ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ И ТЕЛЕКОММУНИКАЦИИ
СПбГУТ))) Июнь 2017 Том 5 № 2 ISSN 2307-1303 http://www.itt.sut.ru
TELECOM IT
June 2017 Vol. 5 Iss. 2 ISSN 2307-1303 http://www.itt.sut.ru

ВЫСОКОПРОИЗВОДИТЕЛЬНАЯ МОДУЛЬНАЯ МНОГОПОТОЧНАЯ СИСТЕМА ОБМЕНА ДАННЫМИ ДЛЯ РОБОТОТЕХНИЧЕСКИХ КОМПЛЕКСОВ

А. Г. Ивин[1*], Д. И. Михальченко[1]

[1] СПИИРАН, Санкт-Петербург, 199178, Российская Федерация
* Адрес для переписки: arssivka@yandex.ru

Аннотация
На сегодняшний день робототехника является довольно молодой и активно развивающейся прикладной междисциплинарной наукой. Одной из основных проблем разработки программного обеспечения в этой области можно считать избыточные трудозатраты при проектировании программных компонентов для управления робототехнической системой. Особенно остро эта проблема встает при разработке мобильных роботов, в которых для повышения автономности используется энергоэффективные аппаратные вычислительные устройства с низкой производительностью. **Предмет исследования.** В данной работе для создания высокопроизводительной многопоточной модульной робототехнической системы исследованы области современных неблокирующих алгоритмов межпоточной синхронизации и алгоритмов сериализации без копирования. В работе представлено два прототипа фреймворка, асинхронное ядро которого разработано с использованием потокобезопасных неблокирующих очередей функторов. **Основной результат.** Результаты тестирования быстродействия системы рассылки сообщений показывают прирост производительности в 6-8 раз в сравнении с использованием библиотеки Boost Signals2. Так же наблюдается линейный прирост производительности в многоядерных системах при увеличении количества потоков. **Практическая значимость.** Разработанные алгоритмы можно применять в разработке ПО как для маломощных аппаратных платформ, так и в производительных высоконагруженных системах.

Ключевые слова
робототехника, асинхронность, многопоточность, lock-free, zero-copy.

Информация о статье
УДК 004.89
Язык статьи – русский.
Поступила в редакцию 16.05.17, принята к печати 02.06.17.

Ссылка для цитирования: Ивин А. Г., Михальченко Д. И. Высокопроизводительная модульная многопоточная система обмена данными для робототехнических комплексов // Информационные технологии и телекоммуникации. 2017. Том 5. № 2. С. 74–84.

HIGH-PERFORMANCE MODULAR MULTITHREADING COMMUNICATION SYSTEM FOR ROBOTS

A. Ivin[1*], D. Mikhalchenko[1]

[1] SPIIRAS, St. Petersburg, 199178, Russian Federation
* Corresponding author: arssivka@yandex.ru

Abstract—Today, robotics is a young and progressive section of applied science. One of the main problems of developing software for robots is excessive complexity in the design of software components controlling the robotic system. Especially complexity arises in the development of mobile robots. These robots are equipped with energy-efficient hardware devices with low performance in order to increase autonomy. In this paper, we study modern non-blocking algorithms for inter-thread synchronization and zero-copy serialization algorithms to create a high-performance multithreaded modular robotic system. This paper presents two prototypes of framework with using thread-safe non-blocking queues of functors. The benchmark results of message distribution system shows a performance gain of 6-8 times compared to the Boost Signals2 library. There is also a linear increase in performance in multi-core systems with an additional number of threads. Developed algorithms can be used in software development both for low-power hardware platforms, and in productive high-loaded systems.

Keywords—Robotics, asynchrony, multithreading, lock-free, zero-copy.

Article info
Article in Russian.
Received 16.05.17, accepted 02.06.17.

For citation: Ivin A., Mikhalchenko D.: High-Performance Modular Multithreading Communication System for Robots // Telecom IT. 2017. Vol. 5. Iss. 2. pp. 74–84 (in Russian).

Введение

В данной работе рассматриваются решения модульного асинхронного потокобезопасного робототехнического фреймворка. Фреймфорк ориентирован преимущественно на мобильную робототехнику с низкой вычислительной мощностью. Повышение производительности программной системы в настоящей работе предлагается обеспечивать за счет снижения использования количества блокирующих примитивов синхронизации в многопоточной среде путем использования неблокирующих аналогов и уменьшения количества операций копирования при межмодульном взаимодействии и десериализации сообщений. Исследование сконцентрировано на разработке модульной системы и коммуникации между модулями с использованием lock-free контейнеров и алгоритмов.

В работе [1] рассматривается алгоритм для передачи большого количества данных через UDP в высоконагруженной многопоточной системе, в которой исполь-

зуется высокопроизводительная реализация очереди без блокировок с открытым исходным кодом «moodycamel ConcurrentQueue» для повышения быстродействия в многопоточной среде. Данные при передаче между компонентами предполагается хранить в сериализированном виде. В работе [2] исследуются алгоритмы сериализации и десериализации для высокопроизводительных вычислительных систем. Лучшие результаты производительности показывают реализации Cap'n Proto и Google Flat Buffers, которые используют алгоритмы десериализации без копирования. Данные в таком формате можно передавать внутри системы сразу в сериализованном виде.

Анализ существующих робототехнических фреймворков

Данный обзор описывает существующие популярные робототехнические фреймворки и выделяет их проблемы при использовании на маломощных аппаратных системах [3, 4].

ROS (*Robot Operating System*) — это один из самых известных на сегодняшний день инструментов для разработки программного обеспечения для роботов. Данное решение позволяет легко расширять и переиспользовать различные программные компоненты, хорошо использует ресурсы многоядерных систем, в том числе позволяет взаимодействовать с отдельными компонентами через сетевой протокол, но имеет ряд существенных недостатков. ROS использует вычислительно затратные алгоритмы для межмодульного общения: блокирующая синхронизация, протокол xmlrpc, множественное копирование при передаче сообщений. Перечисленные недостатки существенно снижают общую производительность системы на маломощных робототехнических системах. Использование фреймворка ROS предполагает, что робот имеет высокие вычислительные ресурсы, хорошее беспроводное соединение и не требуется время отклика в реальном времени. Для преодоления этих ограничений началась разработка ROS 2.0. По информации с официального портала фреймворка следует, что система рассчитана преимущественно на высокоскоростное сетевое взаимодействие в группе между различными робототехническими комплексами (РТК). В работе [5] описывается система OROCOS — платформа для программирования робототехнических систем в реальном времени. Используется скриптовый язык LUA. Платформа не предоставляет никаких преимуществ с точки зрения производительности из-за использования скриптового языка. В работе [6] описана модульная платформа OPRoS для разработок робототехнических систем. Компоненты встраиваются как сетевые модули. Активно используется технология разметки XML. Платформа снабжена утилитой для визуального программирования. Нет упора на производительность, вследствие чего плохо применима к реальным системам. В работе [7] описывается платформа Urbi. Авторами разработан собственный язык программирования с учетом специфики разработок для робототехнических систем. Это требует от разработчика изучения дополнительного специфичного языка, и как следствие, отсутствует возможность использовать любые популярные готовые программные библиотеки, что существенно затрудняет разработку. Также в языке параллелизм реализован в нетрадиционном виде, что может

потребовать дополнительных усилий при разработке. В работе [8] описана архитектура платформы для робототехнических систем PX4. Платформа направлена только на работу с микроконтроллерами, а не на работу с полноценной робототехнической системой.

Проектирование системы обмена данными

При проектировании системы используется абстракция — механизм, который является классом-оберткой вокруг потоконебезопасного класса. Задача данного компонента заключается в том, что он инкапсулирует все вызовы к определенному потоконебезопасному классу в функциональный объект и помещает их в потокобезопасную очередь. Предполагается, что система работает в рамках одного адресного пространства и отдельные компоненты могут исполняться в отдельных потоках. Количество обращений к механизмам достаточно высоко и поэтому блокирующий вызов существенно понижает отклик всей системы.

Для описания принципов работы демонстрируется реализация системы рассылки сообщений с использованием шаблона проектирования «издатель-слушатель». Интерфейс системы рассылки сообщений включает в себя четыре функции: отправить сообщение в топик, получить список топиков, добавить слушателя сообщений в топик и удалить слушателя из топика. Механизм дублирует функции системы рассылки. Топик может существовать только если у него есть хотя бы один подписчик. Топик является контейнером для подписчиков и позволяет проверить наличие у него слушателей, добавить нового слушателя, удалить слушателя и разослать сообщение всем слушателям.

В случае с использованием алгоритмов без блокировок можно обеспечить потокобезопасность системы рассылки с применением очереди функциональных объектов, которая позволит упорядочить вызовы из разных потоков в одну последовательность. При использовании очереди без блокировок операции, добавленные в очередь, не должны быть зависимы как от состояния, в котором находится потоконебезопасный класс в момент добавления, так и друг от друга.

В данном прототипе системы существует возможность исполнять параллельно в нескольких потоках задачи из стадии синхронизации и отдельно модули при гарантии, что во время синхронизации модули обрабатываться не будут.

На рис. 1, представлена схема синхронного прототипа ядра системы. Данная диаграмма классов включает только основной набор классов, которые демонстрируют механику взаимодействия между отдельными компонентами системы:

- *abstract_launcher* — интерфейс исполняющего класса. Реализация данного класса должна хранить указатели на модули и очереди.
- *abstract_queue_adapter* — интерфейс для передачи задач синхронизации в определенные очереди.
- *abstract_node* — интерфейс модуля, каждый из которых может быть ответственен за свой конкретный функционал. Модули «не знают» о существовании других модулей, их взаимодействие и исполнение обеспечено ядром системы.
- *abstract_mechanism* — потокобезопасная обертка вокруг класса, который содержит потокобезопасную очередь функциональных объектов, куда помещаются

все вызовы к базовому классу. Очередь механизма регистрируется в экземпляре abstract_launcher.

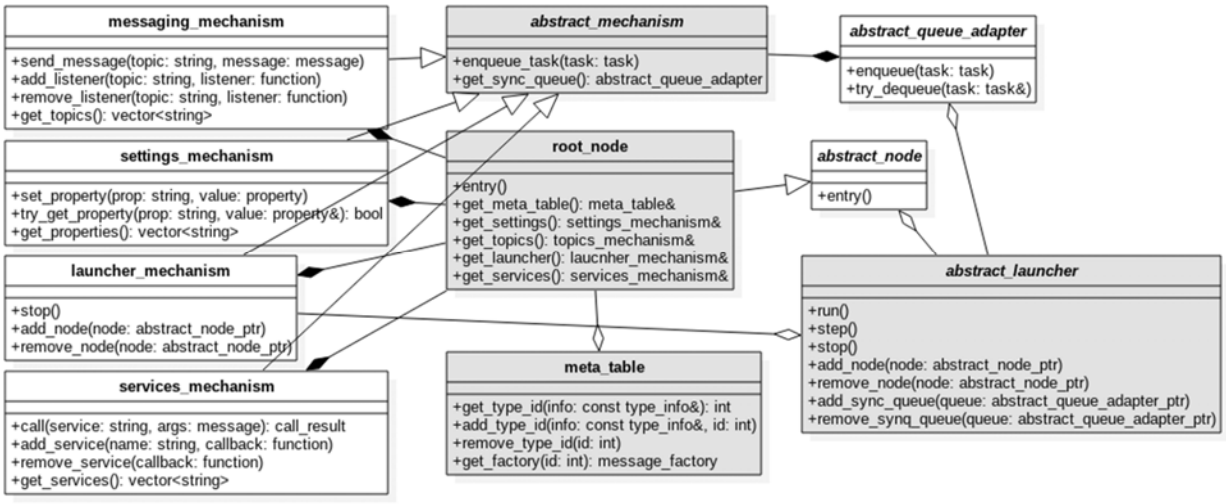

Рис. 1. Диаграмма классов синхронного прототипа

- *launcher_mechanism* — компонент, обеспечивающий взаимодействий с конкретной реализацией abstarct_launcher, позволяет добавлять и удалять модули, запускать и останавливать систему.
- *messaging_mechanism* — компонент, обеспечивающий возможность коммуникации и обмена данными между модулями путем отправки сообщений.
- *services_mechanism* — компонент, обеспечивающий возможность запуска требуемых методов по их имени.
- *meta_table* — класс, который содержит информацию для идентификации сообщений и экземпляр «абстрактной фабрики» сообщения. В данной реализации использовалась библиотека google protocol buffers. Сообщения внутри системы передаются в «сыром» виде для уменьшения дополнительных нагрузок на систему. Сообщения хранятся и передаются внутри системы через умные указатели на экземпляр структуры, которая хранит данные.

При проектировании в дальнейшем было принято решение отказаться от реализации системы сообщений с использованием мета-таблиц и google protocol buffers. Данное решение позволяет использовать контейнеры сообщений без десериализации, но накладывает ограничения на систему для дальнейшего масштабирования. По умолчанию в системе реализован класс abstract_launcher, который поочередно запускает обработку модулей и стадию синхронизации. Такой порядок исполнения позволяет исключить гонку за данные при многопоточном исполнении модулей. В данной реализации сообщение передается в механизм сообщений. Во время стадии синхронизации сообщение передается в очередь модулей, которые подписаны на определённый топик и после обрабатываются во время следующего исполнения

модуля. Из-за данной специфики данный прототип условно назван синхронным. Поскольку в синхронной версии сообщение передается через две очереди был реализован асинхронный прототип фреймворка.

На рис. 2 изображена диаграмма основных классов, используемые в асинхронной версии системы.

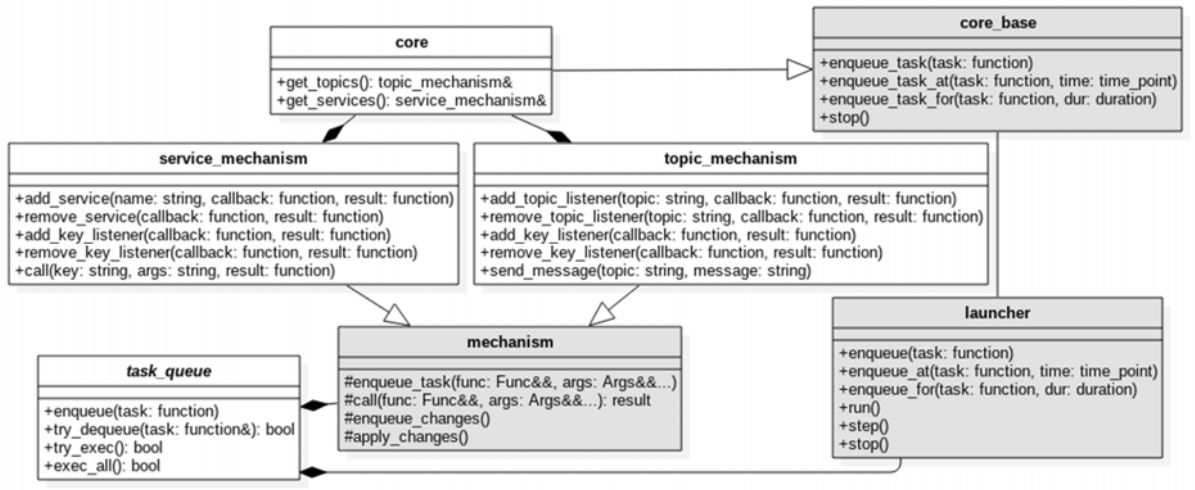

Рис. 2. Диаграмма классов асинхронного прототипа

- *abstract_launcher* — класс определяет порядок и способ исполнения задач. Имеет функции планировщика для выполнения задачи в определенный момент времени или спустя заданный промежуток времени.
- *task_queue* — поскольку в системе потокобезопасные очереди используются исключительно для хранения отложенных для исполнения задач, интерфейс absract_queue_adapter был адаптирован для этих целей.
- *core_base* — базовый класс ядра системы, который предоставляет модулям системы ограниченный набор методов класса abstract_launcher чтобы исключить повторный запуск системы.
- *core* — класс, который расширяет базовый класс ядра и включает в себя набор механизмов, который будет в дальнейшем использоваться модулями.
- *mechanism* — интерфейс механизма, предоставляющий потокобезопасную обертку вокруг потоконебезопасного класса.
- *topic_mechanism* — компонент, реализующий механизм коммуникации между модулями путем рассылки сообщений.
- *service_mechanism* — компонент, реализующий механизм коммуникации между модулями с использованием отложенного вызова метода по имени.

В данной реализации отсутствует класс нода из-за особенностей архитектуры. Каждый модуль можно описать группой методов, которые выполняются в определенном порядке и при определенных условиях. В одном механизме может существовать несколько очередей синхронизации для упорядочивания асинхронных вызовов к механизму. В каждом механизме существует метод добавления в abstract_launcher

задачи на применение изменений. Получение информации из системы так же осуществляется с использованием паттерна «издатель-подписчик». Проблема зацикливания обработки механизмов решается ограничением количества вызовов за одну итерацию обработки входящих событий из очереди. Подобное поведение можно реализовать с использованием библиотек асинхронного программирования, например, Boost Asio [9].

Тестирование производительности

В данном разделе приводятся результаты тестирования производительности межмодульной коммуникации разработанных библиотек в сравнении с аналогами из библиотеки boost. Тестирование производится на двух аппаратных платформах на базе процессоров Intel Atom z530 1.6ГГц и Intel Core i5-6300HQ 2.30ГГц на операционной системе Arch Linux с ядром версии 4.10.13.

В тестах каждый модуль отправляет заданное количество сообщений во все модули сообщений. Данные тесты позволяют проверить влияние на производительность как количества сообщений, так и количества модулей. Тесты рассылки сообщений исполняются в одном потоке.

По результатам из таблиц 1, 2, 3, 4, 5, 6 видно, что количество затраченного времени возрастает линейно с увеличением количества сообщений и модулей и показывают средний прирост производительности в разработанных библиотеках приблизительно в 5–8 раз.

Таблица 1.

Синхронная рассылка сообщений (atom)

Кол-во модулей	Количество сообщений			
	100	1 000	10 000	100 000
10	0,598 мс	4,005 мс	52,678 мс	538,046 мс
100	1,831 мс	14,785 мс	152,465 мс	1553,962 мс
1000	14,130 мс	148,920 мс	1648,494 мс	16764,120 мс

Таблица 2.

Асинхронная рассылка сообщений (atom)

Кол-во модулей	Количество сообщений			
	100	1 000	10 000	100 000
10	0,337 мс	3,295 мс	33,988 мс	339,594 мс
100	0,845 мс	6,006 мс	58,637 мс	582,871 мс
1000	7,508 мс	46,813 мс	444,112 мс	4 589,340 мс

Таблица 3.

Рассылка сообщений через Boost Signals2 (atom)

Кол-во модулей	Количество сообщений			
	100	1 000	10 000	100 000
10	0,698 мс	7,184 мс	72,441 мс	729,576 мс
100	3,392 мс	53,486 мс	515,957 мс	5 898,091 мс
1000	33,071 мс	504,437 мс	6 142,687 мс	65 892,034 мс

Таблица 4.

Синхронная рассылка сообщений (core i5)

Кол-во модулей	Количество сообщений			
	100	1 000	10 000	100 000
10	0,059 мс	0,456 мс	4,621 мс	49,460 мс
100	0,280 мс	1,782 мс	21,069 мс	221,779 мс
1000	3,296 мс	27,850 мс	275,936 мс	2 713,869 мс

Таблица 5.

Асинхронная рассылка сообщений (core i5)

Кол-во модулей	Количество сообщений			
	100	1 000	10 000	100 000
10	0,047 мс	0,437 мс	4,176 мс	45,404 мс
100	0,223 мс	1,664 мс	16,682 мс	170,504 мс
1000	1,842 мс	14,176 мс	146,310 мс	1 461,084 мс

Таблица 6.

Рассылка сообщений через Boost Signals2 (core i5)

Кол-во модулей	Количество сообщений			
	100	1000	10 000	100 000
10	0,094 мс	0,777 мс	7,779 мс	801,600 мс
100	0,578 мс	5,288 мс	56,391 мс	547,934 мс
1000	5,964 мс	56,390 мс	570,350 мс	5 595,921 мс

На данный момент, средняя частота программного взаимодействия между операционной системой и аппаратным обеспечением мобильного робота находится на отметке примерно 8–14 миллисекунд на мобильном роботе Darwin OP [10]. Наибо-

лее вероятно, количество модулей и передаваемых сообщений в практическом использовании подобных робототехнических систем будет порядка 10 модулей и 100 сообщений за одну итерацию, что значительно быстрее, чем скорость программного взаимодействия с аппаратным обеспечением робота.

В таблицах 7 и 8 приведены результаты тестов сравнения производительности очереди задач в RRC и Boost Asio. В данных тестах в очередь добавляется и извлекаются 100 задач из разных потоков. Для core i5 виден существенный прирост в производительности в разработанной библиотеке при увеличении количества потоков [11].

Таблица 7.

Скорость добавления задач в очередь (atom)

Кол-во потоков	1	2	4	8	16	32
RRC	2,197 мс	2,190 мс	2,263 мс	2,482 мс	2,350 мс	4,017 мс
Boost Asio	3,190 мс	3,140 мс	3,160 мс	3,118 мс	3,118 мс	3,103 мс

Таблица 8.

Скорость добавления задач в очередь (core i5)

Кол-во потоков	1	2	4	8	16	32
RRC	0,231 мс	0,167 мс	0,136 мс	0,113 мс	0,055 мс	0,029 мс
Boost Asio	0,712 мс	1,468 мс	1,812 мс	1,952 мс	2,016 мс	2,119 мс

Заключение

Для повышения производительности проектирование модульной системы предполагается в рамках единого адресного пространства на основе многопоточности с минимизацией количества системных вызовов. Для взаимодействия между модулями разработан API ядра для библиотеки с использованием потокобезопасных очередей функторов без блокировок. Для повышения производительности передачи большого количества данных внутри системы сообщения передаются по указателю на буфер данных. Система предполагает хранить данные в сереализированном виде для взаимодействия через сеть и использованием модулей, написанных различных на языках.

В качестве алгоритмов для сериализации и десериализации выбраны реализации Google Protobuf и Flatbuffers. Реализация на Protobuf работает с использованием дополнительной мета-таблицы для передачи внутри системы исходных структур данных, что накладывает существенные ограничения на систему: все сообщения должны быть заранее зарегистрированы в этой метатаблице. Flatbuffers позволяет передавать сериализированные сообщения без существенной потери производительности.

Результаты тестирования показывают прирост производительности системы передачи сообщений в 5–8 раз в сравнении с использованием аналогичных реализаций из библиотеки Boost. Так же наблюдается существенный прирост производительности разработанной библиотеки при увеличении количества потоков на многоядерном процессоре.

Литература

1. Syzov D., Kachan D., Siemens E. Algorithm of handling out-of-order delivery for multithreaded udp-based data transport // 5th International Conference on Applied Innovations in IT (ICAIIT). 2017. pp. 17–23.

2. Zaluzhnyi Y. Serialization and deserialization of complex data structures, and applications in high performance computing: Ph. D. thesis. Technische Universität Dresden. 2016.

3. Гапонов В. С., Дашевский В. П., Бизин М. М. Модернизация программно-аппаратного обеспечения модельных сервоприводов для использования в антропоморфных робототехнических комплексах // Доклады Томского государственного университета систем управления и радиоэлектроники. 2016. Т. 19. № 2. С. 41–50.

4. Павлюк Н. А., Будков В. Ю., Бизин М. М., Ронжин А. Л. Разработка конструкции узла ноги антропоморфного робота Антарес на основе двухмоторного колена // Известия ЮФУ. Технические науки. 2016. № 1 (174). С. 227–239.

5. Klotzbücher M., Soetens P., Bruyninckx H. OROCOS RTT-Lua: an Execution Environment for Building Real-Time Robotic Domain Specific Languages // International Workshop on Dynamic languages for Robotic and Sensors. 2010. pp. 284–289.

6. Jang C., Lee S.-I., Jung S.-W. et al. OPRoS: A New Component-based Robot Software Platform // ETRI journal. 2010. Vol. 32. No. 5. pp. 646–656.

7. Baillie J.-C., De-maille A., Hocquet Q. et al. The Urbi Universal Platform for Robotics // First International Workshop on Standards and Common Platform for Robotics. 2008.

8. Meier L., Honegger D., Pollefeys M. Px4: A Node-based Multithreaded Open Source Robotics Framework for Deeply Embedded Platforms // IEEE International Conference on Robotics and Automation (ICRA). 2015. pp. 6235–6240.

9. Torjo J. Boost. Asio C++ Network Programming. Packt Publishing Ltd. 2013.

10. Ha I., Tamura Y., Asama H. et al. Development of Open Humanoid Platform DARwIn-OP // SICE Annual Conference (SICE). 2011. pp. 2178–2181.

11. Pavluk N., Ivin A., Budkov V., Kodyakov A., Ronzhin A. Mechanical Leg Design of the Anthropomorphic Robot Antares // Lecture Notes in Artificial Intelligence. 2016. pp. 113–123.

References

1. Syzov D., Kachan D., Siemens E. Algorithm of handling out-of-order delivery for multithreaded udp-based data transport // 5th International Conference on Applied Innovations in IT (ICAIIT). 2017. pp. 17–23.

2. Zaluzhnyi Y. Serialization and deserialization of complex data structures, and applications in high performance computing: Ph. D. thesis. Technische Universität Dresden. 2016.

3. Gaponov V., Dashevsky V., Bizin M. Upgrading the Firmware of Model Servos for Use in Anthropomorphic Robotic Systems // Doklady TUSUR. 2016. Vol. 19. No. 2. pp. 41–50.

4. Pavlyuk N., Budkov V., Bizin M., Ronzhin A. Design Engineering of a Leg Joint of the Anthropomorphic Robot Antares based on a Twin-Engine Knee // Izvestiya YUFU. Tekhnicheskie nauki. 2016. No. 1 (174). pp. 227–239.

5. Klotzbücher M., Soetens P., Bruyninckx H. OROCOS RTT-Lua: an Execution Environment for Building Real-Time Robotic Domain Specific Languages // International Workshop on Dynamic languages for Robotic and Sensors. 2010. pp. 284–289.

6. Jang C., Lee S.-I., Jung S.-W. et al. OPRoS: A New Component-based Robot Software Platform // ETRI journal. 2010. Vol. 32. No. 5. pp. 646–656.

7. Baillie J.-C., De-maille A., Hocquet Q. et al. The Urbi Universal Platform for Robotics // First International Workshop on Standards and Common Platform for Robotics. 2008.

8. Meier L., Honegger D., Pollefeys M. Px4: A Node-based Multithreaded Open Source Robotics Framework for Deeply Embedded Platforms // IEEE International Conference on Robotics and Automation (ICRA). 2015. pp. 6235–6240.

9. Torjo J. Boost. Asio C++ Network Programming. Packt Publishing Ltd. 2013.

10. Ha I., Tamura Y., Asama H. et al. Development of Open Humanoid Platform DARwIn-OP // SICE Annual Conference (SICE). 2011. pp. 2178–2181.

11. Pavluk N., Ivin A., Budkov V., Kodyakov A., Ronzhin A. Mechanical Leg Design of the Anthropomorphic Robot Antares // Lecture Notes in Artificial Intelligence. 2016. pp. 113–123.

Ивин Арсений Григорьевич – младший научный сотрудник, СПИИРАН, Санкт-Петербург, 199178, Российская Федерация, arssivka@yandex.ru

Михальченко Даниил Игоревич – младший научный сотрудник, СПИИРАН, Санкт-Петербург, 199178, Российская Федерация, ocnechlahim@mail.ru

Ivin Arseniy – Research Assistant, SPIIRAS, St. Petersburg, 199178, Russian Federation, arssivka@yandex.ru

Mikhalchenko Daniil – Research Assistant, SPIIRAS, St. Petersburg, 199178, Russian Federation, ocnechlahim@mail.ru