирования требуется использовать корректные инструменты. Тестирование производительности можно выполнять множеством различных методов, технические детали, лежащие в основе драйверов тестирования производительности различны, например, для разных архитектур процессоров. В данной работе рассматривается сравнение ряда фреймворков для автоматизации проведения тестирования производительности.

Составив план тестирования, определившись с тестовыми случаями и реализацией тестовых модулей, требовалось проанализировать полученные результаты и составить сравнительные выводы по итогам тестирования производительности.

Объектом исследования в данной работе является множество ППО (промежуточного ПО) для разработки прикладного ПО автономных роботов.

Предметом исследования является производительность подмножества наиболее доступного и используемого МАРППО (многоагентного робототехнического ППО).

Целью исследования является получение результатов тестирования производительности для наиболее доступного и используемого МАРППО.

1 ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Обзор робототехнического ППО

1.1.1 Определение и характеристика робототехнического ППО

Под ППО понимается набор ПО, находящегося по уровню абстракции между ОС (операционными системами) и прикладными приложениями, предназначенного для управления неоднородностью аппаратного обеспечения с целью упрощения и снижения стоимости разработки ПО. В состав ППО входят:

- ПО для осуществления коммуникации между прикладными приложениями, использующими данное ППО, между предоставляемыми компонентами промежуточного слоя данного ППО;
- наборы программных библиотек и их API (application programming interface) для разработки прикладный приложений;
- инструменты для разработки прикладных программ, среди которых возможны собственные компиляторы, менеджеры проектов, системы построения проектов;
- утилиты управления и мониторинга системы, состоящей из ППО и прикладных программ, использующий инфраструктуру ППО.

Особенностями робототехнических систем являются:

- с точки зрения инфраструктуры:
  - модульность;
  - упор на параллельность выполнения процессов системы и распределенность компонентов по различным процессам;
  - надежность и отказоустойчивость;
  - стремление к предсказуемости и соответствию системам реального времени.

- с точки зрения прикладного программирования:
  - большой объем программных библиотек, реализующих алгоритмы, важные в робототехнике, таких как кинематика, компьютерное зрение и распознавание объектов, локализация построение карты окружения и т.д.;
  - большой объем предоставляемых драйверов и программных библиотек, обслуживающих внешние устройства, различную аппаратуру;
  - предоставление ПО для симуляции робототехнических моделей.

Особенности и требования к инфраструктуре робототехнических систем приводят к выводу, что наиболее выгодным является использование распределенной архитектуры и многоагентного подхода. В статье [1] приводятся доводы и рассуждения о преимуществах использования многоагентных фреймворков для разработки робототехнического ППО, а так же примеры многоагентных фреймворков, которые могут использоваться.

Таким образом, многоагентный подход и МАРППО являются наиболее предпочтительными в разработке автономных робототехнических систем и именно их рассмотрение будет в данной работе.

1.1.2 Критерии отбора

Для обсуждения конкретных решений требуется из всего множества существующих фреймворков выбрать наиболее подходящие для данного исследования. Ниже приведены использовавшиеся критерии.

Наличие открытого исходного кода. Для лучшего понимания работы фреймворка, а так же лучшего понимания результатов тестирования и возможного улучшения тестовых задач желательно иметь возможность прочитать исходный код реализации функционала, влияющего на результат тестирования. Это может быть, например, реализация коммуникации между приложениями, которые были написаны при помощи исследуемого фреймворка.

Наличие документации. Написание обоснованных и корректных тестовых задач под конкретный фреймворк очень затруднительно, если отсутствуют инструкций по его использованию, если отсутствует подробное описание API. Без наличия документации разработка приложений становится слишком сложной и корректность их выполнения не гарантируется.

Текущий статус разработки. Проекты, которые больше не поддерживаются разработчиками вполне могут рассматриваться для исследования, но в них скорее-всего используются устаревшие подходы, что приведет к низким показателям производительности.

Архитектура фреймворка. Согласно требованиям к робототехническим системам, описанным в разделе 1.1.1, требуется как минимум распределенная архитектура. Чем ближе архитектура будет к P2P (peer-to-peer, одноранговая сеть), тем надежнее будет все робототехническое ПО. Для данного исследования будут рассматриваться распределенное гибридное и близкое к чистым P2P архитектурам МАРППО.

Наличие инструментов для мониторинга и конфигурации системы. Для анализа ППО и обслуживаемых им приложений требуются такие инструменты, как мониторы используемых ресурсов и системы журналирования. Кроме того, развертывание большого распределенного ПО для проведения тестов является трудоемкой задачей и крайне желательны инструменты конфигурации и развертывания системы на целевой ОС и аппаратном обеспечении.

Поддержка различных языков программирования. Несмотря на то, что в данной работе наибольший интерес представляет именно промежуточный уровень системы, наличие альтернатив между различными языками программирования прикладного слоя робототехнической системы является преимуществом, поскольку позволяет в зависимости от задачи выбрать между, например, низкоуровневым программированием с возможным преимуществом в производительности и высокоуровневыми языками с наличием удобных для разработки интерфейсов и прикладных

библиотек.

В данной работе не важны критерии:

- поддержки ограничений системы реального времени, поскольку это относится не к скорости работы, а к предсказуемости системы. Наличие данного пункта является преимуществом в целом, но относительно производительности оказывается не существенным;
- поддержки конкретных операционных систем, поскольку рассматривается вопрос производительности слоя абстракции между, собственно, ОС и прикладным ПО;
- наличия инструментов симуляции, графических пользовательских интерфейсов и набора прикладных библиотек с реализациями наиболее распространенных алгоритмов, поскольку это относится к функциональному уровню системы, ближе к прикладному. Вопросы производительности отдельных инструментов, которые могут требоваться при разработке, отладке и администрированию робототехнических систем не относятся к проблематике потребления ресурсов на поддержание каркаса, основы системы - фреймворка.

1.1.3 Существующие решения

ROS одно из наиболее распространенных МАРППО, имеется и обширная документация, и открытый исходный код, разработка продолжается, широко используется. Имеет гибридную архитектуру, средства мониторинга и автоматизации развертывания системы. Для разработки по-умолчанию есть возможность использовать C++ и Python 2.7 [2].

MIRA этот проект активно разрабатывается, имеется открытый исходный код и обширная документация. Имеет децентрализованную архитектуру [3], реализован на C++. Имеются возможности для ведения журналирования приложений, мониторы состояния коммуникаций и ресурсов системы. Для разработки предлагается использовать C++, Python и JavaScript [4].

MOOS развивающийся проект, имеется документация и исходный код. На данный момент разрабатывается бета-версия MOOS 10. Общая проблема: клиент-серверная архитектура, которая является спорным решением для разработки робототехнических систем из-за проблем устойчивости ПО к ошибкам. По этой причине в данной работе этот фреймворк рассматриваться не будет [5].

ORoCoS Toolchain и Rock являются очень распространенными фреймворками на основе ORoCoS RTT (Real-Time Toolkit)и OCL (Orocos Component Library) – одного из немногих поддерживающих ограничения реального времени ППО вместе с широко использующимися библиотеками кинематики и динамики, Байесовских фильтров и т.д.. Документация обширная, но из-за удаления сервиса хранения удаленных репозиториев gitorious.org [6] как исходный код, так и документация разбросаны по разным ресурсам и не всегда актуальны. Стабильная разработка в основном ведется над набором инструментов Rock. Используется гибридная децентрализованная архитектура. Разработка ведется на C++, для разработки прикладных программ предоставляются такие языки, как C++, Python, Simulink [1]. Имеются инструменты для развертывания и мониторинга системы [7; 8].

ASEBA является проектом, в котором для программирования прикладного слоя используется концепция языков программирования пятого поколения: GUI, блоки, коннекторы. Разработка является скорее обучающим продуктом с коммерческой составляющей в виде конкретной модели робота thymio. По этой причине рассматриваться в данной работе не будет [9].

SmartSoft разрабатывающийся проект с обширной документацией. Для реализации компонентов используется C++. Практически не используется децентрализация, основной шаблон взаимодействия клиент-сервер. Кроме того, используется многопоточный подход, а не многопроцессный [10], что ставит под вопрос общую устойчивость всей системы. В данной работе рассматриваться не будет.

YARP активно разрабатывающееся МАРППО с открытым исходным кодом. Имеется обширная и подробная документация. Является одним из немногих практически полностью децентрализованных технологий робототехнического ППО, кроме того, поддерживает ограничения систем реального времени. Разрабатывается в основном на C++ и поддерживает такие языки как C++, Python, Java, Octave. Инструменты мониторинга и развертывания приложений имеются [11].

OpenRTM-aist распространенное МАРППО с открытым исходным кодом, является реализацией стандарта RT-middleware [12]. Основная проблема: часть документации, при скромном содержании, на японском языке, а на момент написания работы (апрель – май 2018 года) документация отсутствует ввиду переноса проекта на инфраструктуру GitHub, в связи с этим доступ к старому сайту был убран, по тому же URL (Uniform Resource Locator) находится временная страница проекта. На данный момент разработка не имеет доступной документации [13], авторы на запрос документации не ответили. Архитектура гибридная децентрализованная, имеются инструменты мониторинга, журналирования и развертывания, языки разработки: C++, Java, Python. Рассматриваться не будет по причине отсутствия доступа к документации.

URBI данный проект имеет много недостатков: приостановленная разработка, о чем свидетельствует последнее изменение от 2014 года [14] и не работающий сайт самого проекта [15], централизованная архитектура. Рассматриваться в данной работе не будет.

В таблицах 1.1 и 1.2 отображено соответствие найденного МАРППО предложенным выше критериям.

Таким образом, в исследовании будут учавствовать следующее МАРППО:

- ROS;
- MIRA;

- OROCOS/Rock;
- YARP.

Таблица 1.1 – Соответствие найденного робототехнического ППО выделенным в разделе 1.1.2 критериям (часть 1)

НАЗВАНИЕ	**ОТКРЫТЫЙ КОД**	**НАЛИЧИЕ ПОНЯТНОЙ ДОКУМЕНТАЦИИ**	**ПОСЛЕДНИЕ ИЗМЕНЕНИЯ**
ROS	Да	Да	2018
MIRA	Да	Да	2018
MOOS	Да	Да	2018
OROCOS/Rock	Да	Да	2016
ASEBA	Да	Нет	2018
YARP	Да	Да	2018
OpenRTM-aist	Да	Нет	2016
URBI	Да	Нет	2016

Таблица 1.2 – Соответствие найденного робототехнического ППО выделенным в разделе 1.1.2 критериям (часть 2)

НАЗВАНИЕ	**АРХИТЕКТУРА**	**ИНСТРУМЕНТЫ МОНИТОРИНГА**	**ПОДДЕРЖКА ЯП**
ROS	Гибридная	Да	C++, Python
MIRA	Децентрализованная	Да	C++, Python, JavaScript
MOOS	Централизованная	Да	C++, Java
OROCOS/Rock	Гибридная	Да	C++, Python, Simulink
ASEBA	Распределенная	Да	Собственный язык
SmartSoft	Распределенная	Да	C++
YARP	Децентрализованная	Да	C++, Python, Java, Octave
OpenRTM-aist	Гибридная	Да	C++, Java, Python
URBI	Централизованная	Да	C++, Java, urbiscript

1.2 Тестирование производительности

1.2.1 Постановка задачи и методы

Целью тестирования производительности является демонстрация соответствия системы заявленным требованиям производительности относительно приемлимых показателей времени отклика, требующегося на обработку определенного объема данных [16].

Ниже приведены четыре вида, на которые условно можно разделить тестирование производительности.

Нагрузочное тестирование проводится с целью узнать быстродействие системы при планируемой для системы нагрузке.

Стресс-тестирование проводится с целью узнать нагрузку, при которых система перестает удовлетворять заявленным требованиям.

Конфигурационное тестирование проводится с целью определить быстродействие при различных конфигурациях системы.

Тестирование стабильности проводится с целью определить быстродействие и корректность работы системы при длительной постоянной нагрузке.

Стоит отметить, что приведенное выше разделение является условным: стресс-тестирование является скорее подмножеством нагрузочного, четкой граница между ними нет. Кроме того, при тестировании производительности может использоваться комбинация различных видов тестирования производительности, так как, например, тестирование стабильности можно проводить вместе с нагрузочным.

Ниже приведены метрики, которые могут быть получены в ходе тестирования производительности.

Задержка – время выполнения какой-либо операции. Задержка бывает нескольких видов в зависимости от способа измерения:

- задержка, получаемая как разность тактов ядра процессора до начала и после завершения измеряемого кода пропорционально установленной тактовой частоте процессора;

• задержка реального времени, получаемая как разность значения системных часов до начала и после завершения измеряемого кода.

Разница данных подходов видна в случае обращения к внешним устройствам (устройствам ввода-вывода, вычислительным устройствам, таким как GPU (graphics processing unit)), в случае потери контроля ядра процессора потоком, в котором выполняется измерение (например, в случае перевода потока в состояние ожидания). В случае, если не гарантировано выполнение кода на одном CPU (Central Processing Unit), измерение задержки выполнения данного кода при помощи тактов процессора может быть некорректным.

IOPS (Input/Output Per Second) – скорость обращения к устройству или системам хранения данных, измеряемая в количестве блоков, которое можно записать или прочитать с устройства в единицу времени.

Широта пропускания - объем информации, которое может обрабатываться системой в единицу времени.

В зависимости от предмета тестирования будут различны методы тестирования. В самом простом случае тестирование производительности представляет из себя запуск исследуемой системы и ручное измерение, сбор и анализ характеристик системы. Данный подход имеет следующие недостатки: низкая точность измерения, низкая скорость выполнения тестирования, высокая сложность изменения тестовых сценариев, сложность с вычислением метрик отдельных блоков кода.

Более эффективным способом тестирования производительности является написание и исполнение приложений, которые имеют доступ к интересующему компоненту системы по требуемому интерфейсу:

- вызовов функций на уровне исходного кода рассматриваемого приложения;
- запуска программных модулей на уровне ОС;
- потоков информации на уровне протоколов коммуникации.

Существует множество решений, позволяющих конфигурировать

автоматизированные сценарии тестирования, нагрузку на систему, структурировать и визуализировать результаты тестирования.

Таким образом, возникает проблема выбора инструмента для тестирования производительности МАРППО;

1.2.2 Выбор инструмента для тестирования производительности

1.2.2.1 Критерии выбора

Для тестирования производительности МАРППО будет требоваться доступ на уровне вызова функций из кода программ. В таком случае наиболее подходящим решением будет фреймворк для написания бенчмарков – это программы в изначальном значении (синтетические бенчмарки [17]) для измерения скорости работы процессора. В более общем и подходящем в данном контексте значении – программы для измерения каких-либо количественных характеристик быстродействия системы.

Все МАРППО имеют возможность писать прикладные программы на C++ - эффективном в контексте быстродействия написанных на нем программ. Следовательно, требующийся бенчмарк-фреймворк должен иметь программные интерфейсы для измерения как минимум времени работы блока операторов, написанных на языке C++.

Вышеописанные критерии являются базовыми для выбора инструмента проведения тестирования производительности МАРППО. Для того, чтобы из возможных вариантов выбрать какое-либо подмножество наиболее подходящих, возможны дополнительные критерии для сравнения:

- наличие возможности форматированного вывода результата;
- наличие возможности создавать пользовательские метрики;
- наличие возможности автоматически выполнять блок кода множество раз;
- наличие понятной документации;
- наличие концепции фикстур;
- способ вычисления времени;
- количество зависимостей;

Таким образом, были выделены необходимые критерии для отбора фреймворков для написания бенчмарков на языке C++, а так же приведен список дополнительных критериев для сравнения бенчмарк-фреймворков между собой.

1.2.2.2 Обзор фреймворков для тестирования производительности

В соответствии с базовыми критериями отбора в разделе 1.2.2.2 ниже представлены наиболее освещенные проекты.

Hayai [18] Данный фреймворк является наиболее простым из представленных. Описание тестов происходит при помощи отдельного макроса. Предоставляет возможность выбрать количество запусков теста, количество итераций работы тестируемого блока кода, позволяет компоновать тесты в группы. Вывод результата может производиться в стандартный поток ввода/вывода, либо в файл в форматах json и junit. Подробная документация отсутствует: имеется возможность получить справку из скомпилированного бенчмарка, из которой можно получить информацию о возможных способов запуска бенчмарка, а так же доступна описательная запись на персональном сайте автора [19]. Время вычисляется при помощи системных часов. Имеется поддержка фикстур. Не имеет возможности создавать и использовать пользовательские счетчики, зависит только от кода стандартной библиотеки C++.

Celero [20] Данный фреймворк обладает большим количеством возможностей: вычисление времени выполнения теста во время исполнения, предоставление табличных результатов в формате csv, возможность установки ограничений на время теста. Время вычисляется при помощи системных часов стандартной библиотеки `<chrono>`. Имеет возможность отключать некоторые оптимизации, которые могут влиять на скорость работы тестируемого блока кода. Не имеет возможности определять пользовательские счетчики. Поддержка фикстур имеется.

Nonius [21] Данный фреймворк имеет в сравнении с другими зависимость

от библиотеки Boost, поскольку в реализации `std::chrono` для VS2013 существуют ошибки в реализации. Поскольку тестирование планируется в среде Linux, то в данном контексте это излишняя зависимость. Автоматически считает среднее значение и стандартное отклонение, но не имеет возможности определить пользовательские счетчики. Поддержка фикстур имеется. Имеет возможность вывода результата в стандартный ввод/вывод, а так же форматах csv, junit, html.

Google benchmark [22] Данный фреймворк вычисляет время реальное (wall-clock) на основе машинных часов, так и используя ассемблерную команду `RDTSC` (Read Time Stamp Counter) для аритектуры x86_64, которая возвращает количество тактов ядра процессора с момента включения ядра. Ниже представлены проблемы данного способа вычисления и их возможные решения.

- Из-за влияния работы кэшей процессора будет теряться точность вычислений. Решение: делать несколько итераций вычислений требуемого фрагмента в цикле. Google benchmark самостоятельно выбирает количество итераций выполнения теста на основе затраченного времени: чем меньше была получена задержка, тем с большей погрешностью будет результат, значит требуется больше измерений и, соответственно итераций.
- Проблемы многопоточности, в частности влияние обработки соседних потоков, а так же смена обрабатывающего ядра процессора, на котором, скорее-всего, используется другое значение счетчика тактов. Решение: отслеживание на каком ядре работает, сериализация идентификатора ядра. Решение представленно в статье Intel[23]. В Google Benchmarks данная проблема не учтена. Возможным решением является привязка процесса к конкретному ядру процессора. Для ОС Linux это делается при помощи утилиты `taskset` .

Имеется поддержка фикстур, пользовательских счетчиков, возможность приостанавливать вычисление времени, передача диапазона числовых

аргументов, вывод результата в форматы csv, json, а так же стандартный поток ввода-вывода.

В таблице 1.3 показано сравнение вышеописанных бенчмарк-фреймворков. Таким образом, имея меньше зависимостей, больший функционал и возможность измерения тактов ядер процессора, Google benchmark является наиболее подходящим инструментом для выполнения тестирования производительности МАРППО.

Таблица 1.3 – Соответствие найденных фреймворков для написания бенчмарков на языке C++ выделенным в разделе 1.2.2.1 критериям

	Hayai	**Celero**	**Nonius**	**Google Benchmark**
Вывод	stdio, json, junit	stdio, csv, junit	stdio, csv, junit, html	stdio, csv, json
Счетчики	Нет	Нет	Нет	Да
Итерации	Вручную	Вручную	Автоматически	Автоматически
Документация	Скудная	Подробная	Подробная	Подробная
Фикстуры	Да	Да	Да	Да
Вычисление времени	std::chrono	std::chrono	std::chrono	std::chrono, rdtsc
Зависимости	STL	STL	STL, Boost	STL

1.3 Выводы

Таким образом, в разделе 1 были даны определения ППО и МАРППО, даны их характеристики, а так же был проведен обзор существующего МАРППО результатом которого являются 4 фреймворка, идущие в составе выбранного для тестирования производительности МАРППО, которое будет производиться при помощи бенчмарк-фреймворка Google Benchmark.

2 ПЛАН ТЕСТИРОВАНИЯ

2.1 Предмет тестирования

2.1.1 Факторы влияния на производительность МАРППО

Ниже приведены важные факторы для производительности МАРППО [1].

Тип архитектуры. Большая часть фреймворков использует гибридную архитектуру, поскольку такой подход позволяет избежать ряд проблем, а именно:

- Не требуется инкапсулировать в каждый узел системы общие сервисы, как, например, службу поиска других узлов: эту задачу выполняет специально выделенный узел. Так же поступают и с агенствами - контейнерами, создающие, регистрирующие и хранящие узлы-агенты.
- Уменьшение нагрузки на сеть коммуникаций, поскольку вся нагрузка по общим для всех узлов запросам смещается в сторону обработки на выделенных узлах. Это приводит к появлению критических узлов в системе, в случае отказа которых весь робот может оказаться неспособным продолжать выполнение поставленных задач. Таким образом, в таком подходе требуется отдельно рассматривать производительность таких критических узлов системы, как служба поиска узлов, служба именования и, возможно, другие выделенные сервисы.

В случае, если используется максимально децентрализованная архитектура, то возникают следующие проблемы:

- Увеличивается сложность узлов, а так же нагрузка на обработку данных между узлами. Тем не менее то, насколько эффективно используются возможности фреймворка по взаимодействию между узлами, зависит от разработчика на прикладном уровне.
- Увеличивается нагрузка на систему коммуникации фреймворка. Под системой коммуникации понимается набор технологий и протоколов,

обеспечивающих обмен данными между узлами системы. Часто для обеспечения коммуникации используются отдельные узлы, создаваемые самим фреймворком, будь то брокеры объектных запросов (ORB) при использовании CORBA или реализации шаблона взаимодействия «издатель-подписчик» при помощи топиков (от англ. topic - тема), которые имеют свою внутреннюю реализацию.

Тип взаимодействия между узлами. При использовании многоагентной архитектуры используется несколько разных способов взаимодействия между узлами-агентами:

- порты, реализующие независимое чтение из входных портов, запись во входные и взаимодействие «один ко многим»;
- топики, реализующие шаблон «издатель-подписчик» и взаимодействие «многие ко многим»;
- события, реализующие шаблон «наблюдатель» и асинхронное взаимодействие «один ко многим»;
- сервисы, предоставляющие реактивный способ поведения узлам: возможность отправлять запрос от узла-клиента на узел-сервис, реализуя взаимодействие выполнения удаленных процедур;
- свойства, предоставляющие возможность менять состояние узлов при помощи селекторов параметров агента (get/set), реализация может отличаться в зависимости от архитектуры: гибридная архитектура представляет выделенный узел-сервис - службу свойств, децентрализованная инкапсулирует свойства узлов непосредственно в агентах системы.

Большинство фреймворков дает выбор: какой тип взаимодействия между узлами использовать, но, в случае с реализацией сервисов может возникнуть простой системы до тех пор, пока не работающий или перегруженный обработкой запросов узел, предоставлявший требуемый функционал, не будет восстановлен. Реализация

отдельных типов взаимодействий влияет практически на все аспекты производительности: задержку, вычислительные ресурсы, использование памяти, энергопотребление. Для каждого из фреймворков следует тестировать каждый из способов взаимодействия по всем показателям производительности.

Протоколы коммуникации. Для доставки сообщений обычно используются различные подходы и протоколы. Обычно разработчики реализуют решения на основе стека TCP/IP с некоторыми оптимизациями, например реализация узлов как отдельных потоков внутри процесса-агенства. Данный подход позволяет использовать общую память процесса и ускорить взаимодействие между узлами внутри процесса-агенства, которые могут располагаться в пределах одного вычислительного устройства, что влечет за собой возможность потерять доступ ко всем узлам агенства в случае неисправностей из-за ошибок, допущенных при реализации агентов. Данная проблема решается репликацией агенств на вычислительной системе. Кроме того для доставки сообщений между узлами используются как более высокоуровневые технологии (CORBA, ICE, ACE), так и приближенные к аппаратной части (EtherCAT, I2C, CANBus). Выбор механизма доставки данных между узлами влияет на задержку получения информации, на пропускную способность между узлами, целостность данных, а так же потребление ресурсов вычислительной системы: чем более высокоуровневая технология, тем больше потребление ресурсов системы. Дополнительный функционал на уровне коммуникации, например QoS или шифрование так же влияет на производительность. При наличии такого функционала его следует тестировать отдельно.

Формат сообщений. Сообщения в зависимости от их структуры влияют на производительность. Бинарные типы сообщений имеют меньший объем, а следовательно на их передачу требуется меньше энергии и времени. Структурированные типы, вроде XML и JSON, хорошо поддаются анализу

для разработчика, но могут занимать больше времени на обработку своих данных, а так же требуют больше времени на передачу данных между узлами.

Способ взаимодействия с аппаратурой. Существует два основных способа предоставить интерфейс от аппаратной части системы к прикладному ПО:

- инкапсулировать взаимодействие с аппаратурой в отдельных узлах, отображающих устройства;
- использовать промежуточный слой - сервер, который отвечает на запросы узлов и устройств;
- динамически связывающиеся библиотеки.

От способа обращения будет зависеть отзывчивость системы на внешнее взаимодействие. Кроме того, различные реализации может использовать различное количество ресурсов. Кроме того, в случае использования дополнительного слоя абстракции с клиент-серверным взаимодействием между аппаратным слоем системы и фреймворком является уязвимым подходом с точки зрения устойчивости системы.

В таблице 2.1 показано сопоставление возникающих проблемы и их решений для отобранных для исследования фреймворков [1; 3; 24—27].

Таким образом, рассматриваемые области тестирования:

- централизованные сервисы, если они имеются во фреймворке;
- способы взаимодействия между узлами с целью установления задержки передачи сообщений между узлами;
- коммуникация между узлами многоагентной системы с целью установления пропускной способности, при использовании шифрования - задержку передачи сообщения;
- формат передачи данных между узлами, объема передаваемых данных, скорости извлечения данных в случае, если информация сжимается;

Таблица 2.1 – Реализация критических для производительности задач для отобранных фреймворков

	ROS	**MIRA**	**ORoCoS**	**YARP**
Централизованные сервисы	Сервисы поиска, именования, сервис параметров	Нет	Сервисы поиска, именования	Сервис имен
Взаимодействие между узлами	Топики, параметры, сервисы	Топики, RPC	Порты, сервисы, события, параметры	Порты, топики
Протоколы коммуникации	TCP, UDP, собственный протокол rosserial	IPC, TCP	CORBA, TCP, UDP, SSL,UNIX Sockets, MQueue, EtherCAT, CanOPEN	ACE, TCP, UDP, IPC
Формат сообщений	Бинарный	Бинарный, XML, JSON	Сериализация на основе CORBA	Бинарный
Взаимодействие с аппаратурой	Инкапсуляция в узлах	Инкапсуляция в узлах и RPC-API	Инкапсуляция в узлах и RPC-API	Инкапсуляция в узлах и динамически подключаемые библиотеки

2.1.2 Разбор методов коммуникации МАРППО

2.1.2.1 ROS

Система коммуникации в ROS состоит из двух типов: взаимодействие между узлами при помощи топиков и сервисов [28].

- *Взаимодйствие при помощи топиков* – это широковещательный подход взаимодействия по шаблону «наблюдатель (издатель-подписчик)». Узлы-издатели публикуют «тему» общения – топик – доступную для всех узлов. Все подписавшиеся на опубликованный топик узлы-читатели получают возможность читать отправленные на данный топик сообщения. Факторы, влияющие на производительность:
- *Клиент-сервисное взаимодействие* – это подход, согласно которому

каждый узел может реализовывать сущность RPC сервиса, исполняющего какой-либо процесс по запросу узла-клиента, возможно, с аргументами, и, возвращающий, ответ на запрос, возможно, пустой. Факторы, влияющие на производительность, в сравнении с топиками, идентичные, но, с точки зрения метрик, отдельно требуется рассматривать задержку передачи как для запроса, так и для ответа с разным объемом сообщений для запроса и ответа.

Узлы взаимодействуют напрямую, Мастер-сервис только предоставляет информацию об именах существующих узлов, как DNS (Domain Name Service). Узлы-подписчики посылают запрос узлу, посылающему сообщения по определенному топику, и устанавливает с ним соединение по протоколу TCPROS [29], использующий в себе стандартные TCP/IP сокеты [30].

Управление потоками данных и приоритетом сообщений между узлами в ROS отсутствует. Этот механизм планируется реализовать в ROS 2.0.

Таким образом, факторы, влияющие на производительность коммуникации в ROS:

- тип взаимодействия:
  - через топики;
  - через сервисы.
- количество взаимодействующих узлов
  - подписчиков для топиков;
  - клиентов для сервисов.
- объем сообщений;
- размер буферов каналов коммуникации.

2.1.2.2 YARP

В YARP основной способ передачи информации - система портов. Как и в ROS, это механизм, реализующий шаблон «наблюдатель». Отличие от ROS в том, что реализован этот шаблон через подход портов - устройства потокового

ввода и вывода. Кроме того, как и в ROS, имеется возможность использовать механизм RPC между узлами системы. Ниже разобраны оба механизма.

- *Порты* – это именованные объекты, которые обеспечивают доставку сообщений множеству других портов, подписанных на данные порты по данному имени. Один узел может иметь множество портов внутри.

  Порты в YARP реализованы как объекты потоков, в которые можно писать данные и с которых данные можно считывать. Этот подход позволяет разделить отправителя данных и принимающего данные, таким образом, соответствующим портам не требуется знать много информации друг о друге. Это с одной стороны повышает абстрактность, позволяя работать со множеством портом сразу и допускать, например, перезапуск или временную неработоспособность какого-либо узла, с другой влияет на производительность, так как полностью от зависимости передающего и получателя сообщения избавиться нельзя. В частности, порт знает, когда передача сообщений конкретному порту получателю закончена: только после этого посылающий может приступать к передаче следующему адресату. Объекты, которые пересылаются между портами, не копируются. Эту проблему можно решить разными способами [31], например, хранением сообщений в очереди на отправку.

  Порты бывают двух типов: обычные и с буфером сообщений. Второй тип отличается от первого наличием очереди сообщений, в слотах которой под сообщения резервируются сообщение перед отправкой. В отличии от обычных портов, порты с буфером сообщений теперь отвечают за время жизни объектов, которые требуется пересылать между портами, предоставляя порту самому определять, например, очередность передачи сообщений. Например, поскольку, находящееся в очереди сообщение можно модифицировать, порт с буфером может определить сообщение до и после изменения данных и не станет пересылать старое сообщение, если не выставлен определенный флаг, указывающий на обязательность

передачи всех сообщений. В отличии от ROS размер буфера порта не фиксированный и зависит от количества сообщений в очереди: длина очереди увеличивается если количество сообщений превышает некоторый порог.

Обычные порты `yarp::os::Port` и с буфером `yarp::os::BufferedPort` сильно между собой отличаются и разумно сравнить их между собой для различных данных и различной нагрузке портов-получателей. Примером различия в производительности может служить возможность, если не выставлен специальный флаг, перейти к отправке других сообщений другим портам-адресатам пока все порты-получатели конкретного сообщения заняты.

- *RPC* – механизм синхронизированной передачи данных между портами. Формально, поскольку все порты имеют возможность двусторонней связи, не требуется никаких дополнительных абстракций для реализации подобного подхода в YARP. Тем не менее, разработчики выделили в отдельные классы `yarp::os::RpcClient` и `yarp::os::RpcServer` – наследники базового класса всех портов `yarp::os::Contactable` – для наличия готового инструмента с обеспечением целостности и синхронности передачи данных. В отличии от ROS, в YARP вся коммуникация идет через порты, через единообразный интерфейс устройства потокового ввода и вывода.

В YARP не требуется описывать данные в особых форматах. В самом простом случае, разработчик реализует обычный C++ класс и передает его шаблонным параметром при создании порта. Возможны трудности в случае, если передаются данные между различными машинными архитектурами, если данным требуется не очевидная сериализация. В целом, большинство проблем решаемы при помощи предоставляемых YARP макросами, интерфейсами и классами-обертками.

Для передачи данных может использоваться множество протоколов, а так

же «транспортов» (carriers) - классов-обработчиков объектов-сообщений для передачи между портами. Есть множество транспортов, реализованных в YARP и доступных для работы со распространенными транспортными протоколами:

- TCP;
- UDP;
- multicast – широковещательный протокол, наиболее эффективная реализация для передачи сообщений множеству YARP-портов;
- shared memory – передача данных в пределах одной машины используя разделяемую область оперативной памяти.

Преимущество YARP состоит в том, что способ транспортировки сообщения в соединении изменяем в любой момент времени как реализацией внутри узлов системы, так и внешним конфигурированием узлов, что позволяет гибко управлять коммуникацией внутри робототехнической системы.

Отдельно стоит указать возможность управлять приоритетом соединений, т.е. наличие QoS (Quality of Service). Для определенного соединения можно назначить один из четырех возможных приоритетов передачи данных.

Узлы взаимодействуют напрямую, взаимодействия с сервисом имен можно избежать, задавая имена портам при инициализации самостоятельно. В таком случае контроль уникальности идентификаторов ложится на разработчика.

Таким образом, основным способом связи в YARP являются порты, на производительность передачи данных будут влиять следующие факторы:

- тип порта:
  - обычный;
  - с буфером
  - реализующий синхронизированную передачу данных;
- тип транспорта (carriers):
  - tcp;

– udp;
– mcast – широковещательный транспорт;
– shmem – транспорт при помощи локально разделяемой памяти (используется ACE - Adaptive Communication Environment);
– local – использование разделения памяти внутри одного процесса;
– fast_tcp – реализация tcp транспорта без подтверждения пакетов о доставке.

- размер данных;
- количество портов-получателей;
- приоритет соединений, задаваемых при помощи QoS.

2.1.2.3 MIRA

В MIRA предоставляются стандартные два вида коммуникации: отправка сообщений между узлами по шаблону «наблюдатель» и удаленный вызов методов. В терминологии MIRA, узлами являются модули (units), объединяющиеся в домены. Модули компилируются и используются как разделяемые библиотеки [32], что позволяет загружать нужные модули во время исполнения робототехнической системы на прикладном уровне. Основанный на разделяемом коде, высоком уровне абстракции и сериализации объектов классов при помощи сохранении информации о классе – рефлексии, как аналогичный механизм, использующийся в Java для получения метаинформации о классах объектов во время исполнения программ – подход позволяет уменьшить затраты ресурсов памяти в прикладных приложениях, а так же предоставляет разделение ответственности для разработчиков, позволяя максимально абстрактно реализовывать модули, подобно плагинам.

В отличии от ROS и YARP, MIRA не позволяет как-либо управлять очередью сообщений. Известно, что эта очередь есть и имеется возможность сохранять сообщения в памяти на определенный промежуток времени, который может указать разработчик. Это позволяет получать доступ к

«прошлому» - к сообщениям, которые пришли какое-то время назад.

Для передачи сообщений между процессами используется протокол TCP, внутри одного процесса – разделение памяти.

Объекты сообщений передаются между модулями в бинарном виде путем сериализации объектов, созданных при помощи шаблона «фабрика объектов». Вся метаинформация указывается в переопределяемом методе `template <typename Reflector> void mira::reflect(Reflector& r)` для любого объекта MIRA, таким образом, позволяя передавать любые объекты в среде коммуникации MIRA.

Управление потоками данных, приоритетами сообщений между модулями отсутствует.

Таким образом, основными факторами, возможно, влияющими на производительность системы коммуникации MIRA являются:

- способ передачи данных:
  - по шаблону «наблюдатель»;
  - при помощи вызова удаленных процедур.
- локализация модуля:
  - в одном процессе (используется разделяемая память);
  - в разных (используется протокол TCP).
- объем пересылаемых данных;
- время хранения сообщений в буфере соединения.

2.1.2.4 OROCOS

ORoCoS Toolchain является наиболее сложным МАРППО из всех представленных ранее, т.к. в нем сочетаются все вышеописанные идеи. Первый релиз ORoCoS Toolchain в 2006 году, раньше всех исследуемых МАРППО.

ORoCoS Toolchain состоит из нескольких проектов, среди которых нам наиболее важен ORoCoS RTT – библиотеки для разработки компонентов,

загружающихся во время исполнения приложения. Эта же идея используется в MIRA. Компоненты ORoCoS RTT являются наследниками класса `RTT::TaskContext` и переопределяют ряд методов (hooks), определяющих основу поведения RTT-компонента. Данные компоненты являются аналогом узлов ROS или YARP, модулей MIRA. В отличии от MIRA, ORoCoS имеет возможность написания самостоятельных приложений с отдельной точкой входа.

Методов коммуникации в ORoCoS RTT все так же два: система именованных портов для передачи сообщений и вызов удаленных процедур. Разница состоит во множестве способов применять данные инструменты. По доступным возможностям ORoCoS RTT превосходит YARP. Ниже, основываясь на документации разработчиков [33], приведено описание методов коммуникации не вдаваясь в подробности API ORoCoS RTT, например, сигналов-событий (Signal event handler) и деятельностей (Activities), которые, расширяют возможности реакции компонентов на внешнюю среду и, как следствие, разные способы вступать в коммуникацию. Тем не менее, эти способы реакции не являются методами передачи сообщений.

- *Именованные порты* – аналогично портам в YARP, ORoCoS RTT порты реализуют шаблон «наблюдатель», где на порты-издатели подписываются порты-подписчики. Разница с YARP в том, что в ORoCoS RTT нет деления на «обычные» и «буферизованные» порты, но есть разделение на исходящие `RTT::OutputPort<typename T>` и принимающие порты `RTT::InputPort<typename T>` с соответствующе разными интерфейсами. В то же время, оба типа портов в ORoCoS RTT могут настраивать буферизацию, потокобезопасность и инициализацию соединения пользователем при создании самого соединения.
- *Вызов удаленных методов* – аналогично другим МАРППО этот механизм в ORoCoS позволяет вызывать удаленную процедуру, передавая в нее аргументы и получая результат. Вызов удаленных процедур может производиться двумя способами: при помощи непосредственно вызова

процедур и при помощи сервисов.

Вызов операции (operation calling) отличие главным образом от других подходов в том, что в ORoCoSэто именно процедура, которая может выполняться как в потоке компонента, предоставляющего эту процедуру, так и в потоке клиента, эту процедуру вызывающего. В случае выполнения процедуры в потоке клиента разработчику требуется позаботиться о потокобезопасности разделяемых данных в компоненте клиента. Кроме того, в ORoCoS RTT имеется два подхода к вызову процедур и получению результата: ожидать возврата результата, заблокировав текущий поток выполнения, либо передать ожидание и обработку результата соответствующему объекту-обработчику событий. Последний подход так же требует от разработчика аккуратности при работе с разделяемыми данными, но при этом дает возможность уменьшить реальное (wall clock) время обработки результата.

Сервисы отличаются от вызова операций по-сути возможностью именовать сервис, который предоставляет набор операций, которые можно у сервиса вызвать. Это предоставляет возможность компонентам искать другой компонент-сервис по имени во время исполнения приложения.

В ORoCoS RTT могут передаваться любые пользовательские данные. Стандартные типы языка C++ доступны по-умолчанию, но более сложные требуется описать при помощи библиотеки `boost::serialization`.

Особенностью ORoCoS является возможность использования CORBA для передачи данных, а так же возможность использовать в качестве транспорта для внутрипроцессного взаимодействия `MQueue`. Причем CORBA может использовать протокол OOB (Out-Of-Band) для передачи данных при помощи механизма MQueue. Это позволяет:

- следить за прерванными соединениями;
- быть уверенным, что принимающий поток создается строго после

исходящего, а так же корректно закрывает исходящий поток, если не получилось создать принимающий.

В целом, совместное использование MQueue и CORBA позволяет добиться большей надежности соединений, но путем затрат ресурсов на инфраструктуру CORBA.

ORoCoS предоставляет создавать объект «политики соединения» (connection policies) для конфигурирования соединения между портами. В частности, именно в объекте политики соединения указывается транспорт для сообщений. Тем не менее, ORoCoS позиционируется как наилучшее решение для приложений с внутрипроцессным взаимодействием [34]. Для межпроцессного взаимодействия и для передачи данных по сети требуется использовать дополнительные протоколы и подходы, например, CORBA или, например, Qt TCP Sockets.

В отличии от YARP, ORoCoS Toolchain не предоставляет возможностей для контроля QoS. На практике, параметры соединений можно изменять, но на уровне конфигурирования транспорта. Политики соединений позволяют лишь указать тип транспорта и размер буфера.

Таким образом, для производительности ORoCoS RTT важны следующие факторы:

- локализация компонентов: для связи компонентов в разных процессах потребуется использовать другой транспорт;
- используемый транспорт:
  - использование общей памяти - транспорт по-умолчанию;
  - MQueue;
  - CORBA;
  - MQueue при помощи CORBA OOB.
- размер данных;
- размер буферов соединений;
- подход к взаимодействию компонентов:

Таблица 2.2 – Факторизация параметров для тестирования производительности ROS

Тип взаимодействия	**Размер сообщения**	**Количество подписчиков или клиентов**	**Размер буфера**
Издатель/ подписчик	1 КБ	1	1
Сервисы	4 КБ	2	10
	16 КБ	4	100
	64 КБ	8	1000
	256 КБ	16	10000
	1 МБ	32	
	4 МБ		
	16 МБ		
	64 МБ		

– используя порты;

– используя вызов операций;

– используя сервисы.

2.2 Описание тестовых случаев

Исходя из факторов, описанных в разделе 2.1.2, можно составить набор сценариев для проведения тестирования производительности.

2.2.1 ROS

В столбцах таблицы 2.2 приведены различные значения по выделенным ранее факторам для тестирования. Таким образом, множество всех сценариев тестов производительности – это все возможные подмножества из возможных значений факторов таблицы. 2.2.

2.2.2 YARP

В столбцах таблицы 2.3 приведены различные значения по выделенным ранее факторам для тестирования. Таким образом, множество всех тестов, это все возможные подмножества из возможных значений факторов из столбцов

Таблица 2.3 – Факторизация параметров для тестирования производительности YARP

Тип порта	**Протокол**	**Размер сообщения**
Обычный	tcp	1 КБ
С буфером	udp	4 КБ
RPC	shmem	16 КБ
	fast_tcp	64 КБ
		256 КБ
		1 МБ
		4 МБ
		16 МБ
		64 МБ

таблицы 2.3.

2.2.3 MIRA

В столбцах таблицы 2.4 приведены различные значения по выделенным ранее факторам для тестирования. Таким образом, множество всех сценариев тестов производительности – это все возможные подмножества из возможных значений факторов таблицы 2.4.

2.2.4 ORoCoS

В данном случае имеется четкое разделение на взаимодействие компонентов внутри одного процесса и между процессами. В таком случае, образуется 4 возможных базовых сценариев тестирования:

- внутрипроцессное взаимодействие при помощи разделяемой памяти;
- межпроцессное взаимодействие:
  - MQueue;
  - CORBA;

Таблица 2.4 – Факторизация параметров для тестирования производительности MIRA

Подход к коммуникации	**Реализация модуля**	**Размер сообщения**
Шаблон наблюдатель	В одном процессе (разделяемая память)	1 КБ
RPC	В разных процессах (TCP)	4 КБ
		16 КБ
		64 КБ
		256 КБ
		1 МБ
		4 МБ
		16 МБ
		64 МБ

– CORBA + MQueue;

Таким образом, в таблице 2.5 приведены различные значения по выделенным ранее факторам для тестирования, а множество всех сценариев тестов производительности – это все возможные подмножества из возможных значений факторов таблицы, не считая первую половину столбца 1, который представлен для разделения расположения компонентов: в одном или в разных процессах.

2.3 Реализация

2.3.1 Создание тестового окружения

Для создания единого окружения для тестирования производительности всех МАРППО удобно использовать технологию контейнерной виртуализации на уровне ОС, например, Docker. По сравнению с программной виртуализацией при помощи гипервизора, виртуализация на уровне ОС требует гораздо меньше накладных расходов на абстрагирование за счет уменьшение и упрощение слоев между ОС предоставляющей сервис и гостевым приложением. В Docker

Таблица 2.5 – Факторизация параметров для тестирования производительности ORoCoS Toolchain

Протокол коммуникации		**Тип взаимодействия**	**Размер сообщения**	**Размер буфера**
Расположение	**Протокол**			
Внутри процесса	Разделяемая память	Порты	1 КБ	10
Между процессами	MQueue	Удаленный вызов операции	4 КБ	100
	CORBA	Сервис	16 КБ	1000
	CORBA+ MQueue		64 КБ	2000
			256 КБ	5000
			1 МБ	10000
			4 МБ	
			16 МБ	
			64 МБ	
			256 МБ	

для этого используется механизм ядра Linux - cgroups, изолирующий набор ресурсов компьютера для процессов.

Таким образом, при помощи данного подхода можно организовать изолированные Docker-контейнеры с одним окружением для каждого из рассматриваемых МАРППО.

Для этого был реализован основной Docker-образ `ubuntu-dev`, который основан на ОС Ubuntu 16.04, а так же содержит ряд пакетов:

- для удобства управления и конфигурирования системы (locales, lsb-release);
- для возможности работать с GUI приложениями (ssh, xorg, xauth);
- git для работы с удаленными репозиториями систем контроля версий;
- для удобного написания кода программ (tmux, ranger, vim);
- для сборки и компиляции программ на C++ (build-essential, cmake, pkg-config);
- htop для мониторинга состояния системы;

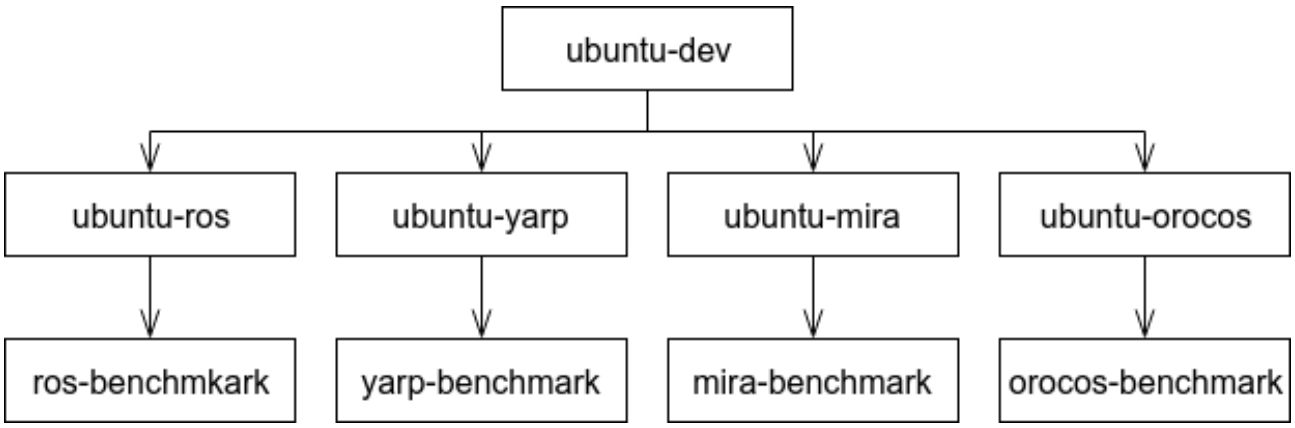

Рис. 2.1 – Иерархия использования Docker-образов.

- пакеты с библиотеками libboost-all-dev и libxml2-dev, использующиеся для компиляции программ для исследуемых МАРППО.

Итоговый Docker-file образа `ubuntu-dev` представлен в листинге А.1.

Используя созданный образ как основу, были созданы 4 образа для каждого из рассматриваемых МАРППО, Docker-файлы которых представлены на листингах А.2, А.3, А.4 и А.5.

Сложности возникли только с ORoCoS Toolchain, поскольку с 2015 года был удален основной репозиторий проекта. После этого исходный код постепенно переносился на инфраструктуру GitHub, но, к сожалению, в сценариях сборки проекта оставалось много зависимостей на недоступные сетевые хранилища исходного кода или библиотек. Кроме того, из-за различия в версиях некоторых пакетов ОС Ubuntu 16.04 с более старыми версиями ОС, возникали ошибки сборки проекта. Данная проблема решалась сборкой в три итерации, после первых двух требовалось ручное исправление промежуточных файлов конфигурации. Результат работы сценариев сборки проекта двух итераций был заархивирован и передается при сборке Docker-образу. Из-за сложностей, возникших при разрешении задачи сборки ORoCoS Toolchain и составления Docker-файла, была составлена подробная инструкция на английском языке. На рисунке 2.1 показана иерархия связей Docker-образов.

2.3.2 Используемое API Google benchmark

Для разработки бенчмарков, согласно изложенному в разделе 1.2.2.2, используется Google Benchmark. Для составления тестов использовались нижеописанные концепции.

Фикстуры – это класс-наследник от `benchmark::Fixture` с переопределенными методами:

- `void SetUp(const benchmark::State &state)` – функция, подготавливающая данные перед началом каждого теста. Примеры применения:
  - создание сообщений определенного размера;
  - установка соединений.
- `void TearDown(const benchmark::State &state)` – функция, вызывающаяся после окончания всех итераций тестирования. Может использоваться для корректного закрытия соединений и завершения потоков.

Фикстуры удобны для создания единого окружения для отдельных бенчмарк-тестов, позволяя комбинировать классы фикстур между различными тестами, а так же переиспользовать код.

Бенчмарк-тест – зарегистрированная для выполнения фреймворком функция. Пример описания приведен в листинге 2.1. В данном описании сразу используется фикстура `FIXTURE_NAME`, для подготовки данных для теста.

Листинг 2.1 – Описание бенчмарка с фикстурой

```
1 BENCHMARK_DEFINE_F(FIXTURE_NAME, BM_TEST)(benchmark::State& state) {
2   for(auto _ : state) {
3     //                -
4   }
5 }
6 BENCHMARK_REGISTER_F(FIXTURE_NAME, BM_TEST)
7   ->Apply(CustomArguments);
```

Кроме того, в бенчмарк-тесты можно передавать аргументы. Передача аргументов может выполняться разными способами, но поскольку значения факторов в рассматриваемых случаях очень различны,

стандартные методы Google Benchmark не подходят. Для передачи произвольного набора аргументов используется метод `Apply`, в который передается идентификатор функции, которая формирует список кортежей аргументов. Пример задания произвольных значений приведен в листинге 2.2.

Листинг 2.2 – Описание функции, формирующей список кортежей аргументов для бенчмарк-теста

```
static void CustomArguments(benchmark::internal::Benchmark* b) {
     for (int topics = 1; topics <= 32; topics *= 2)
         for (int size = (1 << 10); size <= 1 << 28 ; size *= 2)
             for (int buffer = 1; buffer <= 100000; buffer *= 10)
                    b->Args({topics, size, buffer});
}
```

По итогу выполнения данной функции в тесты будут передаваться кортежи параметров, например `(1, 1024, 1)`, `(1, 1024, 10)` и т.д. Для изъятия аргументов для текущего теста используется объект типа `benchmark::State`, обращение к которому через метод `range(index)`, вернет значение аргумента текущего кортежа аргументов по индексу `index`.

Пользовательские счетчики – это возможность по окончанию теста записать значение пользовательского счетчика в результат выполнения теста. Для этого в поле хэш-таблицы `counters` объекта типа `benchmark::State` передается имя (ключ) и значение пользовательского счетчика: `state.counters["bm_id"] = bm_id`. Данные в хэш-таблице хранятся типа **`double`**, таким образом важно следить за тем, чтобы не терялось значение счетчика при сохранении результата. Примером может быть сохранение идентификатора бенчмарка в сообщении, для обработки задержки передачи данных на другом узле. Идентификатором является UNIX-time в наносекундах. Изначально идентификатор был типа **`unsigned long long`**, но после преобразования в **`unsigned long double`** может упасть точность до последних 4 цифр. Это следует учитывать при обработке результатов теста.

Управление временем – это возможность приостанавливать замер времени

выполнения теста. Выполняется соответствующими остановке замера и возобновлению методами `PauseTiming()` и `ResumeTiming()` объекта типа `benchmark::State`.

Указание способа вычисления времени – это возможность указать фреймворку считать время выполнения теста при помощи часов реального времени. Это важно в тех методах, в которых есть многопоточное исполнение, или ожидание ответа. Указать данное требование можно вызвав у бенчмарка метод `UseRealTime()`.

Пример запуска всех зарегистрированных бенчмарк-тестов показан на листинге 2.3.

Листинг 2.3 – Пример запуска всех зарегистрированных тестов при помощи Google Benchmark

```
1 int main(int argc, char** argv) {
2   benchmark::Initialize(&argc, argv);
3   if (benchmark::ReportUnrecognizedArguments(argc, argv)) return 1;
4   benchmark::RunSpecifiedBenchmarks();
5
6   return 0;
7 }
```

Google Benchmark имеет макрос, который генерирует представленную функцию точки входа в листинге 2.3, но в рассматриваемых фреймворках требуется инициализировать узлы во время начала исполнения исполняемого модуля, либо вызывать тесты в методах-обработчиков объектов классов-модулей, например, для MIRA.

При инициализации исполнения бенчмарк-тестов передаются аргументы загрузки исполняемого модуля. Для рассматриваемой задачи тестирования важны следующие пары «ключ-значение» запуска:

–*benchmark_out* – определяет URL файла с результатами тестирования.

–*benchmark_out_format* – определяет формат файла с результатами тестирования. Может быть определен как `console`, `csv` и `json`. Поскольку произвольный вывод консоли сложно поддается разбору и анализу, а csv файл содержит в себе пользовательские счетчики только первого бенчмарк-теста, то единственным подходящим вариантом будет

`json`.

–*benchmark_repetitions* – указывает сколько раз требуется запустить каждый тест. В данном исследовании выборка будет состоять из 10 значений, хотя, анализатор, описываемый в разделе 2.3.4 может обрабатывать выборку от 2 до 40 значений.

Таким образом, были определены необходимые интерфейсы и подходы для реализации бенчмарк-тестов для каждого отдельного МАРППО.

2.3.3 Реализация бенчмарк-фреймворка для MIRA

Поскольку

2.3.4 Автоматизация обработки результатов

2.3.4.1 Описание решения

Результатом выполнения бенчмарк-тестирования является множество json-файлов (не менее одного), в которых записаны результаты тестирования. Для автоматизации обработки множества результатов тестирования, а так же их визуализации была реализована программа, которая обрабатывает все указанные json-файлы с результатами и формирует отчет с таблицами и графиками результатов тестирования в формате rmarkdown. Исходный код программы представлен в приложении В.

Обработку результатов можно разбить на следующие итерации:

1. Анализ файла конфигурации `merge_config.xml` и формирование промежуточного файла `full.json`, содержащего список всех бенчмарк-тестов со всеми полученными значениями измерений.

   Исходные json-файлы можно разделить на два типа: первичные и вторичные.

   Первичный – это json-файл, полученный исполнением непосредственно Google Benchmark исполняемого модуля. В данном json-файле располагается список `benchmarks` с описанием данных каждого

проведенного теста в формате «ключ-значение». Наиболее важные поля:

- `name` – название теста с возможными суффиксами «_mean», «_stddev» и «_median». Суффиксы добавляются средствами Google Benchmark, если указан ключ для повторений тестов и означают соответственно среднеарифметическое результатов, среднеквадратическое отклонение и медианное значение. Поскольку анализ данных будет производиться отдельно, то записи, содержащие в имени данные суффиксы стоит исключить при обработке.
- `real_time` – задержка, основанная на системных часах. При указании замера реального времени, данное значение будет указывать на реальное время в наносекундах, которое было затрачено на выполнение теста.
- `CPU_time` – задержка, основанная на количестве циклов процессора. Наиболее точный возможный показатель, но т.к. большинство методов коммуникации в рассматриваемых МАРППО используют технологии многопоточного программирования, данный результат не может считаться адекватным во всех тестах: данный результат требуется рассматривать вместе с задержкой реального времени.
- `bytes_per_second` – широта пропускания данных, которое вычисляется на основании количества итераций, количества переданных байтов в отдельный счетчик `SetBytesProcessed()` у объекта типа `benchmark::State` и затраченного реального времени. Результат представляется в количестве байтов в секунду.
- `bm_id` – уникальный идентификатор теста. Данное поле задается разработчиком, Google Benchmark не предусматривает никакой автоматической идентификации тестов. Данное поле требуется для связи с результатами производительности полученными на иных узлах связи, отличных от того, на котором работает Google

Benchmark.

Вторичный – json файл с результатами тестирования полученными каким-то отличным от использования Google Benchmark образом. Например, путем вычисления реальной задержки передачи данных на узлах-получателях. Узел-получатель записывает значения измерений в формате «измерение-значение», а для связи с узлом-отправителем, на котором исполняется Google Benchmark, используется уникальный идентификатор теста. В общем случае, не для каждой записи из вторичного json найдется соответствие в первичном, потому что Google Benchmark самостоятельно определяет какие итерации следует пропустить. Как следствие, различие в значениях, полученных во вторичных json-файлах, следует дополнительно объяснять большим количеством переданных сообщений, а так же большим разбросом в значениях. Последнее объясняется тем, что первые итерации измерения производительности скорее-всего не будут адекватными из-за отсутствия часто повторяющихся команд в кэше процессора. Google Benchmark автоматически регулирует количество итераций и набор итераций, которые идут результатом выполнения теста.

В общем случае, для связи первичного и вторичных json-файлов рекомендуется использовать поле `bm_id`, но так же допускается конфигурирование связи между файлами по другому имени поля.

Исходные json-файлы, их тип и ключ связи для вторичных json-файлах указываются в файле конфигурации `merge_config.xml`. Пример файла конфигурации приведен в приложении С.1

Все измерения в найденных по указанному ключу записях из вторичных файлов вставляются в записи первичного json-файла. После слияния первичных json-файлов с соответствующими им вторичными, итоговый файл `full.json` получается путем объединения записей всех первичных json-файлов.

2. Для каждого бенчмарк-теста из списка, полученного на предыдущей итерации, формируется список для бенчарк-тестов без повторений имен (на предыдущей итерации, поскольку тесты могли быть запущены несколько раз, имя записи теста могло повторяться пропорционально количеству запуска тестов) со сгруппированными значениями измерений. Таким образом, сформирована структура данных, которая для каждого набора факторов содержит результаты измерений пропорционально выборке эксперимента. Структура сохраняется в файл `benchmark_processed.json`

3. Далее все результаты статистически обрабатываются: находится среднее значение измерения и среднеквадратичная ошибка среднеарифметического. Результаты записываются в файл `benchmarks_results.json`.

4. Предпоследней итерацией является разбиение результатов, путем получения различных комбинаций тестов, основываясь на определяющих их факторах. Для определения информации о факторах тестирования используется описание факторов в конфигурационном файле первой итерации.

   На каждом шаге выбирается переменный фактор, являющийся абсциссой на графиках результатов тестирования. Для всех остальных факторов требуется найти все возможные фиксированные комбинации, относительно которых будет изменяться выбранный переменный фактор.

   Таким образом будет сформировано множество таблиц, представляющих зависимость результатов измерений от выбранного переменного фактора и фиксированных остальных.

   Результирующий список описания табличных значений записывается в файл `tables.json`.

5. Финальной итерацией является трансляция описаний табличных значений

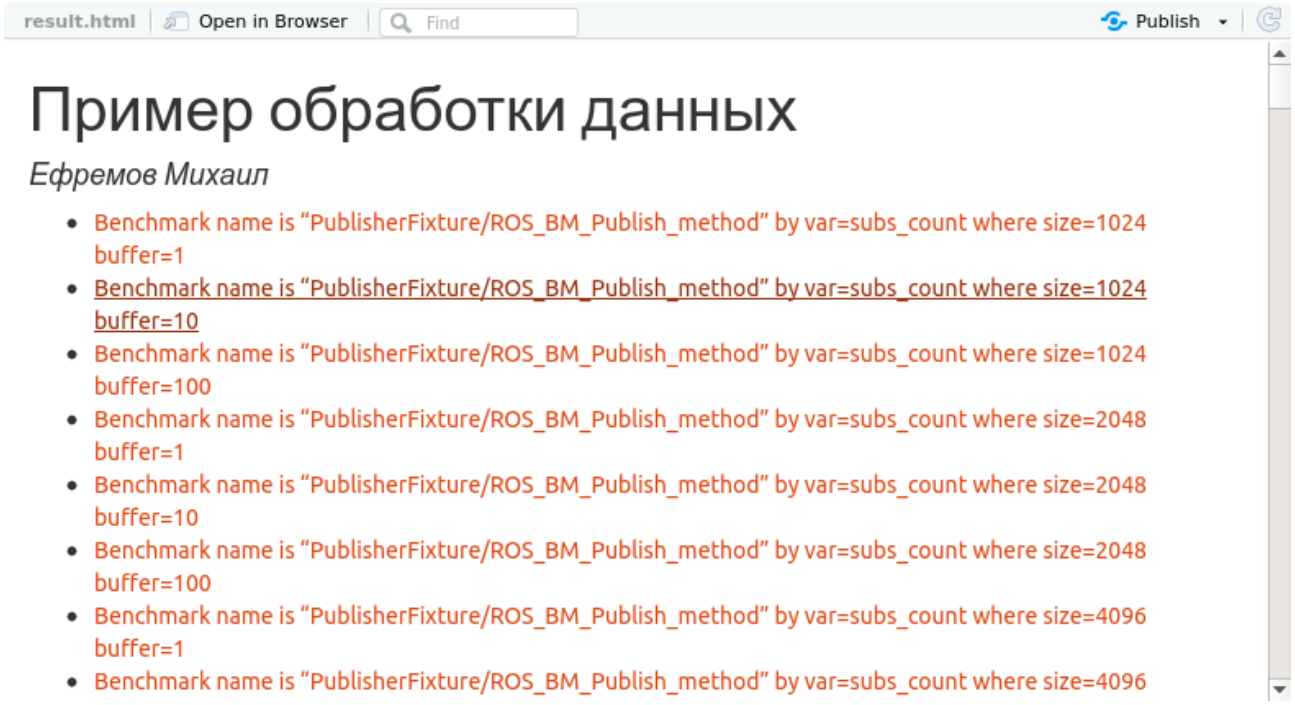

Рис. 2.2 – Содержание, состоящее из гиперссылок на соответствующие результаты тестов

результатов тестирования производительности из файла `tables.json` в rmarkdown файл `result.rmd`. Для конфигурации именования измерений и факторов, а так же описания генерируемых графиков, используется конфигурационный файл `rmd_config.xml`, пример которого представлен в приложении С.2.

Таким образом, было реализовано приложение, принимающее в качестве входных данных множество результатов измерений производительности в формате json, а так же двух конфигурационных файлов с описаниями исходных данных и описанием выходного результата. Результатом работы приложения является rmarkdown-файл, готовый к преобразованию в html страницу при помощи транслятора knit.

2.3.4.2 Примеры работы

На рисунках 2.2, 2.3, 2.4 и 2.5 представлены скриншоты html-страницы, полученной из rmarkdown-файла – результата работы анализатора на некоторых исходных данных.

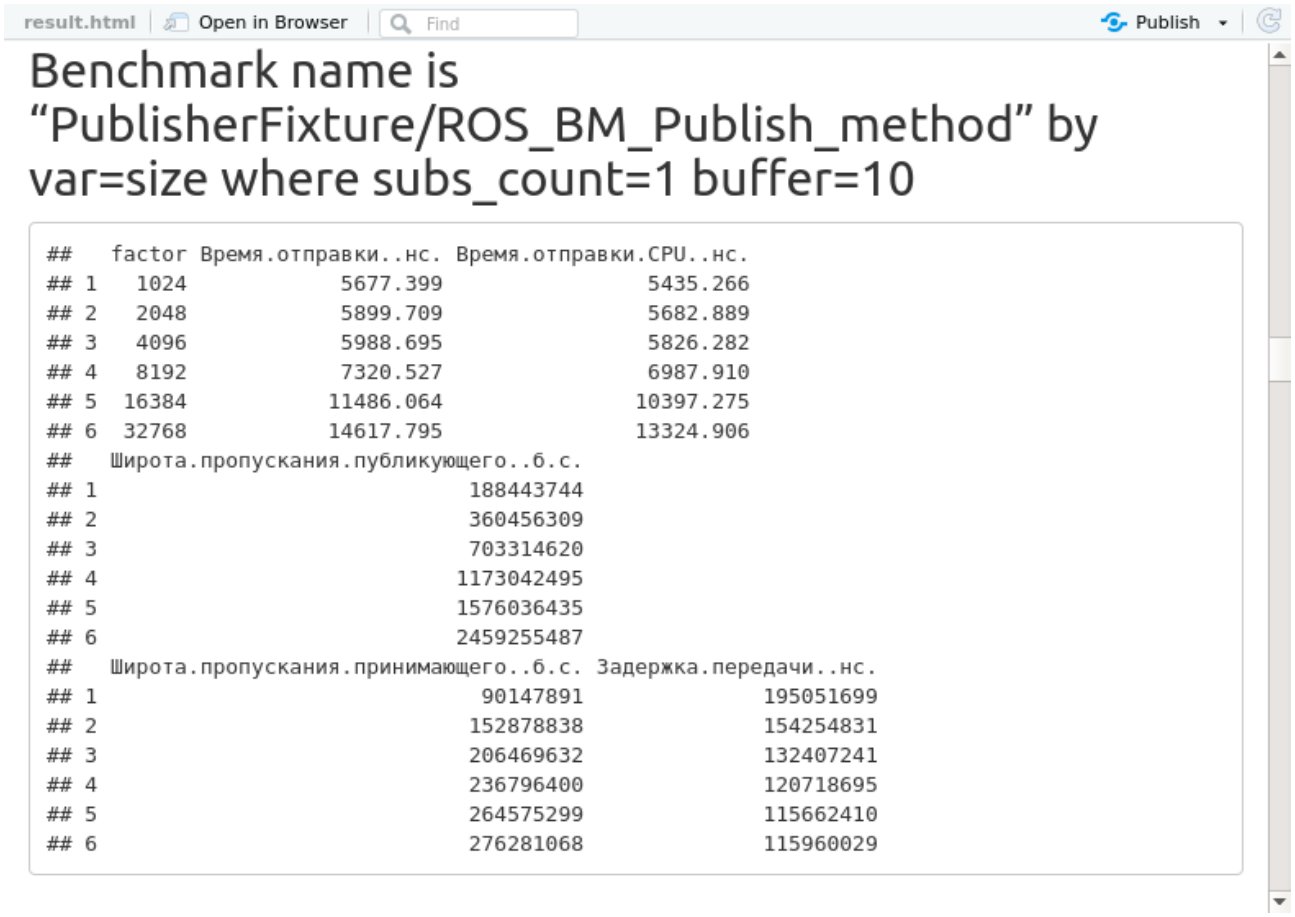

Рис. 2.3 – Таблица средних результатов измерений для переменного фактора «size»

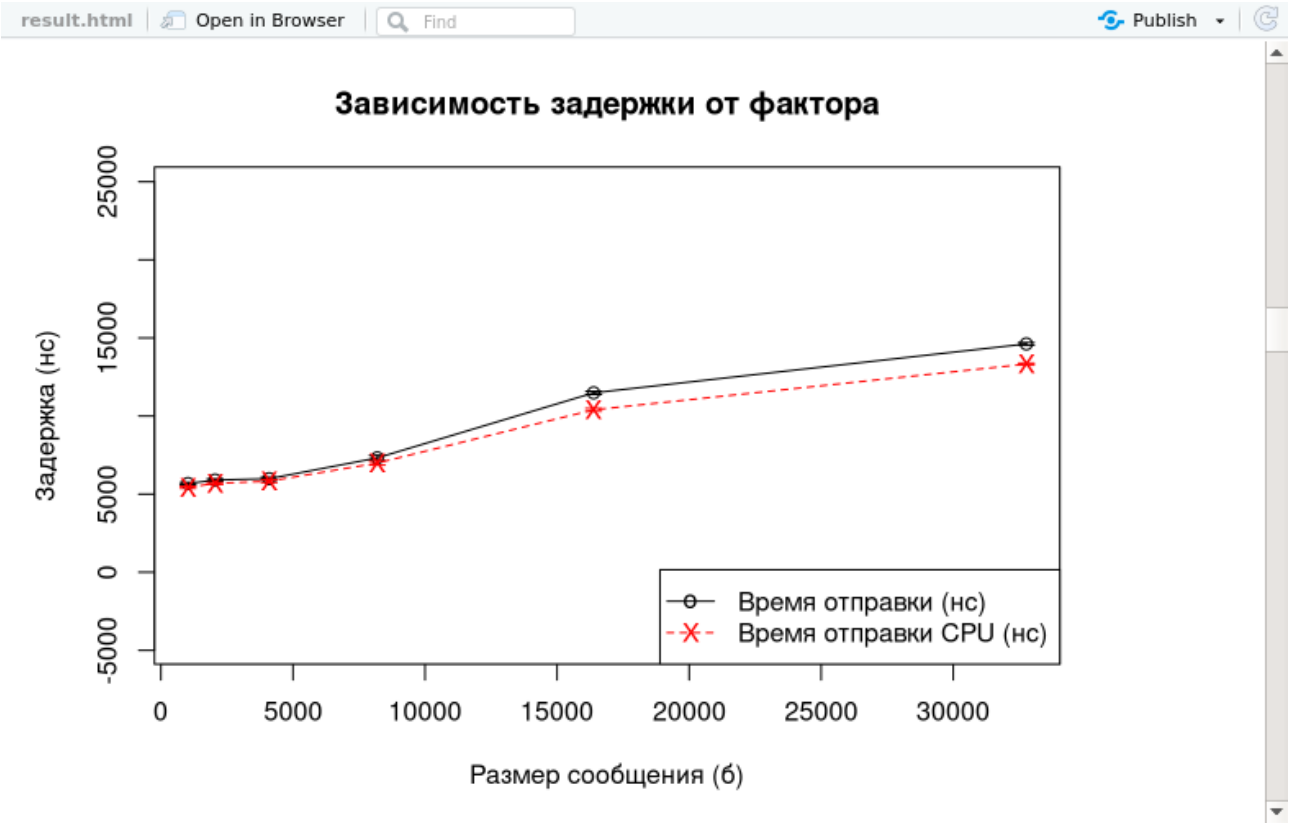

Рис. 2.4 – Пример графика зависимости времени задержки от размера сообщения

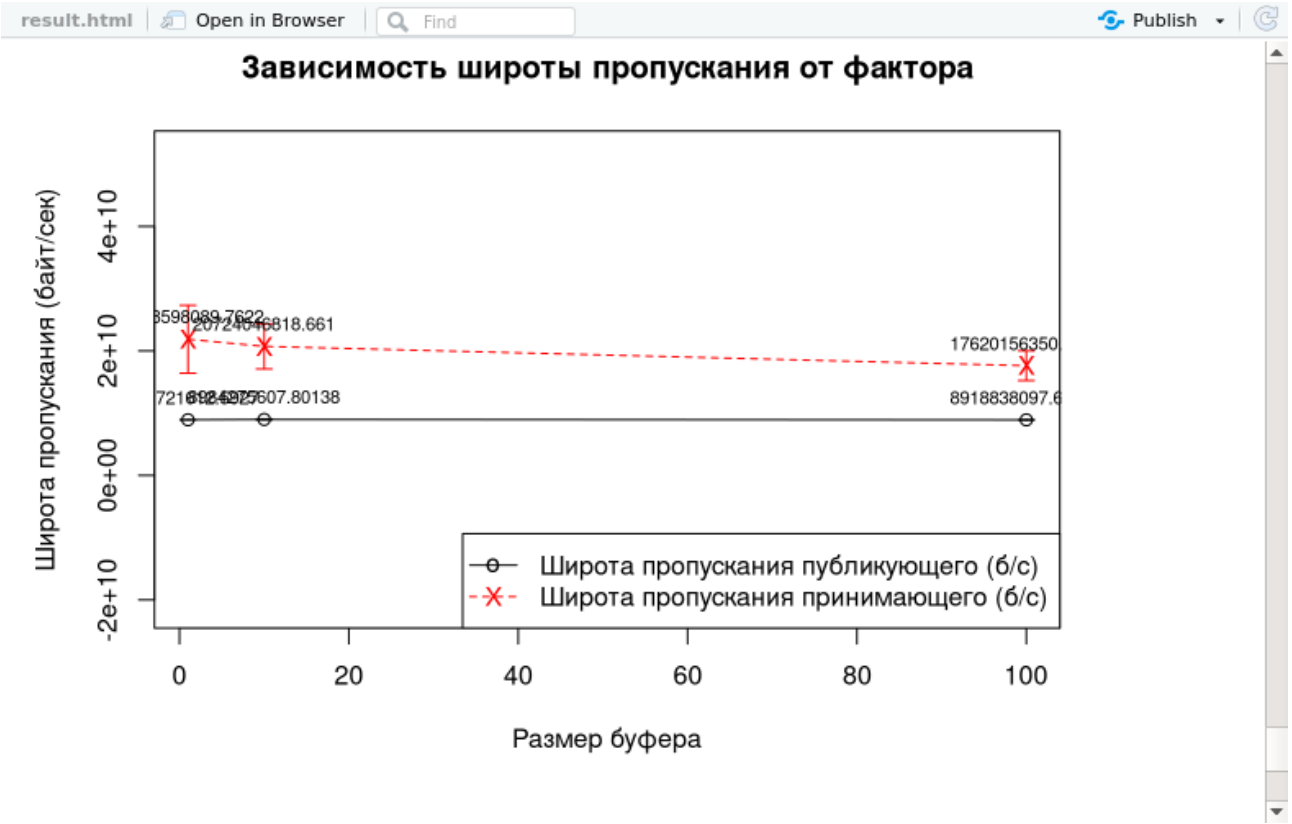

Рис. 2.5 – Пример графика зависимости широты пропускания от размера буфера

2.4 Выводы

Таким образом, были выявлены предмет тестирования и тестовые случаи, было создано тестовое окружение при помощи Docker-контейнеров, реализованы тестовые случаи при помощи описанного API Google Benchmark и было реализовано приложение, обрабатывающее входные результаты экспериментов в формате json и дающее на выходе rmarkdown файл с таблицами и графиками результатов экспериментов, который может быть преобразован в html страницу.

3 РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ

3.1 Характеристики тестируемого окружения

Тесты проводились на следующей аппаратной конфигурации:

- 8 процессоров Intel Xeon E5-2580 2.4 ГГц;
- 64 Гб оперативной памяти.

Операционная система: Ubuntu 16.04. Для тестирования был установлен Docker версии версия.

3.2 ROS

3.2.1 Зависимость задержки от определенных факторов

Разбирая вопрос факторов, влияющих на производительность прикладного ПО для системы ROS, предполагалась важность следующих факторов:

- размер буфера;
- количество подписчиков.

В ходе тестирования были получены данные, которые позволяют подтвердить или опровергнуть гипотезы о значимости данных факторов для производительности ROS приложений.

3.2.1.1 Зависимость от буфера

Для проверки гипотезы о том, что задержка передачи данных как-либо зависит от размера буфера топика, рассмотрим графики на рисунке 3.1, в которых отражены результаты тестирования задержки передачи данных в системе «издатель-подписчик» при единственном подписчике, но при разных объемах данных в зависимости от размера буфера топика. Зависимость примерно линейная и задержка практически никак не зависит от размера буфера топика.

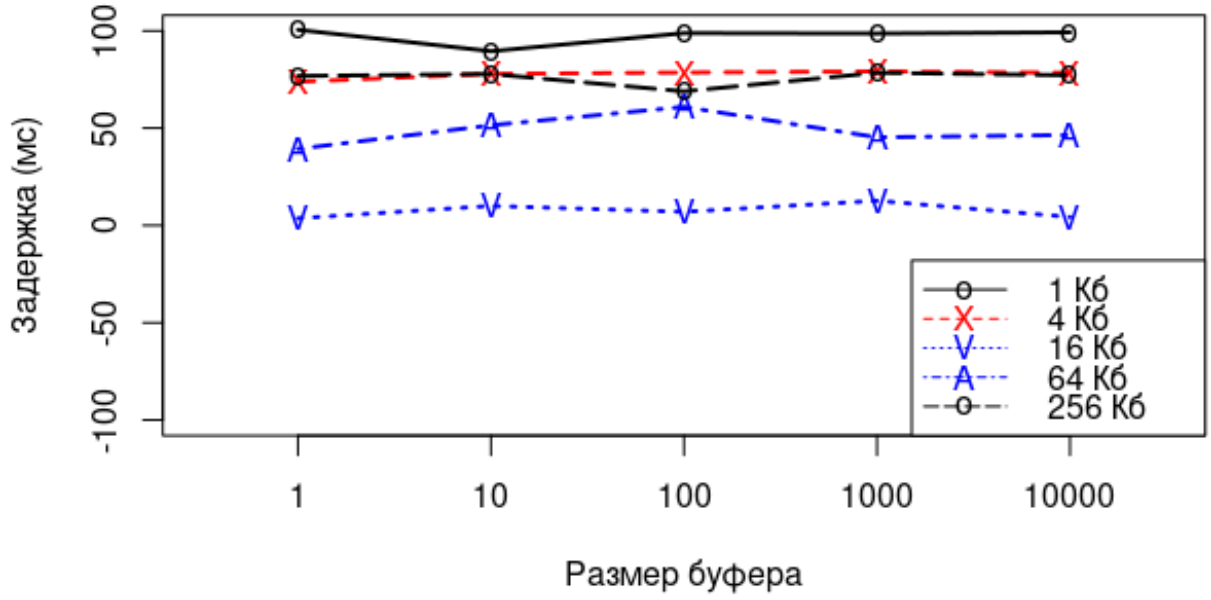

Рис. 3.1 – График зависимостей задержек передачи данных при разных объемах данных в пределах мегабайта от размера буфера топика.

3.2.1.2 Зависимость от количества подписчиков

Для проверки гипотезы о том, что задержка передачи данных как-либо зависит от количества узлов-подписчиков на топик, рассмотрим графики на рисунках 3.3 и 3.2, в которых отражены результаты тестирования задержки передачи данных в системе «издатель-подписчик» при буфере равном 1000, но при разных объемах данных в зависимости от количества подписчиков.

Из графиков видно, что большого влияния количество подписчиков не оказывает на быстродействие системы. Однако, следует заметить высокое, пропорциональное количеству подписчиков, использование памяти. Первоначально тестирование планировалось для 32 подписчиков, но 64 Гб доступной оперативной памяти не хватало даже на 16 подписчиков при объеме сообщений 64 Мб.

3.2.2 Показатели производительности при различных объемах данных

3.2.2.1 «Издатель-подписчик»

Разберем графики с зависимостью задержки (секунды) и пропускной способности (мегабайты в секунду) от различных объемов данных,

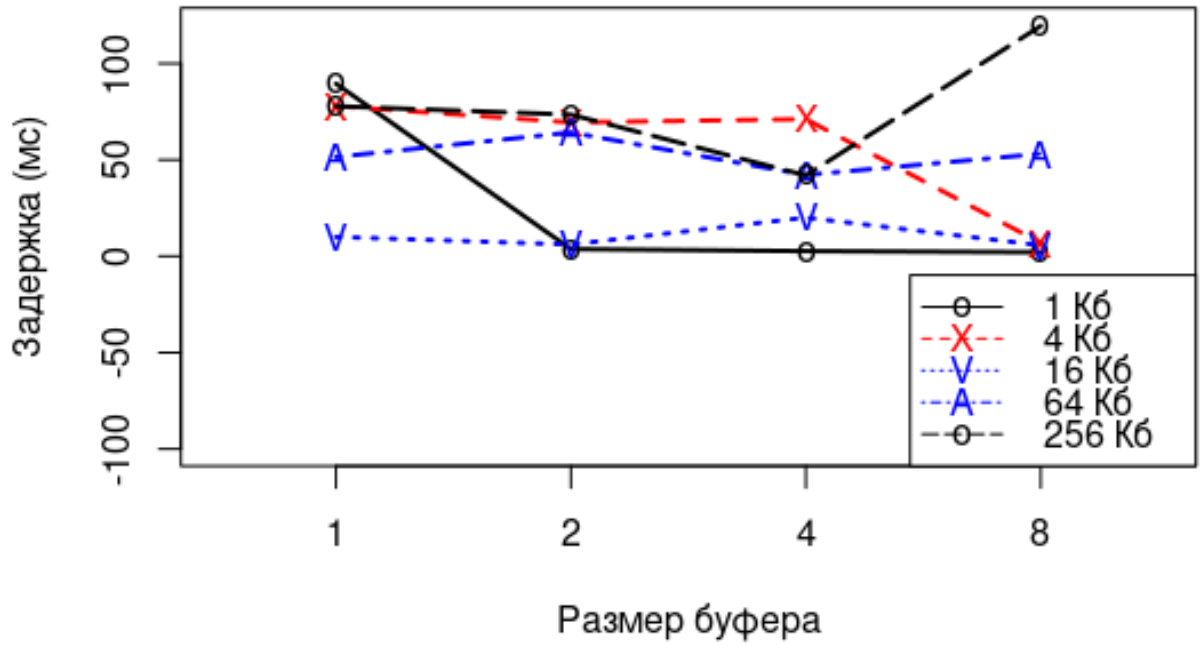

Рис. 3.2 – График зависимостей задержек передачи данных при разных объемах данных в пределах мегабайта от количества подписчиков.

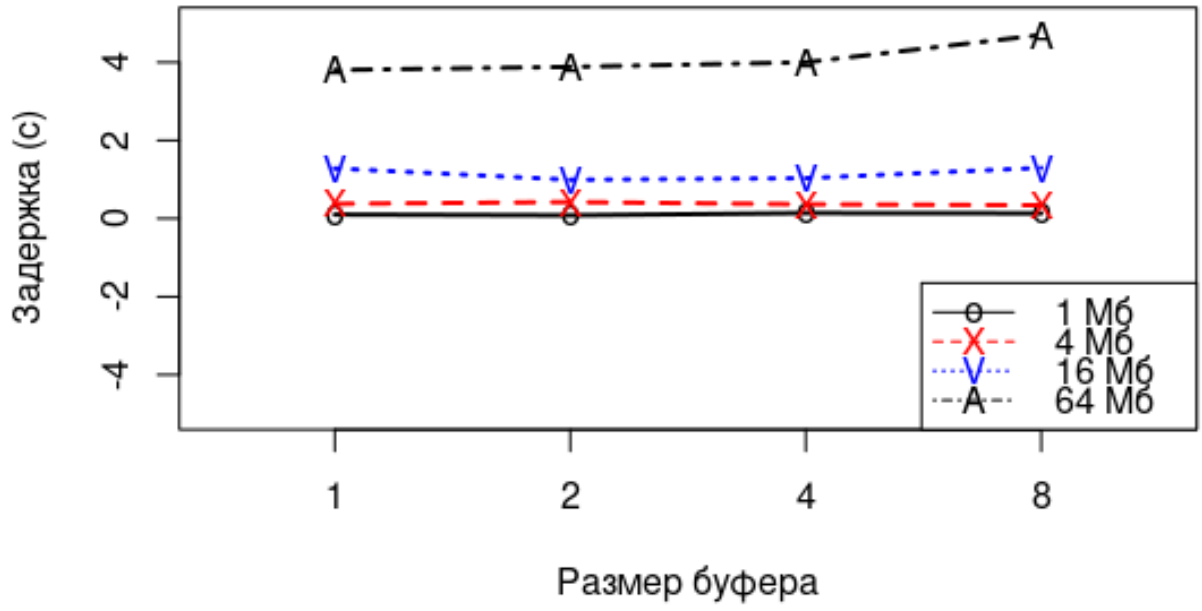

Рис. 3.3 – График зависимостей задержек передачи данных при разных объемах данных больше мегабайта от количества подписчиков.

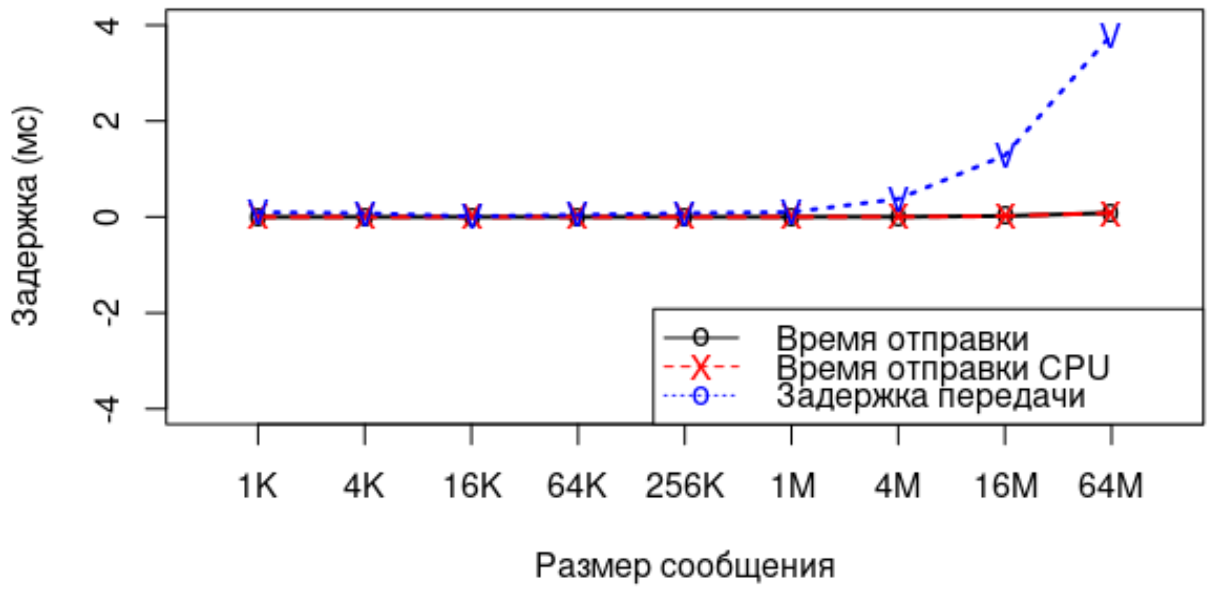

Рис. 3.4 – График зависимостей задержек передачи данных при разных объемах данных больше мегабайта от количества подписчиков.

передаваемых в системе. Для рассмотрения берется система из одного издателя, одного подписчика и буфера размера 1000.

На рисунке 3.4 видно, что при сообщениях от 16 Мб задержка составляет больше секунды. Это для робототехнической системы является неприемлимым с точки зрения предметной области. Следовательно, ROS не рекомендуется использовать для передачи больших объемов данных, либо учитывать большую задержку при обмене информацией.

На рисунке 3.5 видно, что при сообщениях больше 16 Кб реальная пропускная способность падает из-за роста задержки передачи сообщений. Кроме того, пропускная способность относительно узла-издателя так же падает при передаче сообщений больше мегабайта.

3.2.2.2 «Клиент-сервис»

Рассмотрим графики с зависимостью задержки в миллисекундах и пропускной способности в мегабайтах в секунду от различных объемов данных, передаваемых в системе. При рассмотрении клиент-сервисного подхода уникальны следующие детали:

- данные передаются в обе стороны: на запрос и ответ - следовательно объем

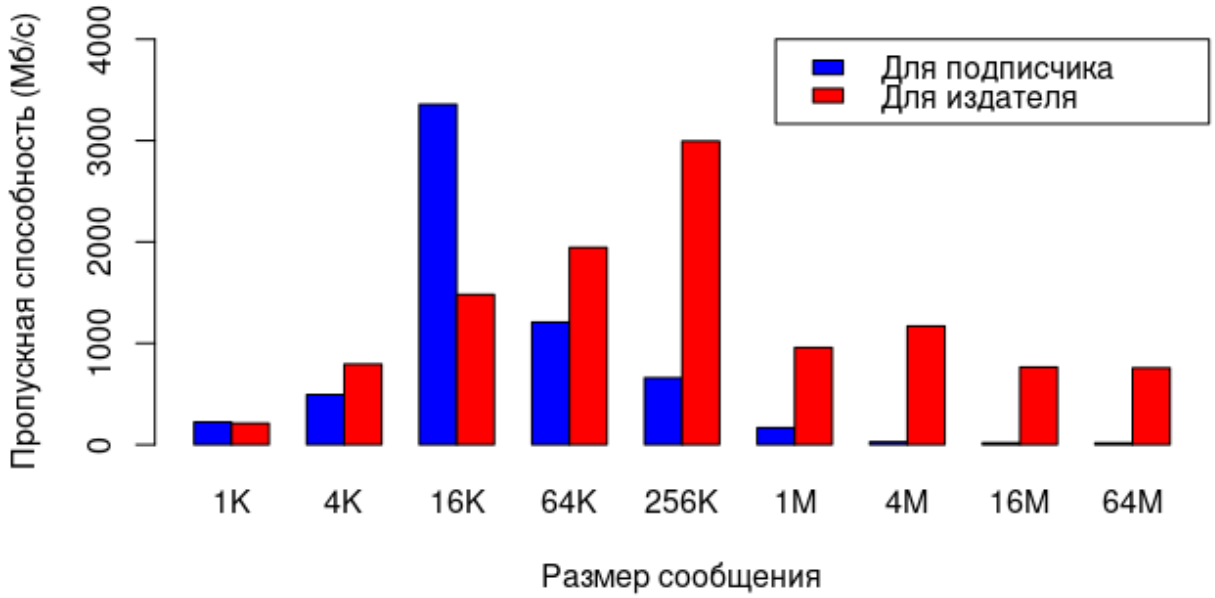

Рис. 3.5 – График зависимостей пропускной способности относительно публикующего и относительно принимающего от разного объемах данных.

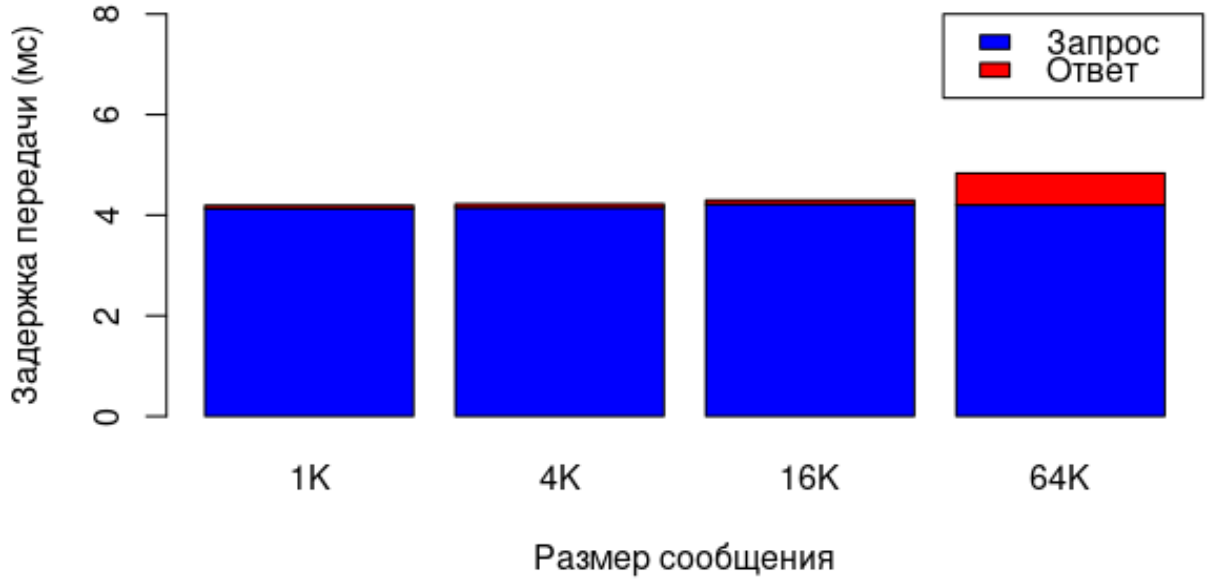

Рис. 3.6 – График зависимости времени запроса и ответа при объеме данных до 64 Кб.

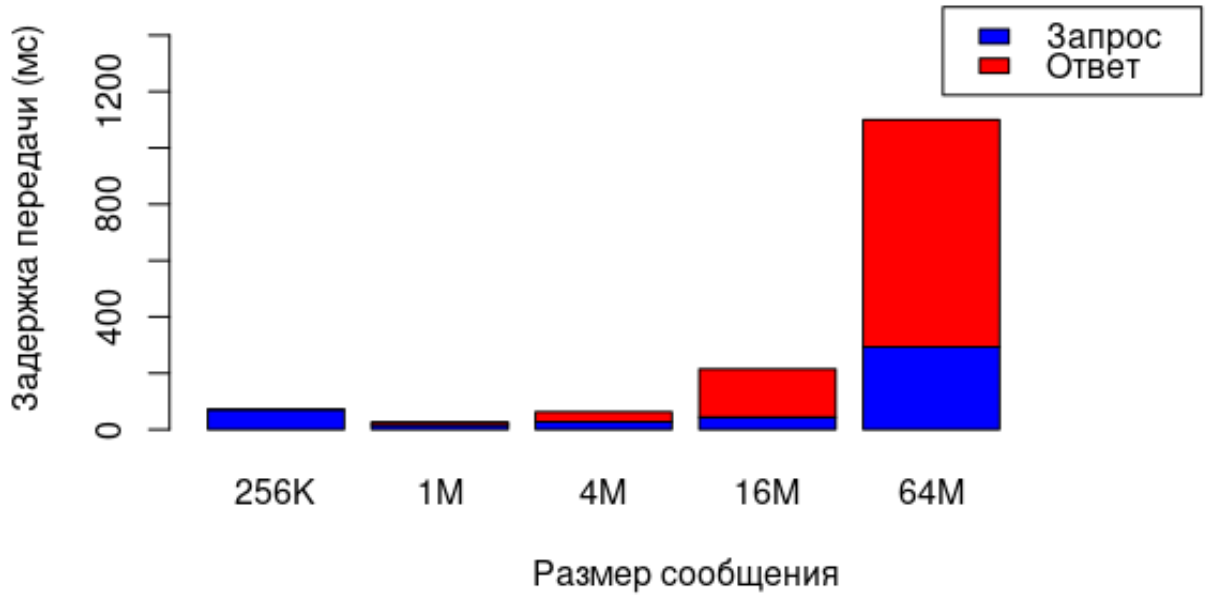

Рис. 3.7 – График зависимости времени запроса и ответа при объеме данных от 256 Кб.

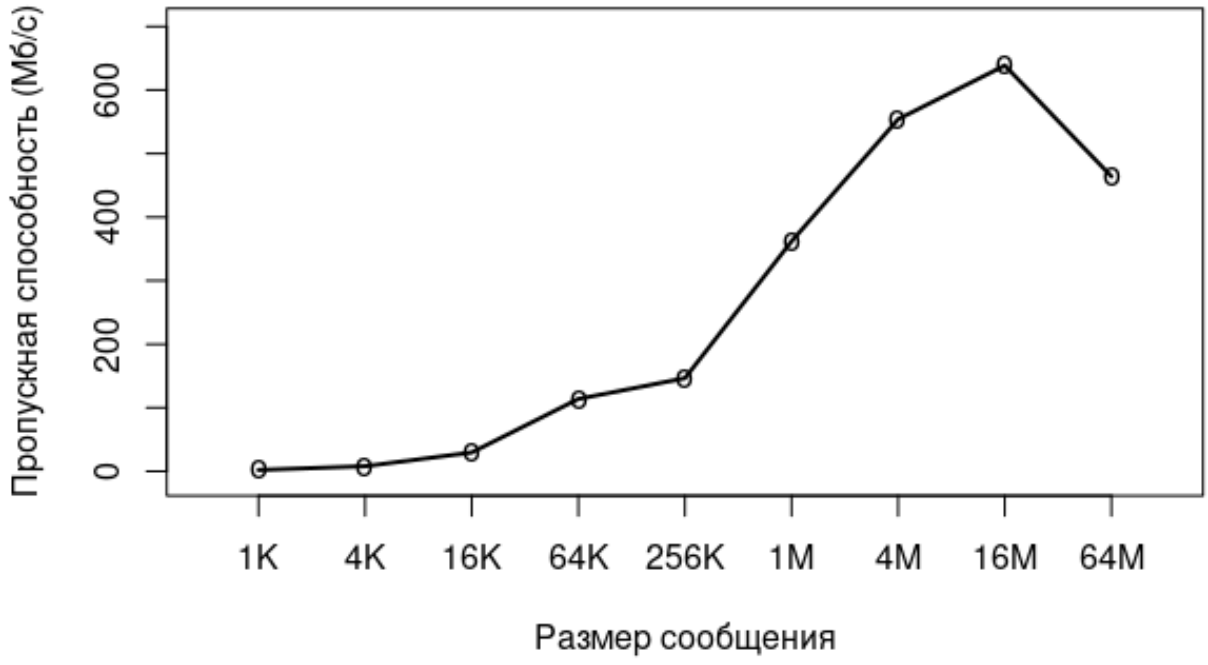

Рис. 3.8 – График зависимости пропускной способности от объема передаваемых даных

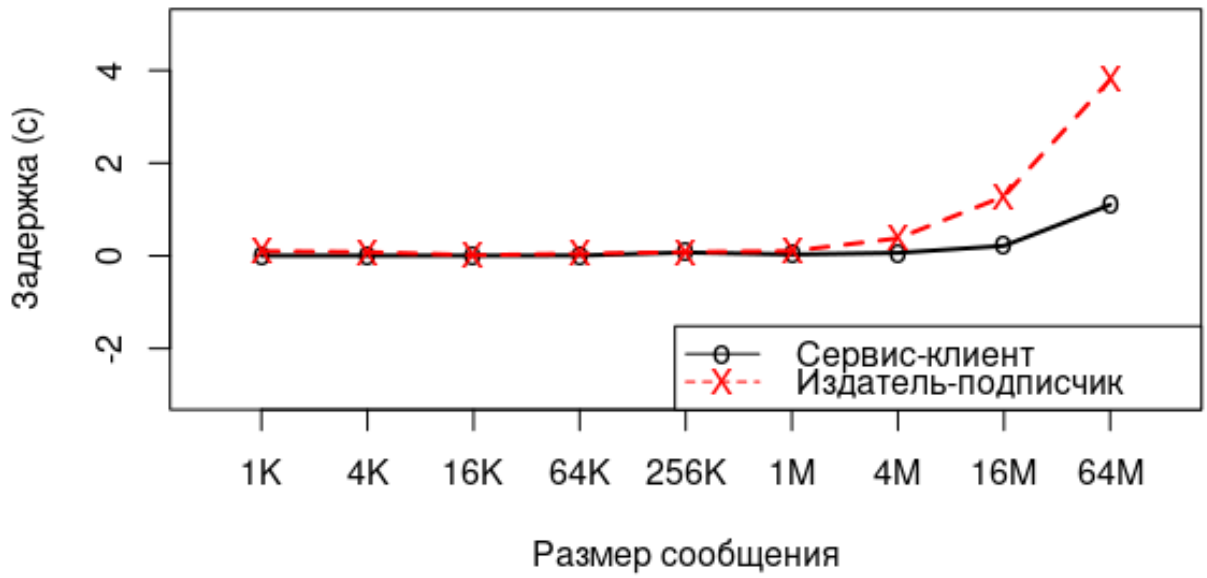

Рис. 3.9 – График зависимостей задержки передачи данных от объема передаваемых даных для разных типов коммуникации в ROS

передаваемых учитывается два раза в полосе пропускания;

- имеется возможность измерить время передачи данных в обе стороны.

На графиках 3.6 и 3.7 видно, что до 256 Кб задержка отправки ответа незначительна, но при передаче данных от 1 Мб роль задержки ответа резко возрастает.

На графике 3.8 видно, что при передаче данных посредством вызова удаленных процедур есть предел полосы пропускания около 650 Мб/с.

Кроме того, интересен график на рисунке 3.9, из которого следует, что передача данных при помощи вызова удаленных процедур, при том, что данных фактически при «сервис-клиент» подходе передавалось в 2 раза больше, задержка у подхода «сервис-клиент» примерно в 4 раза меньше.

3.3 YARP

3.4 MIRA

3.4.1 Различие производительности различных типов модулей

MIRA выделяется подходом к модулям - узлам распределенной системы. В MIRA модулями являются не исполняемые модули, а объекты классов, полученные при помощи реализации шаблона «Фабрика объектов» внутри самого фреймворка. В общем случае, каждый модуль - это объект класса с

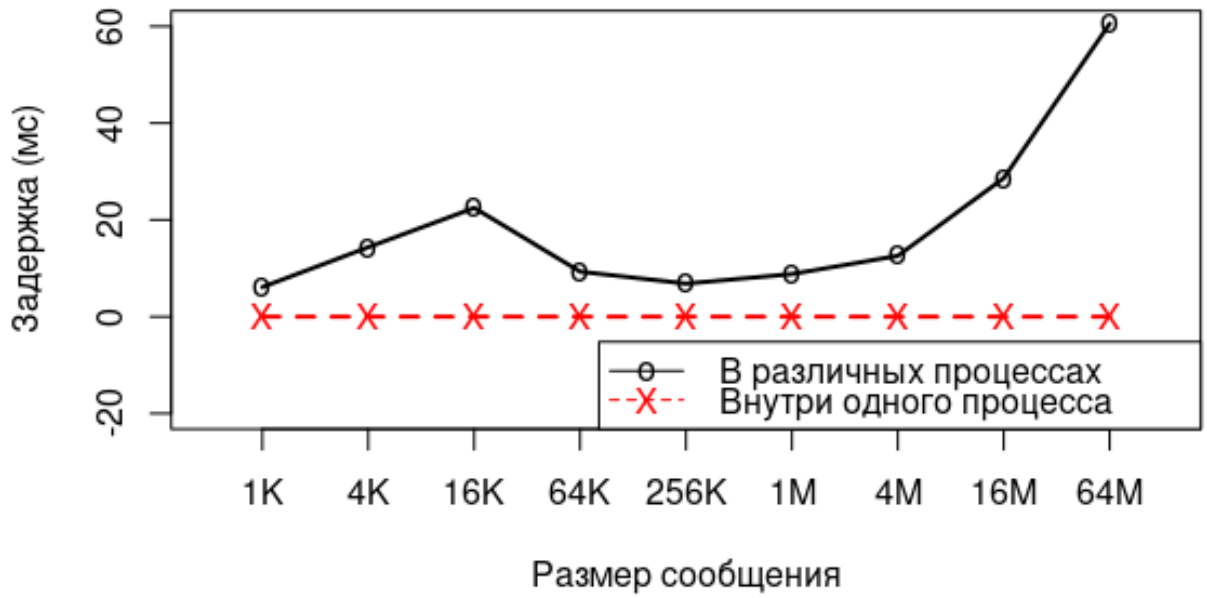

Рис. 3.10 – График зависимостей задержек передачи данных внутри одного процесса и между двумя процессами в зависимости от объема передаваемых данных на стороне принимающего.

определенным интерфейсом. Все модули компилируются как разделяемые библиотеки. Это позволяет реализовать наиболее быстрое межпроцессное взаимодействие: разделяемая память внутри процесса.

Кроме того, имеется возможность запускать модули в разных процессах, разработчики реализовывают взаимодействие в данном случае при помощи протокола TCP. Тем не менее, гарантия доставки сообщений отсутствует, т.к. сообщение может быть на момент прибытия в нужный узел «неактуальным» и будет пропущено. В случае, если модули находятся в разных процессах этот факт очень заметен: в 7 из 90 случаев сообщения в ходе тестирования не были зафиксированы системой тестирования. Этим можно объяснить колебания задержки, измеряемой на получателе. Кроме того, получатель не может формально отличить некорректные сообщения первых итераций тестирования, таким образом результаты, измеряемые на узле-получателе так же имеют дополнительную ошибку в виде первых нескольких измерений для каждого теста.

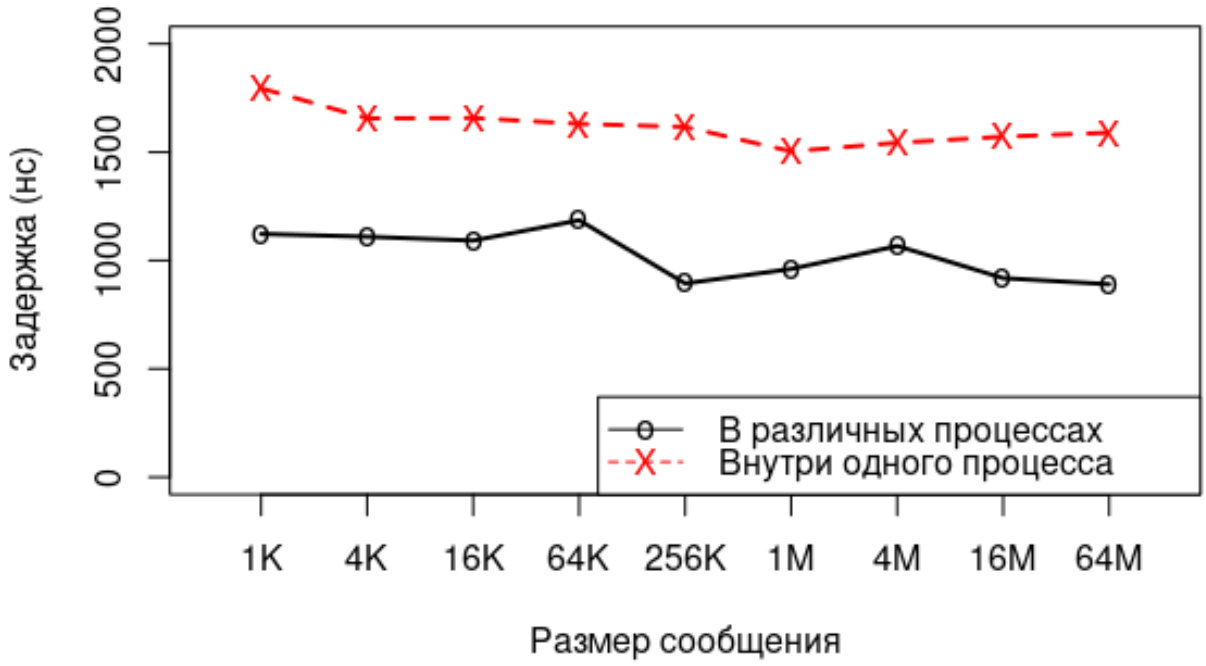

Рис. 3.11 – График зависимостей задержек передачи данных внутри одного процесса и между двумя процессами в зависимости от объема передаваемых данных на стороне посылающего.

3.4.1.1 Задержка передачи данных

На рисунке 3.10 показаны графики задержки передачи сообщений. На них четко видно, что реальная задержка передачи сообщений между модулями внутри одного процесса на несколько порядков (порядок 1000 наносекунд внутри одного процесса и 10 миллисекунд в разных) ниже, чем в различных процессах.

При этом, на рисунке 3.11 отображен график, из которого следует, что время непосредственно публикации данных на модуле-передатчике внутри одного процесса примерно в 1.5 раза дольше, чем в разных процессах.

3.4.1.2 Пропускная способность

Пропускную способность стоит рассматривать с учетом времени, которое занимает передача сообщения от передатчика к получателю. Если рассматривать время только на передатчике, то результат будет не соответствовать реальности.

Как видно из графика на рисунке 3.12, пропускная способность передачи

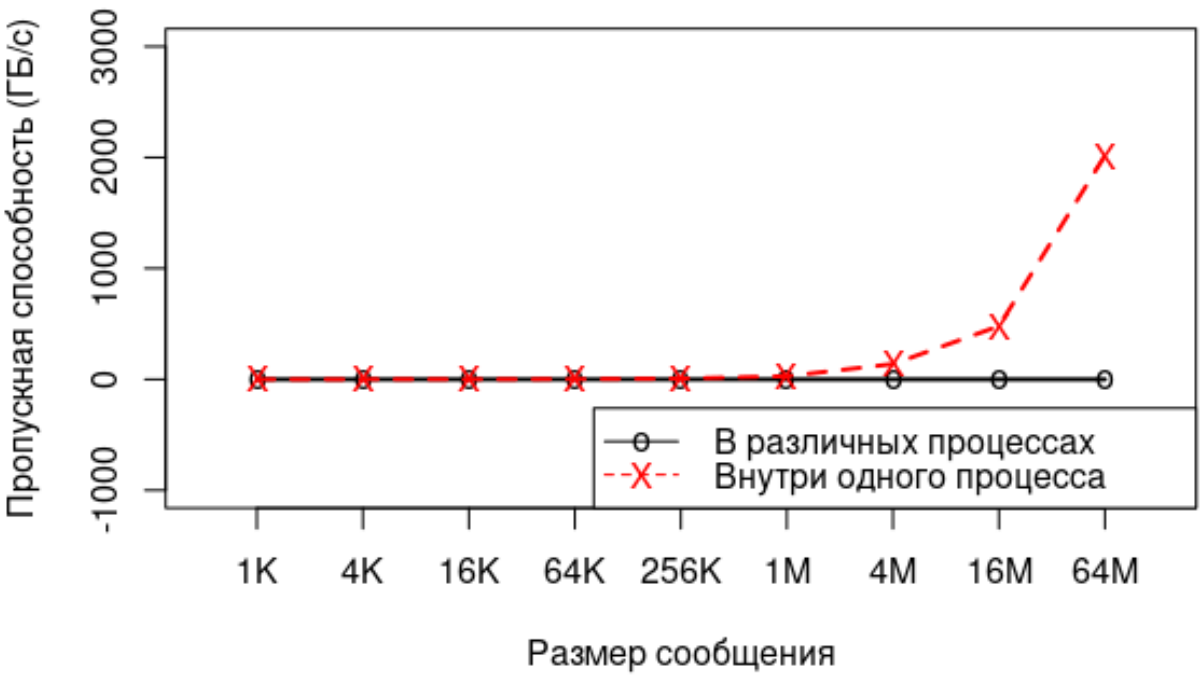

Рис. 3.12 – График зависимостей пропускной способности передачи данных внутри одного процесса и между двумя процессами в зависимости от объема передаваемых данных.

данных внутри процесса составляет гигабайты в секунду. Фактически, в случае внутрипроцессного взаимодействия, пропускная способность ограничена лишь доступом к оперативной памяти. В случае же межпроцессной передачи данных, пропускная способность достаточно низкая, около десятков мегабайт в секунду.

3.4.2 Производительность различных подходов коммуникации

В MIRA, кроме шаблона «издатель-подписчик», доступно взаимодействие по шаблону «сервис-клиент». Сравнение будет вестись для модулей в разных процессах, поскольку чаще всего стоит задача удаленного вызова процедур у другого процесса.

График на рисунке 3.13 показывает, что при удаленном вызове процедур с увеличением объема данных резко начинает возрастать время на передачу данных. Стоит отметить, что в тестах объем данных передавался в обе стороны: от клиента к сервису и обратно. Тем не менее, это не объясняет стремительный рост времени передачи данных при больших объемах данных.

График на рисунке 3.14 показывает, что при удаленном вызове процедур

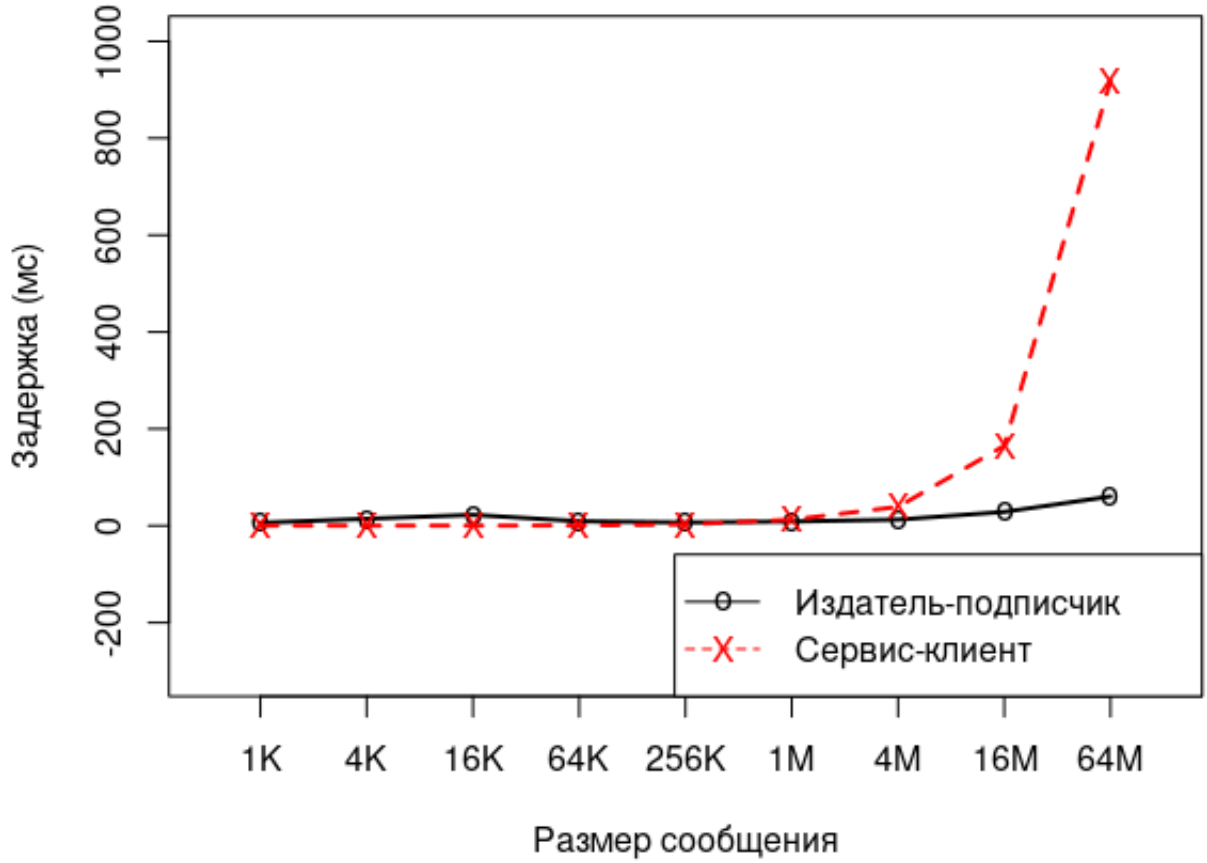

Рис. 3.13 – График зависимостей задержки передачи данных для различных подходов коммуникации в зависимости от объема передаваемых данных.

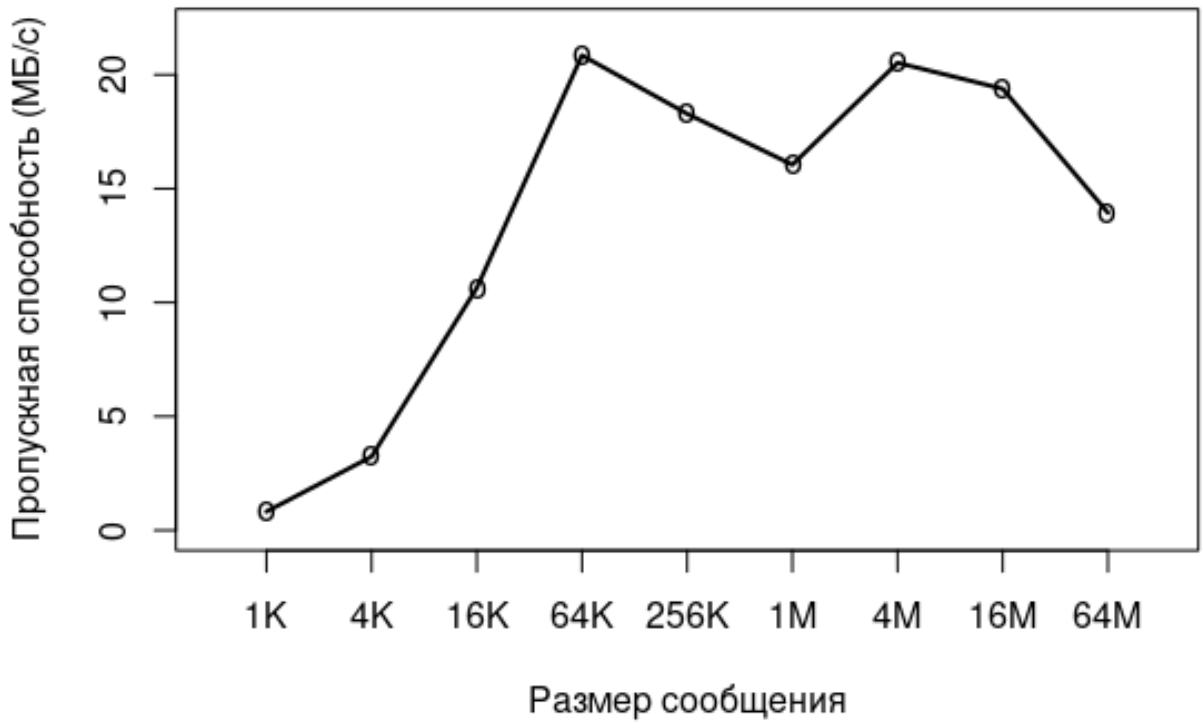

Рис. 3.14 – График зависимости пропускной способности передачи данных при удаленном вызове процедуры от объема передаваемых данных.

виден порог пропускной способности: примерно 22 мегабайта в секунду.

3.5 OROCOS

3.6 Сравнение результатов

3.7 Выводы

4 КОММЕРЦИАЛИЗАЦИЯ РЕЗУЛЬТАТОВ НИР

4.1 Резюме

В данном документе рассмотрен бизнес-план по разработке и поддержке анализатора результатов тестирования производительности на основе Google Benchmark для проектов на C/C++.

Продукт ориентирован на C/C++-программистов, в большей степени на тех из них, кто занимается анализом и тестированием ПО.

Маркетинговая стратегия основывается на следующих идеях:

- Оплата идет по месячной подписке.
- Рекламную функцию выполняет ограниченная в функционале бесплатная версия с открытым кодом, а так же новости о продукте на ресурсах общения целевой аудитории.
- Вытеснение конкурентов будет происходить узкой направленностью программы и низкой ценой, основанной на большом объеме продаж.

Команда разработчиков состоит всего из одного программиста. Были минимизированы косвенные затраты, используя бесплатные площадки для размещения и распространения продукта, практически не жертвуя при этом качеством. Это обуславливается небольшим размером и низкой сложностью самого продукта и способов его продвижения в современное время.

Для начала кампании требуются небольшие инвестиции, которые разработчик способен покрыть самостоятельно. Эти инвестиции окупаются спустя уже в первом квартале от начала кампании по наиболее реалистичному сценарию и приносят стабильный доход. Чистая текущая стоимость проекта в конце года гораздо больше 0 и составляет 713224.2435 руб. Внутренняя норма доходности по итогам года составляет 504%. Дисконтный индекс рентабельности равен 20.3778.

Основные риски приходятся на колебание объема продаж.

4.2 Описание продукции

4.2.1 Наименование продукта

Продуктом является программное средство для реализации и анализа результатов тестов производительности. Определение «программа» и «программный модуль» регулируется ГОСТ 19781-90 и формулируется следующим образом:

Программа - данные, предназначенные для управления конкретными компонентами системы обработки информации в целях реализации определенного алгоритма.

Программный модуль - программа или функционально завершенный фрагмент программы, предназначенный для хранения, трансляции, объединения с другими программными модулями и загрузки в оперативную память.

4.2.2 Назначение продукта

Продукт разработан для разработчиков ПО, которым требуется:

- выполнить сравнение производительности разработанного и конкурирующих решений;
- определить время исполнения критически важных модулей разработанного ПО,
- установить предельную нагрузку на разработанную систему, для доказательства удовлетворения нефункциональных потребностей заказчика.

Разработанный программный продукт предоставляет библиотеку Google Benchmark с API для реализации тестов производительности, а так же программу-анализатор, обрабатывающую результаты тестирования. Анализатор имеет следующие особенности:

- простота реализации: любой разработчик может изменить обработку данных под требования конкретной задачи;

- анализ формата json, как наиболее простого для понимания и обработки формата данных;
- выдача результата в виде текстового файла rmarkdown, что позволяет сохранить отчет в удобный для систем контроля версий текстовый файл и, при надобности, преобразовать отчет в любой из форматов: html, pdf, LaTeX;
- генерация графиков по всем факторам анализа производительности;
- конфигурируемость входных данных, возможность выбирать выборки входных данных и факторы для анализа результатов;
- конфигурируемость отчета, возможность выбирать информацию, отображаемую на графиках;
- конфигурация выполняется путем редактирования простых для понимания xml файлов.

4.2.2.1 Основные характеристики продукции

Оценка и характеристика качества ПО регулируется ГОСТ Р ИСО/МЭК 9126-93.

Функциональные возможности. В данный момент в анализаторе реализованы все заявленные выше, в подразделе 4.2.2, функции.

Надежность. На данный момент приложение тестировалось на результатах измерения производительности МАРППО. При соблюдении требований к формату исходных данных, результат работы разработанной программы остается стабильным. тем не менее, анализатор не устойчив к ошибкам пользователя: ошибкам в формате результатов тестов или конфигурационных файлах.

Практичность. Разработанное решение является практичным засчет высокой степени автоматизации процессов анализа и получения наглядного результата. Для выполнения анализа результатов тестирования производительности требуется заполнить два xml файла для передачи информации об измеряемых факторах, об анализируемых тестах, о

требующихся для построения графиках.

Эффективность. Анализатор не является самым эффективным (под эффективностью понимается скорость работы и ресурсоемкость ПО) решением, но при этом занимает удобную нишу на рынке: прямых конкурентов у данной программы нет, так как в большинстве случаев для тестов столь низкого уровня анализируют результаты вручную, либо при помощи разработанных внутри предприятия инструментов. Для анализатора результатов важна точность результатов и удобные интерфейсы для разработки тестов. По сравнению с выполнением самих тестов производительности, время работы анализатора не значительно.

Сопровождаемость. Исходный код подробно документирован. Так же, реализация на языке Python позволяет проще писать самодокументиуемый код, т.е. код, который понятен в большинстве случаев без излишних комментариев со стороны разработчика. Так же была написана инструкция по использованию данного программного решения.

Мобильность. Анализатор разработан на языке Python для интерпретатора Python3.7 и данную работу можно запустить на любой ОС с наличием соответствующего интерпретатора.

4.2.3 Потребительские свойства

В ходе опроса разработчиков распределенных систем было установлено, что в ходе тестирования производительности для анализа результатов тестирования производительности в большинстве случаев используется занесение данных в Excel таблицы и построение результатов в соответствующем офисном пакете. У данного подхода имеется ряд недостатков, так как каждый раз для анализа либо придется переделывать старые таблицы, либо создавать новые. В данной работе предоставляется подход к тестированию производительности от составления плана тестирования до получения конечного результата в виде таблиц и графиков.

Анализатор прост в использовании, для выполнения анализа результатов

требуется подготовка двух конфигурационных файлов. Далее всю работу выполняет реализованная программа.

Кроме того, автоматизация работы и параллельность выполнения анализа позволяет свободным от этой рутины разработчикам выполнять свои должностные обязанности, например, работать над тестированием нескольких разработок одновременно.

4.2.4 Основные конкурентные преимущества продукции

Способ распространения. Распространение работы происходит через интернет, имея возможность загрузить исходный код, документацию, обоснование работы, описание, а так же примеры реализации тестирования производительности для МАРППО.

Дальнейшее сопросвождение ПО и поддержка клиентов. Инструменты анализа производительности будут улучшаться в дальнейшем: был получен ряд отзывов и предложений о желаемом функционале, который планируется добавить в последующей разработке. Как пример: увеличение гибкости и настраиваемости графиков, возможность сравнения различных тестов по отдельным факторам.

Кроме того, предоставляется поддержка клиентов при возникновении трудностей при работе с разработкой или при обнаружении ошибок в реализации.

Простота использования. Для использования анализатора на практике в общем случае не требуется знать специальных знаний из области программирования. Тем не менее, если потребуется внести коррективы в алгоритм обработки данных, то разработчик это может сделать не затратив много времени из-за простоты языка программирования и открытого исходного кода.

Предоставление методики проведения тестирования производительности.

К исходному коду программы предоставляется описание и примеры реализации тестов производительности МАРППО. Разработчики,

выбравшие предоставленное в данной работе решение могут использовать испробованную на практике методику тестирования различных распределенных систем, учитывая решения возникавших проблем.

Использование промежуточных файлов вычислений в формате json

Анализатор сохраняет промежуточные результаты различных этапов анализа в отдельных файлах в формате json, простом для обработки как с программной точки зрения, так и с точки зрения удобства чтения для человека. Промежуточные вычисления можно использовать для расширения возможностей анализатора и уточнения результатов тестов.

4.2.5 Основные потребители и направления использования продукции

Анализатор будет востребован для разработчиков и тестировщиков ПО, которым требуется проводить анализ разрабатываемых программных решений с точки зрения улучшения качества ПО относительно собственной разработки, так и относительно возможных конкурентов. На основе сравнительных данных разработчики будут иметь возможность предоставлять конструктивные аргументы технического характера для обоснования принятия решений при согласовании ПО с заказчиком, при разработке и анализе качества ПО, а так же может использоваться в качетсве одного из инструментов маркетинга, как демонстрация преимущества перед конкурентами.

Основными потребителями являются разработчики ПО на языке C++, специализирующиеся на тестировании и анализе качества ПО.

4.2.6 Структура выпуска продукции

Плагин будет выпускаться в двух версиях:

1. Бесплатная версия с базовым функционалом с открытым исходным кодом под лицензией BSD. Эта версия нужна по причинам:
   - Открытый исходный код позволит снизить нагрузку на поддержку, т.к. заинтересованные в новом функционале разработчики смогут дописать

его сами. Кроме того, эти наработки можно использовать в дальнейшем для улучшения функционала платной версии продукта.

- Бесплатная версия часто воспринимается как демо-версия продукта и является частью стратегии маркетинга.

2. Платная версия, распространяемая по подписке с полным функционалом, с документацией и примерами полного цикла выполнения тестирования производительности, поддержкой пользователей и дальнейшее сопровождение анализатора будет продолжаться только для платной версии.

4.3 Анализ рынка и сбыта продукции

4.3.1 Потребители продукции

Разработка программного обеспечения – одна из ведущих отраслей рынка развивающихся и развитых стран. Существует множество методологий, стандартов, принципов, которые позволяют систематизировать и поставить на конвейер производство программных продуктов. Один из важных этапов разработки ПО, без которого практически не обойтись при разработке крупной автоматизированной системы – тестирование. На этом этапе важно правильно выделить критические для производительности всей системы элементы, иметь возможность в кратчайшие сроки реализовать тестовые случаи и получить понятный и удобный для анализа результат.

Таким образом, круг заинтересованных в данном решении: программисты, использующие язык C++ как основной в своих проектах. Кроме того, т.к. анализатор в первую очередь используется для автоматизации рутины – составление отчетов с таблицами и визуализацией данных, удобство работы с данными, анализ и сравнение производительности ПО – задачи разработчиков, ответственных за качество ПО, то именно для этого, более узкого круга специалистов предназначенно разработанное приложение.

Стоит отметить, что потребители данной продукции – разработчики в секторе крупных разработчиков ПО, которые уделяют большое внимание

контролю качества выпускаемого ПО. Именно на таких потребителей будет ориентирован платный вариант продукта. Сектор открытого программного обеспечения так же является заинтересованным лицом, т.к. может использовать и разрабатывать в своих интересах бесплатную версию. Оценить бесплатный сектор во-первых сложно (он черезвычайно хаотичен и по нему нету статистики как таковой), во-вторых ожидаемая польза от этого сектора в виде сторонних открытых разработок, которые мы сможем применить - это скорее возможный приятный бонус, нежели сколько-либо гарантируемая прибыль. Наиболее важная цель бесплатной версии с открытым кодом: маркетинг. Таким образом, сегмент открытого ПО не приносит нам прямой прибыли, но уменьшает расходы на рекламу.

4.3.2 Основной сегмент рынка

Основным сегментом рынка, как следует из предыдущего раздела, является сектор крупных разработчиков ПО, основными модулями которого являются разработки на языке C/C++, проекты данной отрасли будут состоять из высоконагруженных и больших проектов, требующих внимательного подходу к качеству, производительности и устойчивости.

4.3.3 Товарные особенности сегмента и позиционирование продукта

Сегмент, работающий с подобными большими проектами хочет получить быстрый результат, минуя как можно больше рутинных задач. Именно так и будет позиционироваться плагин: решение конкретной задачи, вместо с одной стороны написания небольших временных внутрикорпоративных решений, и дорогих программных средств с перегруженным функционалом с другой. Предлагаемое решение за меньшую стоимость, дающий стандартизированный и понятный для подавляющего большинства программистов результат, удобный в использовании, а так же решение, запуск которого возможен на всех распространенных платформах.

4.3.4 Каков спрос на продукт; каков потенциал рынка в целом и по секторам

Со стабильным трендом на C/C++ как язык разработки по данным рейтинга tiobe и растущим спросом на программистов, знающих эту технологию, за 4 года точно был большой рост потенциальной аудитории данного анализатора. Более того, абсолютно все аналитические сайты указывают на рост потребности в C/C++ разработчиках и дальше. По информации с того же рейтинга Tiobe, интерес к C/C++ по отдельности с 2000 года конкурировал лишь с языком Java. На данный момент C/C++ суммарно занимают 21.668% от всех остальных технологий.

По данным Google Trends запрос «C++ benchmark» имеет стабильно высокий спрос с 2014 года. В целом, способ маркетинга ПО, основанном на сравнительной статистике с производительностью конкурентов является выигрышным и распространенным. Формально, для этой цели может подойти любой продукт, если у разработчиков будет серьезное желание коммерциализировать проект.

Для оценки потребительского спроса, можно использовать статистику по проектам на крупнейшем сайте разработчиков ПО с открытым исходным кодом: github.com. Согласно статистике, количество проектов, написанных преимущественно на C/C++ составляет 1.42 миллиона. Каждый из этих проектов - потенциальный клиент.

Из-за сконцентрированности на поставленной задаче, особенностях маркетинга, наше решение будет дешевле и выгоднее для обоих сегментов: и открытого ПО, которые для своих проектов будут более чем способны заплатить за инструмент анализа, так и для больших компаний, которые будут ценить время и затраты человеческих ресурсов в разы дороже, чем стоимость анализатора. Для сравнения, средняя з/п в месяц системного архитектора по Москве за апрель - от 100 тыс. руб., т.е. 625 руб в час. В среднем, чтобы подробно разобраться с большим проектом, уходит 10 рабочих дней, что эквивалентно 80 человеко-часам. Следовательно, за только разбор

большого проекта, не говоря о создании подробного отчета, работодатель потратит в среднем около 50 тыс. руб. В аутсорсинговой компании, с большим количеством «чужих» проектов и быстро текущих проектов, эта затрата будет отнюдь не единовременная.

Т.е. образом реализованный анализатор сэкономит затраты анализ результатов тестирований и подготовку отчетной информации. Поддержка проекта должна помочь потеснить с рынка менее качественных бесплатных конкурентов, а так же дорогих решений, предоставляющих слишком большой и, зачастую, ненужный функционал за большую плату.

4.4 Анализ конкурентов

В разделе 1.2.2.2 приведено сравнение по основным решениям для тестирования производительности. Данная работа является выигрышной поскольку предоставляет сам анализатор результатов, который не предоставляется ни одной из рассмотренных работ.

4.5 План маркетинга

4.5.1 План продаж

Как показывают тренды последних лет, то ПО все чаще не продается «как есть», а предоставляется по платной подписке за гораздо меньшую по сравнению с себестоимостью всего продукта цену.

Поскольку сам проект небольшой, то идея распространения по очень дешевой платной подписке с большим объемом продаж, взамен на которую пользователи получают постоянные обновления и поддержку по продукту - потенциально прибыльный план продажи нашего продукта.

Вычислим безубыточную цену подписки в месяц: за месяц на разработку тратится 58328 руб (из раздела 4.6.4), а минимальный объем ожидаемых клиентов по истечению третьего месяца 2000 пользователей (вывод раздела 4.3.4). Следовательно безубыточная цена, представленная покупателям в

начале второго месяца выхода продукта на рынок:

$$\frac{58328}{2000} = 29.17.$$

С учетом, что равных по функционалу конкурентов на рынке нет, то цена в $1 на иностранном рынке и 60 руб. на отечественном в 2 раза превышает себестоимость при этом являясь очень низкой в сравнении с затратами на ручной анализ результатов тестирования.

В таблице 4.1 представлен предполагаемый объем продаж на 2018 год.

Показатели	Квартал				Всего
	II	III	IV	IV	
Ожидаемый объем продаж	2000	2500	3000	3500	11000
Цена с НДС	180	180	180	180	720
Нетто выручка (без НДС)	360000	450000	540000	630000	1980000
Выручка с НДС (18%)	424800	531000	637200	743400	2336400
Сумма НДС	64800	81000	97200	113400	356400

Таблица 4.1 – План продаж на 2018 год

4.5.2 Стратегия маркетинга

Таким образом, ставка делается на качественный и узконаправленный продукт с длительным жизненным циклом. Ставка в ценообразовании делается на большой объем сбыта с низкой ценой, а так же экономии на затратах.

В частности, распространение будет производиться через бесплатный хостинг OpenShift от RedHat. Для нашего не такого уж и большого проекта вычислительных мощностей более чем достаточно: исходя из цифр раздела 4.3.4 база данных размером в 400,000 успешно размещается в этом сервисе, не требуя за это плату. Проблема с доменным именем решается путем

переадресации со страницы OpenSourse версии разработки на GitHub. В итоге получатель будет иметь доступ к понятному доменному имени cppbenchmark.github.io, в которой уже удобный процесс получения платной подписки будет перенаправлен на веб-приложение, размещенное на openshift.com на бесплатной основе.

Говоря о рекламе, поскольку у нас имеется версия с открытым исходным кодом, то она будет рекламироваться на целевых для программистов площадках: linux.org.ru, habrahabr.ru, opennet.ru, reddit.com/unix и т.д. Написание рекламных тем не требует никакой оплаты, лишь затрат времени, учтенного в таблице 4.2, как последний день подготовки к релизу. Остальное сообщество сделает самостоятельно. Т.о., предоставив открытый код мы расширили круг заинтересованных лиц и получили сильную рекламу своего продукта.

4.6 План производства

Производство ПО имеет достаточно большие и выигрышные особенности, по сравнению с производством большинства материальных благ и услуг. В частности:

- Почти все затраты в производстве ПО - на зарплату разработчиков. В данном случае не требуется даже оплачивать аренду, т.к. разработчик один и способен вести разработку по месту проживания. В итоге косвенных затрат почти нет, а постоянные состоят из выплат зарплат разработчикам.

- Процесс разработки ПО достаточно четко разбивается на этапы, т.к. уже есть множество наработок в этом направлении: как правильно устроить рабочий процесс.

- В нынешнее время цифрового распространения ПО так же избегаются затраты на транспортировку продукта до конечного пользователя.

4.6.1 Описание производственного процесса

Был выбран жизненный цикл ПО RAD (Rapid Application Development). В настоящий момент была написана основа решения, а так же одностраничный сайт, для продажи продукта. Все это было сделано за 14 дней, т.е. 112 рабочих часов. На данный момент разработка находится на этапе тестирования, после чего будет повторяться цикл из итераций с этапами, описанными в таблице 4.2.

4.6.2 Оплата труда

Разработчиком является один человек, час работы каждого составляет 190 р. Итого, учитывая уже затарченные 112 часов, стоимость продукта равна 21,280 руб.

За каждую итерацию в ЖЦ ПО по таблице 4.2 на оплату труда будет уходить не меньше 53200 руб.

В месяц на оплату труда будет уходить 42560 руб.

Отчисления на соц.нужды (в месяц):

$$42560 \cdot 0.3 = 12768 \text{ руб.}$$

4.6.3 Амортизационные отчисления

Каждый из разработчиков имеет ноутбук Dell Inspirion n7110, покупавшийся за 36,000 руб. с гарантией 2 года. Ежемесячные амортизационные отчисления будут рассчитаны линейным образом и составят:

$$H_a = \frac{1}{T} = \frac{1}{24} = 0.42$$

$$A_{\text{месяц}} = C_{\text{ноутбуков}} \cdot H_a = \frac{36,000 \cdot 2}{24} = 3,000 \text{ руб.}$$

#	**Этап итерации**	**Результат**	**Сроки (дни)**	**Затраты (руб)**
1	Анализ полученных за предыдущую итерацию отзывов	Список функционала, который требуется реализовать	2	3040
2	Анализ и проектирование решения задачи по реализации этого функционала	Набор алгоритмов, дизайн интерфейса, проектные диаграммы	5	7600
3	Разработка прототипа	Прототип с реализованным, но не протестированным функционалом	5-10	7600-15200
4	Внутреннее тестирование	Закрытие как можно большего количества критических ошибок за ограниченный период времени	2	3040
5	Внешнее тестирование ограниченным числом пользователей с получением от них отзывов и автоматически генерируемых программой отчетов о выполнении работы программы	Внесение исправлений в элементы дизайна, проекта функционала, исправление выявленных в ходе внешнего тестирования ошибок	10	15200
6	Подготовка к релизу и релиз с обновленным функционалом	Список изменений на сайте продукта и новостных ресурсах, уведомление клиентов об обновлении, готовая новая версия продукта, готовая для загрузки на клиентские машины	1	1520
∑			25-35	38000-53200

Таблица 4.2 – Этапы выполнения итерации жизненного цикла

4.6.4 Итоги

В итоге, месячная стоимость продукта составляет 58328 руб., а объем первоначальных инвестиций составит

$$24280 + 58328 = 109608 \text{ руб.}$$

4.7 Организационный план

Над проектом работает один программист, отлично разбирающийся в исходном проекте и в предметной области решаемой задачи.

Работа с кодом осуществляется при помощи технологии git с хостинг-сервисом BitBucket, предоставляющий бесплатные приватные репозитории для разработки.

Общение с клиентами происходит через почту на сервисе GMail.

4.8 Финансовый план

4.8.1 План прибылей и убытков

Таблица 4.3 представляет собой финансовый документ, в котором отражаются доходы, расходы и финансовые результаты за период реализации проекта. Как видно из таблицы 4.3, продажа анализатора является безубыточной даже при очень низкой цене за подписку и небольшом приросте потребителей. Основная статья затрат – зарплата единственному разработчику.

4.8.2 План движения денежных средств

Таблица 4.4 представляет собой финансовый документ, в котором отражаются денежные потоки от операционной (текущей), инвестиционной и финансовой деятельности на планируемый период.

Показатели			Квартал				Всего
#			I	II	III	IV	
1	Выручка-нетто (без учета НДС) от реализации		360000	450000	540000	630000	1980000
2	Переменные производственные затраты		127680	127680	127680	127680	510720
	1	Переменные материальные затраты	0	0	0	0	0
	2	Переменные затраты на оплату труда	127680	127680	127680	127680	510720
	3	Переменные общепроизводственные затраты	0	0	0	0	0
3	Валовая прибыль		223320	313320	403320	493320	1433280
4	Переменные управленческие и коммерческие затраты		0	0	0	0	0
5	Маржинальная прибыль		232320	322320	412320	502320	1469280
6	Постоянные затраты		9000	9000	9000	9000	36000
	2	Постоянные общепроизводственные затраты	9000	9000	9000	9000	36000
	3	Постоянные управленческие и коммерческие затраты	0	0	0	0	0
7	Прибыль от продаж		223320	313320	403320	493320	1433280
8	Прочие доходы и расходы		24280	0	0	0	24280
	1	Прочие доходы	0	0	0	0	0
	2	Прочие расходы	24280	0	0	0	24280
9	Прибыль до налогообложения		199040	313320	403320	493320	1409000
10	Налог на прибыль (20%)		39808	62664	80664	98664	281800
11	Чистая (нераспределенная) прибыль		159232	250656	322656	394656	1127200
12	Капитализируемая прибыль						
	1	Резервы	35000	194232	444888	767544	1162200
	2	Реинвестиции	0	0	0	0	0
13	Потребляемая прибыль						
	1	Дивиденды	0	0	0	0	0
	2	Прочие цели	0	0	0	0	0

Таблица 4.3 – План прибылей и убытков

4.8.3 Оценка эффективности реализации проекта

По имеющимся данным, будем брать за срок отчетности - один квартал. Тогда ср. чистая прибыль из таблицы 4.3:

$$\overline{CF}_T = \frac{1}{T}\sum_{t=1}^{T} CF_t = 281800$$

Ставку дисконта примем за $R = 16\%$ в квартал, постаравшись вложить туда и высокие риски, и инфляцию. Если даже с таким R эффективнось будет высока, то сомневаться в рентабельности и выгодности инвестиций будет сложно.

Единственные инвестиции I, которые мы получаем, это 35000 руб

Показатели			Квартал				Всего
#			I	II	III	IV	
1	Остаток денежных средств на начало периода		35000	145208	357560	641912	1179680
2	Поступление денежных средств		424800	531000	637200	743400	2336400
	1	Поступления от продажи продажи продукции	424800	531000	637200	743400	2336400
	2	Прочие поступления	0	0	0	0	0
3	Всего наличие денежных средств		459800	676208	994760	1385312	3516080
4	Выбытие денежных средств		136680	136680	136680	136680	546720
	1	Оплата поставщикам	0	0	0	0	0
	2	Оплата труда	127680	127680	127680	127680	510720
	3	Оплата общепроизводственных расходов	9000	9000	9000	9000	36000
	4	Оплата управленческих и коммерческих расходов	0	0	0	0	0
5	Инвестиции		35000	0	0	0	35000
6	Уплата налогов		142912	181968	216168	250368	791416
	1	НДС	64800	81000	97200	113400	356400
	2	Отчисления на соц. нужды	38304	38304	38304	38304	153216
	3	Налог на прибыль	39808	62664	80664	98664	281800
	4	Прочие налоги	0	0	0	0	0
7	Всего оттоков		314592	318648	352848	387048	1373136
8	Чистый денежный поток		145208	357560	641912	998264	2142944
9	Дополнительное финансирование		0	0	0	0	0
	1	Привлечение кредитов	0	0	0	0	0
	2	Погашение кредитов	0	0	0	0	0
	3	Погашение %% по кредиту	0	0	0	0	0
10	Остаток денежных средств на конец периода		145208	357560	641912	998264	2142944

Таблица 4.4 – План движения денежных средств

изначально требующегося капитала. Больше во внешних притоках мы не нуждаемся.

Имея данные по таблицам 4.3 и 4.4, считать будем за 1 год по каждому из 4-х кварталов.

4.8.3.1 Статические критерии

Простая норма рентабельности:

$$ROI = \frac{\overline{CF}}{I} = 8.0514$$

Простой период окупаемости

$$T = \frac{I}{CF} = 0.1242$$

4.8.3.2 Динамические критерии

Дисконтный период окупаемости:

$$DPP = \sum_{t=0}^{T} \frac{CF_t}{1+R^t} = -I + \sum_{t=1}^{T} \frac{CF_t}{1+R^t}$$

Посчитав до T = 1 уже видно, что в I квартале инвестиции окупятся:

$$DPP_{T=1} = -35000 + \frac{159232}{1+0.16} = 102268.9655$$

Уточняя результат: $\frac{Balance}{CF_{disc\_I}} = \frac{35000}{159232} = 0.2198$ I квартала.

Чистая текущая стоимость проекта:

$$NPV = -I + \sum_{t=1}^{T} \frac{CF_t}{(1+R)^t} =$$

$$= -35000 + 137268.9655 + 186278.2402 + 206712.0423 + 217964.9955 =$$

$$= 713224.2435$$

Критерий NPV > 0 является обязательным и в нашем случае он выполняется.

Дисконтированный индекс рентабельности: В данном конкретном случае DPI будет равен обычному индексу рентабельности PI, поскольку команда нужнается только в первоначальных инвестициях.

$$DPI = \frac{NPV}{\sum_{t=0}^{T} \frac{I_t}{(1+R)^t}} = \frac{NPV}{I} = 20.3778$$

Внутренняя норма рентабельности: IRR - это такое R, что NPV = 0. В

итоге требуется решить уравнение:

$$-I + \sum_{t=1}^{T} \frac{CF_t}{(1 + IRR)^t} = 0$$

Рассчитанное IRR = 504%

4.9 Анализ и оценка рисков

Проведем анализ рисков *сценарным* способом:

Пессимистичный прогноз, когда на количество подписчиков в 2000 пользователей придется выходить не 1 квартал, а весь год, начиная с 500 в первом квартале. В таком случае:

- ROI = 1.88
- T = 0.5319
- DPP = 2.422
- NPV = 108817.2257
- DPI = 3.1091
- IRR = 57%

Как видно, даже в ситуации, когда рост пользователей будет в 4 раза медленнее, инвестиции окупятся уже в 3 квартале.

Реалистичный прогноз был рассмотрен в ходе составления настоящего Бизнес-плана и в разделе 4.8.3 были получены следующие значения:

- ROI = 8.0514
- T = 0.1242
- DPP = 0.2198
- NPV = 713224.2435
- DPI = 20.3778

- IRR = 504%

Оптимистичный прогноз, когда количество пользователей не только начинается с 2000, но и растет в 2 раза быстрее. В таком случае:

- ROI = 11,1371
- T = 0.0898
- DPP = 0.2198
- NPV = 978281.5546
- DPI = 27.9809
- IRR = 539%

Таким образом, даже при пессимистичном прогнозе инвестиции вернутся в течении года. Оптимистичный прогноз не сильно улучшает ситуацию, но на такой быстрый рост, с другой стороны, надеяться не приходится.

ЗАКЛЮЧЕНИЕ

Кратко (на одну-две страницы) описать основные результаты работы, проанализировать их соответствие поставленной цели работы, показать рекомендации по конкретному использованию результатов исследования и перспективы дальнейшего развития работы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Robotics software frameworks for multi-agent robotic systems development / P. Iñigo-Blasco [и др.] // Robotics and Autonomous Systems. — 2012. — Т. 60, № 6. — С. 803—821.
2. ROS.org | Powering the world's robots [Электронный ресурс] / Open Source Robotics Foundation. — URL: http://www.ros.org/ (дата обр. 10.05.2018).
3. Mira-middleware for robotic applications / E. Einhorn [и др.] // Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on. — IEEE. 2012. — С. 2591—2598.
4. MIRA: MIRA Reference Documentation [Электронный ресурс]. — URL: http://www.mira-project.org/MIRA-doc/index.html (дата обр. 21.12.2017).
5. MOOS : Main - Documentation browse [Электронный ресурс]. — URL: http://www.robots.ox.ac.uk/~mobile/MOOS/wiki/pmwiki.php/Main/Documentation (дата обр. 21.12.2017).
6. *Archiveteam*. gitorious valhalla [Электронный ресурс]. — URL: https://gitorious.org/ (дата обр. 10.05.2018).
7. The Orocos Toolchain | The Orocos Project [Электронный ресурс]. — URL: http://www.orocos.org/orocos/toolchain (дата обр. 21.12.2017).
8. Rock the Robot Construction Kit : Table of Contents [Электронный ресурс]. — URL: https://www.rock-robotics.org/stable/documentation/toc.html/ (дата обр. 21.12.2017).
9. Overview & Download - Thymio & Aseba [Электронный ресурс]. — URL: https://www.thymio.org/en:start (дата обр. 21.12.2017).
10. *Ulm H.* SmartSoft Approach [Электронный ресурс]. — URL: http://www.servicerobotik-ulm.de/drupal/?q=node/19 (дата обр. 21.12.2017).

11. YARP : Welcome to YARP [Электронный ресурс]. — URL: http://www.yarp.it/index.html (дата обр. 21.12.2017).

12. General Information | OpenRTM-aist [Электронный ресурс]. — URL: http://www.openrtm.org/openrtm/en/node/495 (дата обр. 21.12.2017).

13. OpenRTM-aist web on the github | OpenRTM-aist [Электронный ресурс]. — URL: http://openrtm.org/ (дата обр. 10.05.2018).

14. GitHub repository aldebaran/urbi: Robotic programming language [Электронный ресурс]. — URL: https://github.com/aldebaran/urbi (дата обр. 21.12.2017).

15. URBI web-site status: is comming soon [Электронный ресурс]. — URL: http://www.urbiforge.org/ (дата обр. 21.12.2017).

16. *Dustin E.*, *Rashka J.*, *Paul J.* Automated software testing: introduction, management, and performance. — Addison-Wesley Professional, 1999. — С. 39, 250.

17. *Curnow H. J.*, *Wichmann B. A.* A synthetic benchmark // The Computer Journal. — 1976. — Т. 19, № 1. — С. 43—49.

18. *Bruun N.* nickbruun/hayai: C++ benchmarking framework [Электронный ресурс]. — URL: https://github.com/nickbruun/hayai (дата обр. 11.05.2017).

19. *Bruun N.* Easy C++ benchmarking – Nick Bruun [Электронный ресурс]. — URL: https://bruun.co/2012/02/07/easy-cpp-benchmarking (дата обр. 11.05.2017).

20. *Farrier J.* DigitalInBlue/Celero: C++ Benchmark Authoring Library/Framework [Электронный ресурс]. — URL: https://github.com/DigitalInBlue/Celero/ (дата обр. 11.05.2017).

21. *Martinho F.* libnonius/nonius: A C++ micro-benchmarking framework [Электронный ресурс]. — URL: https://github.com/libnonius/nonius (дата обр. 11.05.2017).

22. *Hamon D.* google/benchmark: A microbenchmark support library [Электронный ресурс] / Google LLC. — URL: https://github.com/google/benchmark/ (дата обр. 11.05.2017).

23. *Paoloni G.* How to benchmark code execution times on Intel IA-32 and IA-64 instruction set architectures // Intel Corporation. — 2010. — С. 123.

24. *Bruyninckx H.* Open robot control software: the OROCOS project // Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on. Т. 3. — IEEE. 2001. — С. 2523—2528.

25. *Metta G.*, *Fitzpatrick P.*, *Natale L.* YARP: yet another robot platform // International Journal of Advanced Robotic Systems. — 2006. — Т. 3, № 1. — С. 8.

26. *Mohamed N.*, *Al-Jaroodi J.*, *Jawhar I.* Middleware for robotics: A survey // Robotics, Automation and Mechatronics, 2008 IEEE Conference on. — Ieee. 2008. — С. 736—742.

27. *Elkady A.*, *Sobh T.* Robotics middleware: A comprehensive literature survey and attribute-based bibliography // Journal of Robotics. — 2012. — Т. 2012.

28. *O'Kane J. M.* A gentle introduction to ROS. — 2014.

29. ROS/TCPROS - ROS Wiki [Электронный ресурс] / Open Source Robotics Foundation. — URL: http://wiki.ros.org/ROS/TCPROS (дата обр. 12.05.2017).

30. ROS/Concepts - ROS Wiki [Электронный ресурс] / Open Source Robotics Foundation. — URL: http://wiki.ros.org/ROS/Concepts (дата обр. 12.05.2017).

31. *Fitzpatrick P.* YARP: Port Power, Going Further with Ports [Электронный ресурс]. — URL: http://yarp.it/port_expert.html (дата обр. 12.05.2017).

32. MIRA: Units [Электронный ресурс]. — URL: http://www.mira-project.org/MIRA-doc/UnitsPage.html (дата обр. 13.05.2017).

33. *Soetens P.* The Orocos Component Builder's Manual [Электронный ресурс] / Flanders Mechatronics Technology Centre. — URL: https://orocos-toolchain.github.io/rtt/toolchain-2.9/xml/orocos-components-manual.html (дата обр. 13.05.2017).

34. *Bruyninckx H.* OROCOS component model | The Orocos Project [Электронный ресурс]. — URL: http://www.orocos.org/forum/orocos/orocos-users/orocos-component-model#comment-32633 (дата обр. 13.05.2017).

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД DOCKER-ФАЙЛОВ

Листинг А.1 – ubuntu-dev:latest

```
FROM ubuntu:16.04
MAINTAINER JAkutenshi <jakutenshi@gmail.com>

WORKDIR /root/
ADD vim_setup.tar.gz /root/

RUN apt-get update && DEBIAN_FRONTEND=noninteractive apt-get install -y \
    locales          \
    lsb-release      \
    ranger           \
    htop             \
    vim              \
    git              \
    tmux             \
    ssh              \
    xorg             \
    xauth            \
    build-essential  \
    cmake            \
    libxml2-dev       \
    libboost-all-dev \
    pkg-config

RUN locale-gen en_US.UTF-8 ru_RU.UTF-8
ENV LANG=en_US.UTF-8 LANGUAGE=en_US:en LC_ALL=en_US.UTF-8

EXPOSE 22
```

Листинг А.2 – ubuntu-ros:latest

```
FROM jakutenshi/ubuntu-dev

RUN sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.
    list.d/ros-latest.list' \
    && apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv-key 421
        C365BD9FF1F717815A3895523BAEEB01FA116      \
    && apt-get update  \
    && apt-get install -y ros-kinetic-desktop-full \
    && rosdep init     \
    && rosdep update   \
    && echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc \
    && apt-get install -y python-rosinstall python-rosinstall-generator python-wstool
```

Листинг А.3 – ubuntu-yarp:latest

```
FROM jakutenshi/ubuntu-dev:latest

RUN apt-get update &&          \
    apt-get install -y         \
    cmake-curses-gui           \
    libeigen3-dev              \
    libace-dev                 \
    libedit-dev                \
    qtbase5-dev                \
    qtdeclarative5-dev         \
    qtmultimedia5-dev          \
    qml-module-qtquick2        \
    qml-module-qtquick-window2 \
    qml-module-qtmultimedia    \
    qml-module-qtquick-dialogs \
    qml-module-qtquick-controls

ADD yarp.tar.gz /root/
WORKDIR /root/yarp/build/
RUN cmake -DCREATE_GUIS=ON -DCREATE_LIB_MATH=ON ..
RUN make && make install && ldconfig
```

Листинг А.4 – ubuntu-mira:latest

```
FROM jakutenshi/ubuntu-dev:latest

RUN apt-get update           \
    && apt-get install -y    \
    zip                      \
    unzip                    \
    subversion               \
    libxml2-dev              \
    libssl-dev               \
    libsqlite3-dev           \
    libboost-all-dev         \
    libogre-1.9-dev          \
    libsvn-dev               \
    libopencv-core-dev       \
    libopencv-flann-dev      \
    libopencv-ml-dev         \
    libopencv-calib3d-dev    \
    libopencv-dev            \
    libopencv-imgproc-dev    \
    libopencv-gpu-dev        \
    libopencv-objdetect-dev  \
    libopencv-contrib-dev    \
    libopencv-features2d-dev \
    libopencv-legacy-dev     \
    libopencv-highgui-dev \
    libopencv-video-dev   \
    libcvaux-dev          \
    libcv-dev             \
    binutils-dev          \
    libiberty-dev         \
# Qt4
    qt4-dev-tools         \
    libqt4-dev            \
    libqt4-opengl-dev     \
    libqtwebkit-dev       \
    libqwt-dev            \
# Qt5
    qt5-default           \
    libqwt-qt5-dev        \
    libqt5webkit5-dev     \
    libqwtmathml-qt5-dev  \
    libqt5opengl5-dev     \
    libqt5svg5-dev        \
    qt*5-dev              \
    qttools5-dev-tools

WORKDIR /root/
ADD mira-installer-no-interact.sh /root/
RUN chmod +x /root/mira-installer-no-interact.sh \
    && /root/mira-installer-no-interact.sh
RUN echo "export MIRA_PATH=/root/mira" >> .bashrc                              \
 && echo "export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/root/mira/lib" >> .bashrc \
 && echo "export PATH=$PATH:/root/mira/bin" >> .bashrc
```

Листинг А.5 – ubuntu-orocos:latest

```
FROM jakutenshi/ubuntu-orocos-toolchain-2.8-16.04:manual

# Step 1
# RUN git config --global user.name "Your_name"
# RUN git config --global user.email yourname@yourmail.com
# RUN apt-get update
# RUN apt-get install -y ruby-dev bundler clang
# RUN git clone https://github.com/orocos-toolchain/build.git
# WORKDIR build/
# RUN git checkout toolchain-2.8-16.04

# Step 2-3
WORKDIR /root/
ADD orocos-before-update.tar.gz /root/

# Step 4
WORKDIR /root/orocos
RUN echo "source /root/orocos/env.sh" >> ~/.bashrc                  \
    && /bin/bash -c ". env.sh && autoproj update && autoproj build" \
# Clear workspace
    && rm -rf /root/build/ && rm /root/manual_from_step_2.md

WORKDIR /root/
```

ПРИЛОЖЕНИЕ В. ИСХОДНЫЙ КОД АВТОМАТИЧЕСКОГО АНАЛИЗАТОРА РЕЗУЛЬТАТОВ ТЕСТИРОВАНИЯ ПРОИЗВОДИТЕЛЬНОСТИ

```
import json
import re
import csv
import xml.etree.ElementTree as ET
import itertools

json_file_urls = []


#        ,    bm_id
#          google benchmark       long double,    python - int.
#    :             ,         ,           15
def id_compare(first, second) -> bool:
    first = str(int(first))[0:15]
    second = str(int(second))[0:15]
    return first == second


def read_json_from_file(url: str) -> json:
    with open(url, 'r') as input_json:
        json_obj = json.load(input_json)
    return json_obj


def write_json_to_file(filename: str, json_obj):
    with open(filename, 'w') as json_in:
        json.dump(json_obj, json_in, separators=(',', ': '), indent=4)


def process_primary(primary: ET.Element):
    json_url = primary.attrib['url']
    primary_json = read_json_from_file(json_url)

    #        ,                 _mean, _meidan    _stddev
    regexpr = re.compile(r'_mean|_median|_stddev$')
    benchmarks = [benchmark for benchmark in primary_json['benchmarks'] if not re.search(regexpr,
        benchmark["name"])]

    #
    benchmarks_descriptions = {}
    for benchmark in primary.findall("benchmark"):
        name = benchmark.attrib["name"]
        benchmarks_descriptions[name] = []
        for factor in benchmark.findall("factor"):
            benchmarks_descriptions[name].append(factor.attrib["name"])
    write_json_to_file("benchmarks_descriptions.json", benchmarks_descriptions)

    #                json -
    for secondary in primary.findall('secondary'):
        url = secondary.attrib['url']
        key = secondary.attrib['key'] # key == "bm_id"
        secondary_json = read_json_from_file(url) #   json       ,

        #                                 json
        for benchmark in benchmarks:
            #                    ?
            if key in benchmark.keys():
                #    bm_id   ,
                for secondary_measurement in secondary_json:
                    #            id
                    if id_compare(benchmark[key], secondary_measurement[key]):
                        #
                        measurements = list(secondary_measurement.keys())
                        #        key
                        measurements.remove(key)
                        for measure in measurements:
                            benchmark[measure] = secondary_measurement[measure]
                        #
                        break
        ########################
        #
    ############################
    return benchmarks


```

```
def process_xml(url: str):
    tree = ET.parse(url)
    root = tree.getroot()
    primaries = []
    for primary in root:
        benchmarks = process_primary(primary)
        for benchmark in benchmarks:
            primaries.append(benchmark)

    #                 full.json
    write_json_to_file("full.json", primaries)


def process_full_json():
    with open("full.json", "r") as input_json:
        benchmarks = json.load(input_json)

    #                     :
    # FX_val    1 2 3 4 5
    # [Mi]_mean 1 2 3 4 5
    # [Mi]_sd   1 2 3 4 5
    #              :
    # FX -
    # Mi -
    # Fj -
    #          := BM_NAME/Fj/F_NAME/Fj - Mi
    #                         ,   Mi := all

    #
    benchmarks_list = benchmarks #
    measures_grouped = {}        #    ,
    while len(benchmarks_list) != 0:
        benchmark = benchmarks_list.pop(0)
        #
        benchmark.pop("bm_id", None)
        benchmark.pop("time_unit", None)
        benchmark.pop("iterations", None)
        if benchmark["name"] not in measures_grouped.keys():
            #                          ,           ,
            measures_grouped[benchmark["name"]] = {
                "count": 0,
                "measures": []
            }

        measures_node = measures_grouped[benchmark["name"]]
        measures_node["count"] += 1
        #              -
        measure = {}
        for key in benchmark.keys():
            if key != "name":
                measure[key] = benchmark[key]
        #
        measures_node["measures"].append(measure)
    # write_json_to_file("tmp.json", measures_grouped)

    #                       ,
    benchmarks_processed = {}
    for benchmark in measures_grouped:
        count = measures_grouped[benchmark]["count"]        #
        measures = measures_grouped[benchmark]["measures"]  #
        #      -
        if len(measures) == 0:
            print("SUDDENLY NO MEASURES IN BM_NAME=" + benchmark)
            return
        benchmarks_processed[benchmark] = {}
        for measure in measures:
            for measure_name in measure:
                if measure_name not in benchmarks_processed[benchmark]:
                    benchmarks_processed[benchmark][measure_name] = []
                benchmarks_processed[benchmark][measure_name].append(measure[measure_name])
    write_json_to_file("benchmark_processed.json", benchmarks_processed)

    #            P=95%
    S = {
        2: 12.706,
        3: 4.303,
        4: 3.182,
        5: 2.776,
        6: 2.571,
        7: 2.447,
        8: 2.365,
        9: 2.306,
        10: 2.262,
        11: 2.228,
```

```
        12: 2.201,
        13: 2.179,
        14: 2.160,
        15: 2.145,
        16: 2.131,
        17: 2.120,
        18: 2.110,
        19: 2.101,
        20: 2.093,
        21: 2.086,
        22: 2.080,
        23: 2.074,
        24: 2.069,
        25: 2.064,
        26: 2.060,
        27: 2.056,
        28: 2.052,
        29: 2.048,
        30: 2.045
    }
    #
    benchmarks_results = {}
    for benchmark_name in benchmarks_processed:
        measures = benchmarks_processed[benchmark_name]
        benchmarks_results[benchmark_name] = {}
        for measure_name in measures:
            results = measures[measure_name]
            #
            mean = 0.0
            for value in results:
                mean += value
            mean = float(mean) / count
            #
            deviations = []
            deviations_in_square = []
            deviations_in_square_sum = 0
            for value in results:
                deviation = mean - value
                deviations.append(deviation)
                deviation_in_square = pow(deviation, 2)
                deviations_in_square.append(deviation_in_square)
                deviations_in_square_sum += deviation_in_square
            std_mean_dev = pow(deviations_in_square_sum / (count * (count - 1)), 0.5)
            error = S[count] * std_mean_dev

            benchmarks_results[benchmark_name][measure_name] = {
                "mean": mean,
                "error": error
            }
    write_json_to_file("benchmarks_results.json", benchmarks_results)


def process_results_to_rmd():
    #
    benchmarks_descriptions_from_json = read_json_from_file("benchmarks_descriptions.json")

    #                          .                                  -
    benchmarks_descriptions = {}
    for benchmark_name in benchmarks_descriptions_from_json:
        for factor_name in benchmarks_descriptions_from_json[benchmark_name]:
            if benchmark_name not in benchmarks_descriptions:
                benchmarks_descriptions[benchmark_name] = {}
            benchmarks_descriptions[benchmark_name][factor_name] = []

    #                                        ,
    #
    benchmarks = read_json_from_file("benchmarks_results.json")
    benchmark_name_regexp = re.compile(r"([a-zA-Z_]*\/[a-zA-Z_]*)\/.*")
    factors_string_regexp = re.compile(r"[a-zA-Z_]*\/[a-zA-Z_]*(\/.*)")
    factor_match = re.compile(r"\/(\d*)")

    for benchmark in benchmarks:
        name = benchmark_name_regexp.findall(benchmark)[0]
        if name in benchmarks_descriptions:
            factors = factors_string_regexp.findall(benchmark)
            factors_list = factor_match.findall(factors[0])
            for i in range(0, len(factors_list)):
                factor_name = benchmarks_descriptions_from_json[name][i]  #
                if factors_list[i] not in benchmarks_descriptions[name][factor_name]:
                    benchmarks_descriptions[name][factor_name].append(factors_list[i])
    #print(benchmarks_descriptions)

    #
    #
```

```
    tables = []
    for benchmark_info in benchmarks_descriptions:
        factors_dict = benchmarks_descriptions[benchmark_info]
        #print(factors_dict)
        factors_order = list(benchmarks_descriptions[benchmark_info].keys())
        #                   ,           -,
        for current_factor in factors_dict:
            fixed_factors = []
            #
            for factor in factors_dict:
                if factor != current_factor:
                    fixed_factors.append(factors_dict[factor])
                else:
                    fixed_factors.append([-1])
            #
            combinations = list(itertools.product(*fixed_factors))
            #
            current_factor_place = factors_order.index(current_factor)
            for combination in combinations:
                #
                table_name = "Benchmark name is \"" + benchmark_info + "\" by var=" + current_factor
                     + " where"
                #
                table_pattern = benchmark_info
                index  = 0
                for factor in combination:
                    if index != current_factor_place:
                        table_name += " " + factors_order[index] + "=" + factor
                        table_pattern += "\/" + factor
                    else:
                        table_pattern += r"\/(\d*)"
                    index += 1
                table_pattern += "$"
                #print(table_name)
                #print(table_pattern)

                #
                regexp = re.compile(table_pattern)
                valid_benchmarks = []
                for benchmark in benchmarks:
                    if regexp.findall(benchmark):
                        valid_benchmarks.append(benchmark)

                #
                table = {}
                table["factor"] = {
                    "name": current_factor,
                    "values": factors_dict[current_factor]
                }
                for benchmark in valid_benchmarks:
                    for measure in benchmarks[benchmark]:
                        if measure not in table:
                            table[measure] = {
                                "mean": [],
                                "error": []
                            }
                        table[measure]["mean"].append(benchmarks[benchmark][measure]["mean"])
                        table[measure]["error"].append(benchmarks[benchmark][measure]["error"])

                tables.append({
                    "title": table_name,
                    "name": benchmark_info,
                    "table": table
                })
    #print(tables)
    #
    write_json_to_file("tables.json", tables)


def list_to_rvector_var(vec_name: str, list: list) -> str:
    output = vec_name + " <- c("
    for v in list:
        output += str(v) + ", "
    output = output[0 : len(output) - 2] + ")\n"
    #output += vec_name + "\n"
    return output

def list_to_rvector(list: list) -> str:
    output = "c("
    for v in list:
        output += str(v) + ", "
    output = output[0 : len(output) - 2] + ")"
    return output

```

```
def config_rmd():
    rmd_config = ET.parse("rmd_config.xml")
    rmd_config_root = rmd_config.getroot()

    benchmarks_tables_config = {}
    for benchmark in rmd_config_root.findall("benchmark"):
        benchmark_name = benchmark.attrib["name"]
        benchmarks_tables_config[benchmark_name] = {}
        titles = benchmark.findall("titles")[0]
        titles_root = titles.findall("remap")
        benchmarks_tables_config[benchmark_name]["titles"] = {}
        for remap in titles_root:
            benchmarks_tables_config[benchmark_name]["titles"][remap.attrib["title"]] = remap.attrib
                ["remap"]

        plots_root = benchmark.findall("plots")[0]
        plots = plots_root.findall("plot")
        benchmark_plots = benchmarks_tables_config[benchmark_name]["plots"] = []
        for plot in plots:
            measures = []
            for measure in plot.findall("measure"):
                measures.append({
                    "name": measure.attrib["name"],
                    "col": measure.attrib["col"],
                    "type": measure.attrib["type"],
                    "lty": measure.attrib["lty"]
                })
            benchmark_plots.append({
                "title": plot.attrib["title"],
                "ylab": plot.attrib["ylab"],
                "values": plot.attrib["values"],
                "measures": measures
            })

    write_json_to_file("benchmarks_tables_config.json", benchmarks_tables_config)


def generate_rmd():
    tables = read_json_from_file("tables.json")
    benchmarks_tables_config = read_json_from_file("benchmarks_tables_config.json")

    head = '''---
title:    "         "
author:     "    "
output:
    html_document:
        toc: true
        theme: united
---
'''
    with open("result.rmd", 'w') as out:
        out.write(head)
        for table in tables:
            #
            table_config = benchmarks_tables_config[table["name"]]
            print(table_config)
            vectors_list = []
            #
            out.write("## " + table["title"] + "\n")
            out.write("``` {r echo=FALSE}\n" +
                      list_to_rvector_var(table["table"]["factor"]["name"], table["table"]["factor"
                          ]["values"])
                      )
            vectors_list.append({
                "name": table["table"]["factor"]["name"],
                "title": "factor",
                "vector": table["table"]["factor"]["values"]
            })
            #
            for measure in table["table"]:
                if measure != "factor":
                    measure_mean_name = measure
                    measure_mean_list = table["table"][measure]["mean"]

                    out.write(list_to_rvector_var(measure_mean_name, measure_mean_list))
                    vectors_list.append({
                        "name": measure_mean_name,
                        "title": table_config["titles"][measure_mean_name],
                        "vector": measure_mean_list
                    })

            #
            df = "df <- data.frame("
            for vector in vectors_list:
```

```
                df += "\"" + vector["title"] + "\"=" + vector["name"] + ", "
            df = df[0 : len(df) - 2] + ")\n"
            out.write(df + "df\n")

            #
            for plot in table_config["plots"]:
                #
                measures_max = []
                measures_min = []
                measures_max_err = []
                for measure in plot["measures"]:
                    measures_max.append(max(table["table"][measure["name"]]["mean"]))
                    measures_min.append(min(table["table"][measure["name"]]["mean"]))
                    measures_max_err.append(max(table["table"][measure["name"]]["error"]))

                plot_max = max(measures_max)
                plot_min = min(measures_min)
                vector_err_max = max(measures_max_err)
                height = plot_max - plot_min + 2 * vector_err_max
                border = height * 1.05
                ylim = "c(" + str(plot_min - vector_err_max - border) + ", "
                ylim += str(plot_max + vector_err_max + border) + ")"

                factor_name = table["table"]["factor"]["name"]
                r_plot = ""
                legend_titles = []
                legend_cols = []
                legend_lty = []
                legend_pch = []
                for index_measure in range(0, len(plot["measures"])):
                    measure = plot["measures"][index_measure]
                    legend_titles.append("\"" + table_config["titles"][measure["name"]] + "\"")
                    legend_cols.append("\"" + measure["col"] + "\"")
                    legend_lty.append(measure["lty"])
                    legend_pch.append("\"" + measure["type"] + "\"")

                    if index_measure == 0:
                        r_plot += "plot(" + factor_name + ", " + measure["name"]
                        r_plot += ", type=\"" + measure["type"] + "\", col=\"" + measure["col"] + "
                            \", lty=" + measure["lty"] + ", "
                        r_plot += "ylim=" + ylim + ", xlab=\"\", ylab=\"\")\n"
                        #
                        r_plot += "title(main=\"" + plot["title"] + "\", xlab=\"" + table_config["
                            titles"][factor_name]
                        r_plot +=  "\", ylab=\"" + plot["ylab"] + "\")\n"
                    else:
                        r_plot += "points(" + factor_name + ", " + measure["name"] + ", "
                        r_plot += "col=\"" + measure["col"] + "\", pch=\"" + measure["type"] + "\")\
                            n"
                        r_plot += "lines(" + factor_name + ", " + measure["name"] + ", "
                        r_plot += "col=\"" + measure["col"] + "\", lty=" + measure["lty"] + ")\n"

                    r_plot += list_to_rvector_var("err", table["table"][measure["name"]]["error"])
                    r_plot += "avg <- " + measure["name"] + "\n"
                    r_plot += "arrows(" + factor_name + ", avg-err, " + factor_name
                    r_plot += ", avg+err, length=0.05, angle=90, code=3, col=\"" + measure["col"] +
                        "\")\n"
                    if plot["values"] == "True":
                        r_plot += "text(" + factor_name + ", " + measure["name"] + ", labels=" +
                            measure["name"] + ", cex= 0.7, pos=3)\n"
                r_plot += "legend(\"bottomright\", legend=" + list_to_rvector(legend_titles)
                r_plot += ", col=" + list_to_rvector(legend_cols) + ", lty=" + list_to_rvector(
                    legend_lty)
                r_plot += ", pch=" + list_to_rvector(legend_pch) + ", ncol=1)\n"


                out.write(r_plot)
            out.write("```\n\n")


if __name__ == "__main__":
    process_xml("merge_config.xml")
    process_full_json()
    process_results_to_rmd()
    config_rmd()
    generate_rmd()
```

ПРИЛОЖЕНИЕ С. ПРИМЕРЫ ФАЙЛОВ КОНФИГУРАЦИИ

С.1 Файл конфигурации исходных json-файлов

```
<merger>
    <json url="result.json">
        <benchmark name="PublisherFixture/ROS_BM_Publish_method">
            <factor name="subs_count"/>
            <factor name="size"/>
            <factor name="buffer"/>
        </benchmark>
        <secondary url="subscriber_results.json" key="bm_id"/>
    </json>
</merger>
```

С.2 Файл конфигурации rmarkdown отчета

```
<rmd>
    <benchmark name="PublisherFixture/ROS_BM_Publish_method">
        <titles>
            <remap title="subs_count" remap="  -      "/>
            <remap title="size" remap="         ()"/>
            <remap title="buffer" remap="      "/>
            <remap title="real_time" remap="        ()"/>
            <remap title="cpu_time" remap="       CPU  ()"/>
            <remap title="bytes_per_second" remap="                (/)"/>
            <remap title="manual_latency" remap="          ()"/>
            <remap title="manual_bandwidth" remap="                (/)"/>
        </titles>
        <plots>
            <plot title="               " ylab="     ()" values="False">
                <measure name="real_time" col="black" type="o" lty="1"/>
                <measure name="cpu_time" col="red" type="X" lty="2"/>
                <!--measure name="manual_latency" col="blue" type="+" lty="3"/-->
            </plot>
            <plot title="                     " ylab="            (/)" values="True">
                <measure name="bytes_per_second" col="black" type="o" lty="1"/>
                <measure name="manual_bandwidth" col="red" type="X" lty="2"/>
            </plot>
        </plots>
    </benchmark>
</rmd>
```