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.

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	Недавно
MIRA	Да	Да	Недавно
MOOS	Да	Да	Недавно
OROCOS/Rock	Да	Да	2016
ASEBA	Да	Нет	Недавно
YARP	Да	Да	Недавно
OpenRTM-aist	Да	Нет	2016
URBI	Да	Нет	2016

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.

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

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

- интерфейсы между аппаратной частью и фреймворком с целью установления задержки реакции системы на окружение.

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, размер буфера и состояние инициализации.

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:
  - 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 таблицы 2.3 из столбцов 1-5, понимая трактовку последнего фактора – QoS – следующим образом: если QoS применяется, то организовать набор из постоянных 9 соединений без QoS и одного соединения, для которого применяются факторы в столбцах 1-4, а так же приоритеты QoS 5го столбца.

2.2.3. MIRA

В столбцах таблицы 2.4 приведены различные значения по выделенным ранее факторам для тестирования. Таким образом, множество всех сценариев

Таблица 2.2 – Факторизация параметров для тестирования производительности ROS

Тип взаимодействия	**Размер сообщения**	**Количество подписчиков или клиентов**	**Размер буфера**
Издатель/ подписчик	1 КБ	1	10
Сервисы	4 КБ	2	100
	16 КБ	4	1000
	64 КБ	8	2000
	256 КБ	16	5000
	1 МБ	32	10000
	4 МБ		
	16 МБ		
	64 МБ		
	256 МБ		
	1 ГБ		

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;
  - CORBA + MQueue;

Таким образом, в таблице 2.5 приведены различные значения по выделенным ранее факторам для тестирования, а множество всех сценариев тестов

Таблица 2.3 – Факторизация параметров для тестирования производительности YARP

Тип порта	**Транспорт соединения**	**Размер сообщения**	**Число входящих портов**	**QoS**	
				Статус	**Приоритет**
Обычный	tcp	1 КБ	1	Применяется	LOW
С буфером	udp	4 КБ	2	Не применяется	NORMAL
RPC	mcast	16 КБ	4		HIGH
	shmem	64 КБ	8		CRITICAL
	local	256 КБ	16		
	fast_tcp	1 МБ	32		
		4 МБ			
		16 МБ			
		64 МБ			
		256 МБ			
		1 ГБ			

Таблица 2.4 – Факторизация параметров для тестирования производительности MIRA

Подход к коммуникации	**Реализация модуля**	**Размер сообщения**	**Время хранения сообщений в очереди**
Шаблон наблюдатель	В одном процессе (разделяемая память)	1 КБ	1 мс
RPC	В разных процессах (TCP)	4 КБ	10 мс
		16 КБ	100 мс
		64 КБ	1 с
		256 КБ	10 с
		1 МБ	100 с
		16 МБ	
		64 МБ	
		256 МБ	
		1 ГБ	

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
			16 МБ	
			64 МБ	
			256 МБ	
			1 ГБ	

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

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;
- 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 (ПРИЛОЖЕНИЕ?). На рисунке (ССЫЛКА) показана иерархия связей Docker-образов.

2.3.2. Используемое API Google benchmark

2.3.3. Детали реализации тестовых случаев

2.3.3.1. ROS

Узел инициализируется при помощи `ros::init(int argc, char** argv)` и данные об узле передаются при инициализации объекта класса `ros::NodeHandle`.

Для получения возможности публиковать сообщения по определенному топику или подписываться на топики, требуется инициализировать объекты `ros::Publisher` для узла-издателя и `ros::Subscriber` для узла-подписчика. Инициализация происходит при помощи вызова методов объекта узла класса `ros::NodeHandle`:

- `advertise()` для публикации топика и инициализации объекта издателя, где `topic_name` - название топика, а

Сообщения в ROS описываются текстовым форматом, отдельно для сообщений для механизма топиков и отдельно для сервисов, но в ходе компиляции они преобразуются в заголовочные C++ файлы. Для сообщений сервисов создается несколько дополнительных заголовков: для запроса и ответа, а так же для случая, если ответ не требуется. Имеется возможность описать данные как массив, который в реализации C++ является объектом класса

.

2.3.3.2. YARP

2.3.3.3. MIRA

2.3.3.4. OROCOS

2.3.4. Обработка результатов

2.3.5. Автоматизация тестирования и обработки результатов

2.3.5.1. Архитектура

2.3.5.2. Реализация

2.3.5.3. Примеры работы

2.3.5.4. Тестирование

2.4. Выводы

3. РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ

3.1. Характеристики тестируемого окружения

3.1.1. Аппаратные

3.1.2. Программные

3.2. ROS

3.3. YARP

3.4. MIRA

3.5. OROCOS

3.6. Сравнение результатов

3.7. Выводы

4. ТЕХНИКО-ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ

4.1. Резюме

4.2. Описание продукции

4.3. Анализ рынка и сбыта продукции

4.4. Анализ конкурентов

4.5. План маркетинга

4.6. План производства

4.7. Организационный план

4.8. Финансовый план

4.9. Инвестиционный план и стратегия финансирования

4.10. Анализ и оценка рисков

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/
```