UNIFESP

UNIVERSIDADE FEDERAL DE SÃO PAULO

1933

Tiago Yukio Gashu

INTEGRAÇÃO DO SISTEMA OPERACIONAL ROS COM O MÓDULO DE ALTA PERFORMANCE DA PLATAFORMA ROBODECK

São José dos Campos, SP

Tiago Yukio Gashu

INTEGRAÇÃO DO SISTEMA OPERACIONAL ROS COM O MÓDULO DE ALTA PERFORMANCE DA PLATAFORMA ROBODECK

Trabalho de conclusão de curso apresentado ao Instituto de Ciência e Tecnologia – UNIFESP, como parte das atividades para obtenção do título de Bacharel em Ciência da Computação.

Universidade Federal de São Paulo – UNIFESP

Instituto de Ciência de Tecnologia

Bacharelado em Ciência da Computação

Orientador: Prof. Dr. Luiz Eduardo Galvão Martins

São José dos Campos, SP

Junho de 2014

Tiago Yukio Gashu

INTEGRAÇÃO DO SISTEMA OPERACIONAL ROS COM O MÓDULO DE ALTA PERFORMANCE DA PLATAFORMA ROBODECK

Trabalho de conclusão de curso apresentado ao Instituto de Ciência e Tecnologia – UNIFESP, como parte das atividades para obtenção do título de Bacharel em Ciência da Computação.

Trabalho aprovado em 01 de Julho de 2014:

Prof. Dr. Luiz Eduardo Galvão Martins
Orientador

Prof. Dr. Arlindo Flavio da Conceição
Convidado 1

Prof. Dr. Ezequiel Roberto Zorzal
Convidado 2

São José dos Campos, SP
Junho de 2014

Este trabalho é dedicado aos meus avós, aos meus pais e aos meus irmãos.

Agradecimentos

Agradeço primeiramente aos meus avós pelo apoio durante todo o período da graduação, aos meus pais, irmãos e professores. Agradeço à orientação do prof. dr. Luiz Eduardo Galvão Martins, que sanou minhas dúvidas durante o desenvolvimento deste trabalho.

Agradeço aos meus amigos Luis Tacioli Cavaglieri, Daniel Makita, Raíssa Albano Fujioka, Bruno Gabriel dos Ouros, Aryel Fernandes, Victor Ronconi, Luis Filipe Ferreira Fraga, Victor Presumido, Victor Luiz Pereira, Edward Hsiao, Érico Prado, João Gabriel, Robson Rigatto, Thiago Rossener, Gustavo Lopes, Gustavo Rosolen, Rafael Ito, Regis Claus, Eduardo Padua, Raquel Marcondes, Kátia Nakamura, Thiago Gibin, Yuri Wessel Prado, Maria Fernanda Campanholo, Noan dos Santos Ribeiro, Marina Miyuki, Rodrigo Santos, Adriano Quast, Adilson Ivan, Renan Augusto, Luiz Felipe Nardin Barreta, Leonardo Wada, Douglas Muraoka, Gustavo Konishi, Henrique Genoves, Shih Yun Huang, Rafael Moreti e também a todos os amigos e amigas que esqueci de citar nesse agradecimento.

"A força não provém da capacidade física e sim de uma vontade indomável."

Mahatma Gandhi

Resumo

O desenvolvimento de software para robôs não é uma tarefa trivial, pois sua dimensão de código pode ser complexa, já que este abrange o nível de *drivers* até a abstração de raciocínio e além. Neste contexto, um sistema operacional robótico auxilia nesse desenvolvimento de software, oferendo abstração de hardware e outras ferramentas. Por isso, para este trabalho foi escolhido o sistema operacional robótico ROS para realizar sua integração com o Módulo de Alta Performance (MAP) da plataforma RoboDeck. A finalidade desta integração é permitir que aplicativos clientes escritos para o RoboDeck possam se comunicar com o MAP por meio do ROS.

Palavras-chaves: RoboDeck. ROS. Integração. Sistema Operacional Robótico. Robótica Móvel. Middleware.

Abstract

The software development for robots is not a trivial task, because the dimension of code can be daunting, as it covers the driver-level up to the abstract reasoning and beyond. In this context, a robot operating system assists the software development, offering hardware abstraction and other tools. Therefore, for this work the robot operating system ROS was chosen to perform the integration with the High Performance Module (MAP) of the RoboDeck platform. The goal of the integration is to allow client applications, written for the RoboDeck, to be able to comunicate with the MAP through ROS.

Key-words: RoboDeck. ROS. Integration. Robot Operating System. Mobile Robotics. Middleware.

Lista de ilustrações

Figura 1 – Plataforma RoboDeck 25
Figura 2 – Arquitetura geral do software do RoboDeck. Na figura as setas indicam as dependências entre os módulos do robô. 26
Figura 3 – Módulo de Controle de Sessão e seus submódulos 28
Figura 4 – Módulo de Controle Robótico e seus submódulos 29
Figura 5 – Módulo de Alta Performance 31
Figura 6 – Diagrama de classes do Módulo de Alta Performance 31
Figura 7 – Diagrama da estrutura do RoboticsSDK 33
Figura 8 – publish subscribe entre nós por meio de um tópico 38
Figura 9 – diagrama de comunicação do *middleware*. 44
Figura 10 – estrutura de um pacote ROSJava. 45
Figura 11 – estrutura do pacote do *middleware*. 45
Figura 12 – estrutura do pacote de mensagens. 46
Figura 13 – diagrama de classes do SDK. 47
Figura 14 – diagrama de classes do *middleware*. 49
Figura 15 – diagrama de classes do *middleware*. 50
Figura 16 – diagrama de sequência do serviço de conexão. 52
Figura 17 – diagrama de sequência do serviço de movimentação. 52
Figura 18 – aplicação GUI de teste. 53
Figura 19 – arquitetura para simular chamadas síncronas em Java. 56

Lista de tabelas

Tabela 1 – Tabela de Serviços do *middleware* 51

Lista de abreviaturas e siglas

API Application Programming Interface

GUI Graphical User Interface

HAL Hardware Abstract Layer

MAP Módulo de Alta Performance

MCC Módulo de Controle de Comunicação

MCR Módulo de Controle Robótico

MCS Módulo de Controle de Sessão

ROS Robot Operating System

SDK Software Development Kit

UART Universal Asynchronous ReceiverTransmitter

Sumário

1 Introdução . . . 21
1.1 Contextualização e Motivação . . . 21
1.2 Objetivos . . . 22
1.2.1 Objetivo Geral . . . 22
1.2.2 Objetivos Específicos . . . 22
1.3 Procedimentos Metodológicos . . . 22
1.4 Organização do Texto . . . 23

2 RoboDeck . . . 25
2.1 Visão geral do RoboDeck . . . 25
2.2 Arquitetura Geral de Software do RoboDeck . . . 26
2.3 Módulo de Controle de Sessão (MCS) . . . 27
2.4 Módulo de Controle Robótico (MCR) . . . 28
2.5 Módulo de Controle de Comunicação (MCC) . . . 30
2.6 Módulo de Alta Performance (MAP) . . . 30
2.6.1 Estrutura do núcleo do MAP . . . 30
2.6.2 Estrutura dos Aplicativos Robóticos . . . 33
2.6.3 Métodos do MAP . . . 33

3 ROS (Robot Operating System) . . . 35
3.1 Introdução . . . 35
3.2 Conceitos do ROS . . . 36
3.2.1 Nível de Sistema de Arquivos . . . 36
3.2.2 Nível de Grafo de Computação . . . 37
3.2.2.1 Nós . . . 37
3.2.2.2 Mestre . . . 37
3.2.2.3 Servidor de Parâmetro . . . 37
3.2.2.4 Mensagens . . . 37
3.2.2.5 Tópicos . . . 38
3.2.2.6 Serviços . . . 38
3.2.2.7 Bolsas . . . 38
3.2.3 Descrição da interação entre os componentes do ROS . . . 39
3.2.4 Nível de Comunidade . . . 40
3.3 Nomes . . . 40
3.3.1 Graph Resource Names . . . 40
3.3.2 Package Resource Names . . . 41

3.3.3 Resolução de nomes 41

4 Desenvolvimento 43

4.1 Ideias iniciais, empecilhos e comentários 43
4.2 Visão Geral do *Middleware* 43
4.3 ROSJava 44
4.4 SDK em Java 46
4.5 Arquitetura do SDK 47
4.6 Arquitetura do *middleware* 48
4.6.1 Classes do pacote *org.ros.node* 48
4.6.2 Classes do pacote *com.robodeck.middleware* 49
4.7 Serviços do *middleware* 50
4.8 Diagramas de Sequência de Funcionamento 50

5 Analise e Discussão de Resultados 53

5.1 Teste do funcionamento do SDK da XBOT 53
5.2 Teste do funcionamento do SDK em Java 53
5.3 Análise do *middleware* 54
5.4 *Framework* ROS 54
5.5 Teste do funcionamento do *middleware* 55
5.5.1 Programa cliente em Java 55

6 Conclusão 57

6.1 Contribuições 57
6.2 Trabalhos Futuros 57

Referências 59

Apêndices 61

APÊNDICE A Código cliente cm C++ para a movimentação do robô 63

APÊNDICE B Código cliente em Python para a movimentação do robô . . 65

APÊNDICE C Código cliente em Java para a movimentação do robô . . . 67

1 Introdução

Este Capítulo tem o intuito de apresentar a contextualização e motivação deste trabalho, além de evidenciar seus objetivos, procedimentos metodológicos e sua organização.

1.1 Contextualização e Motivação

Segundo Wolf et al. (2009), o estudo da robótica móvel apresentou notável evolução nas duas últimas décadas. Além disso, a aplicação de robótica em diversas áreas da nossa sociedade demonstra o quão promissor é o futuro dessa área. Pode-se exemplificar o seu uso em aplicações domésticas (ex. aspiradores de pó robóticos), industriais (ex. transporte automatizado), militares, de segurança e defesa civis e militares (WOLF et al., 2009).

Quigley et al. (2009) afirmam que a produção de software para robôs é uma tarefa difícil, pois diferentes tipos de robôs podem ter variações de hardware de um para outro, tornando não trivial a reutilização de código. Além disso, a dimensão do código pode ser complexa, já que este contém uma pilha partindo do nível de drivers e subindo até a percepção, raciocínio abstrato e além. As arquiteturas de software para robótica devem dar suporte à integração de software em larga escala, pois a amplitude de expertise necessária ultrapassa a habilidade individual de qualquer pesquisador (QUIGLEY et al., 2009). Neste contexto, um sistema operacional para robôs auxilia não apenas na integração de software como também no desenvolvimento de um sistema robótico.

Para este trabalho foi escolhido o ROS (Robot Operating System) como sistema operacional robótico para ser utilizado conjuntamente com a plataforma robótico RoboDeck. O ROS provê serviços de sistema operacional, como: abstração de hardware, controle de dispositivos de baixo nível, comunicação entre processos etc. A ideia de se usar um sistema operacional robótico é de oferecer funcionalidade padrão para se abstrair a camada de hardware, assim como os sistemas operacionais de computadores (KERR; NICKELS, 2012). Além de ser um sistema operacional robótico, o ROS também é considerado um *framework* e um *middleware* robóticos.

O RoboDeck foi desenvolvido com o intuito de promover o desenvolvimento educacional na área de pesquisa. Ele é constituído de uma plataforma robótica (o robô), um ambiente de programação em C/C++ com um software para teste e SDK (XBOT, 2011).

O Módulo de Alta Performance (MAP) do RoboDeck é um software projetado para ser executado em uma placa microprocessada qualquer. O MAP permite que sejam desenvolvidos aplicativos robóticos capazes de atuar como controladores do robô. Estes aplicativos podem ser utilizados também para aumentar a gama de comandos robóticos (MUñOZ, 2010).

Este trabalho tem o objetivo de realizar a integração entre a plataforma robótica Robo-

Deck e o *framework* robótico ROS. O framework ROS é utilizado para se criar um *middleware* para o MAP do RoboDeck. Este *middleware* faz a interface de comunicação entre os aplicativos robóticos e o MAP. A integração por meio deste *middleware* também permite o desacoplamento entre a implementação dos aplicativos e a plataforma robótica, no caso, o MAP do RoboDeck. Além disso, segundo Elkady e Sobh (2012) as principais vantagens de um *middleware* robotico são: modularidade de software, abstração da arquitetura de hardware, independência de plataforma e portabilidade. Assim, a plataforma RoboDeck pode ser controlada por meio do *middleware* utilizando as bibliotecas clientes do ROS.

1.2 Objetivos

Esta seção apresenta os objetivos deste trabalho.

1.2.1 Objetivo Geral

O objetivo deste trabalho de pesquisa é criar um *middleware* robótico para o RoboDeck com o intuito de integrar a plataforma robótica com o *framework* ROS. Após a implementação do *middleware*, foi realizado um estudo de caso constituído de uma coleção de aplicativos simples, com o intuito de se explorar os comandos do robô através da nova interface de comunicação. Os aplicativos utilizam as bibliotecas do ROS para se comunicar com o *middleware*.

1.2.2 Objetivos Específicos

Este trabalho têm quatro objetivos especificos, sendo eles:

- Compreender o framework ROS, entendendo sua arquitetura e seus principais componentes.
- Entender a arquitetura e os componentes da plataforma RoboDeck, dando ênfase ao MAP.
- Pesquisar trabalhos correlatos, estudar formas de se implementar a interface para o MAP e realizar um comparativo com trabalhos semelhantes pesquisados.
- Propor um middleware robótico que funcione como uma interface para o MAP.

1.3 Procedimentos Metodológicos

Na primeira parte deste trabalho foram realizadas algumas atividades principais: pesquisa e leitura de referências bibliográficas e estudo dos componentes, sendo estes a plataforma RoboDeck e o *framework* ROS.

Primeiramente, foram pesquisadas referências bibliográficas. Foram utilizados os sites www.acm.org e www.ieee.org para se pesquisar artigos relacionados ao tema. Também, foram utilizadas referências sugeridas pelo orientador. Depois, foi realizada a leitura e foram filtradas as referências relacionadas ao tema deste trabalho.

Em seguida, foi realizado um estudo aprofundado do ROS e da plataforma RoboDeck. Tanto para o ROS quanto para o RoboDeck, foram utilizadas principalmente suas respectivas documentações, além de conteúdos encontrados em artigos e tutoriais. Em relação ao RoboDeck, foi estudada sua arquitetura de componentes de software dando ênfase ao módulo de alta performance.

Na segunda parte, foram realizados o desenvolvimento, a análise e o teste do *middleware*. O desenvolvimento foi dividido em duas partes: a implementação do SDK em Java e a implementação do *middleware* (utilizando tanto o SDK quanto o ROS). A análise foi feita a partir do estudo do *framework* ROS e dos resultados obtidos nos testes do *middleware*.

1.4 Organização do Texto

O restante deste trabalho está organizado da seguinte forma: os Capítulos 2 e 3 abordam a fundamentação teórica deste trabalho, sendo que o Capítulo 2 apresenta os conceitos relacionados à plataforma RoboDeck, enquanto o Capítulo 3 discorre a respeito dos conceitos associados ao *framework* ROS. O Capítulo 4 expõe o desenvolvimento do *middleware* robótico. O capítulo 5 faz uma análise a partir dos resultados obtidos. Por fim, o Capítulo 6 evidencia as contribuições deste trabalho e expõe ideias para a sua continuação.

2 RoboDeck

Este capítulo apresenta primeiramente uma visão geral da plataforma robótica e, depois, aborda a arquitetura de software do RoboDeck. Foram utilizados como referências o manual de software (XBOT, 2011) e o documento do projeto de software do RoboDeck (MUñOZ, 2010).

2.1 Visão geral do RoboDeck

O RoboDeck foi concebido para ser um robô modular e que permitisse ao usuário expandir, além dos comandos, o seu próprio físico. Essa expansão pode ser feita a partir da adição de garras, braços, sensores ou outros periféricos robóticos (XBOT, 2011).

Figura 1 – Plataforma RoboDeck. Figura extraida de http://www.xbot.com.br/educacional/robodeck

Partindo deste conceito de modularidade, o hardware do RoboDeck pode ser dividido em duas partes principais: o hardware essencial e o hardware opcional. O hardware essencial é composto de dois microcontroladores que se comunicam por uma UART. Os softwares es-

critos para o hardware essencial são independentes de sistemas operacionais e são escritos em linguagem C (XBOT, 2011).

Em contrapartida, o hardware opcional é composto por uma placa qualquer que seja capaz de executar um sistema operacional Linux. Os softwares que são executados nessa placa são escritos em linguagem C++ e têm o sistema operacional Linux como base. O hardware opcional tem o objetivo de agregar autonomia e comunicação banda larga ao robô. O software que funciona neste hardware é chamado de Módulo de Alta Performance (MAP). Para atribuir autonomia ao robô, o MAP permite que aplicativos controlem o hardware essencial do robô por meio de comandos enviados diretamente ao Módulo de Controle de Sessão (MCS) (MUñOZ, 2010).

2.2 Arquitetura Geral de Software do RoboDeck

O software do RoboDeck está dividido em níveis. No nível mais baixo, estão os softwares dos microcontroladores escritos em C/C++. Um nível acima se encontra o conjunto de instruções do módulo de alta performance (MAP), que possui a capacidade de interagir com sistemas autônomos ou controladores externos. Além destes, existem os SDKs, sendo que um escrito em C#/CLI e outro em Java ME. Os dois podem ser utilizados para o desenvolvimento de aplicações de controle do robô através de periféricos (XBOT, 2011).

A Figura 2 a seguir ilustra a arquitetura geral do RoboDeck.

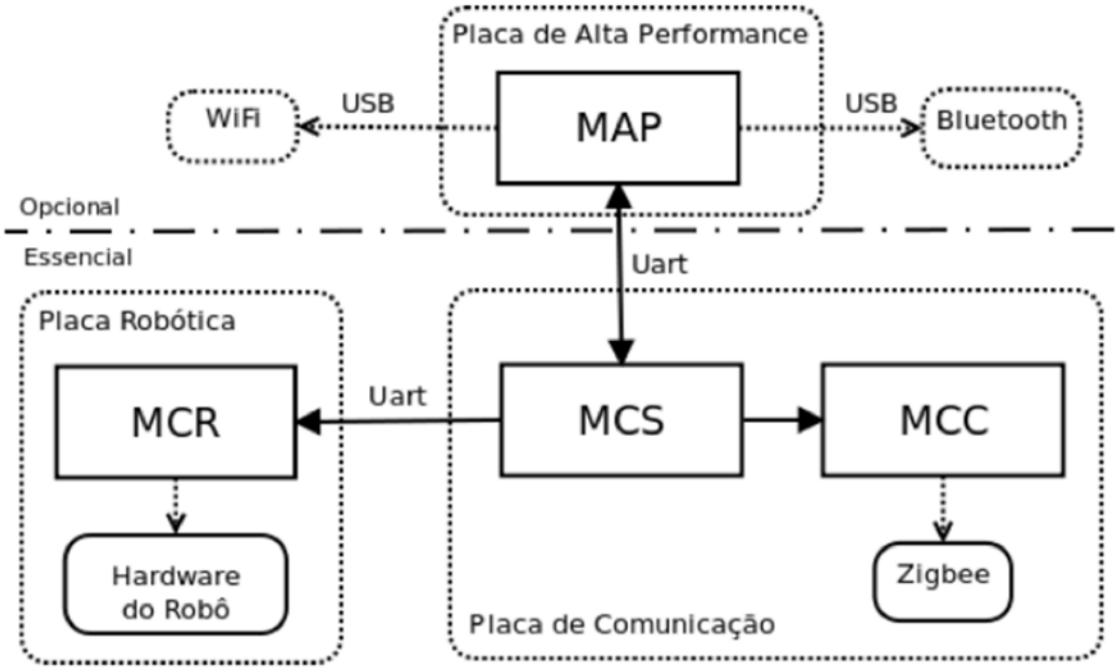

Figura 2 – Arquitetura geral do software do RoboDeck. Na figura as setas indicam as dependências entre os módulos do robô. Figura extraída de (MUñOZ, 2010).

O primeiro microcontrolador da parte essencial é responsável por gerenciar o hardware relacionado aos seus sensores e atuadores. Foi implementado para este microcontrolar o Módulo de Controle Robótico (MCR). Este módulo age como um robô escravo que executa sequencialmente os comandos que chegam pela UART (XBOT, 2011).

Já o segundo microcontrolador é responsável por gerenciar a comunicação de controle do robô e também por manter sua identidade. A comunicação é implementada através do protocolo Zigbee[1] pelo Módulo de Controle de Comunicação (MCC). Este módulo implementa o conceito de sessão aos comandos provenientes de um programa que esteja controlando o robô, para que ele não responda a dois programas distintos simultaneamente. Ele é responsável também pela autenticação dos aplicativos, além de repassar para o MAP os comandos provenientes do MCS. (XBOT, 2011).

O MAP foi implementado no hardware opcional e possui a capacidade de enviar comandos ao MCS. Desta forma, ele permite que qualquer aplicativo que seja executado no MAP manipule o hardware essencial do robô (XBOT, 2011).

As próximas seções abordam os módulos de software RoboDeck. A referência utilizada em todas as próximas seções deste capítulo foi a documentação escrita por Muñoz (2010).

2.3 Módulo de Controle de Sessão (MCS)

O MCS representa a identidade do robô e é responsável por armazenar informações do robô como: identificador único, nome, versão do robô, versão do protocolo de comunicação etc. Ele também decide quem e quando as sessões podem ser abertas ou fechadas.

Ele realiza a autenticação dos aplicativos (controladores), ou seja, os comandos provenientes de um aplicativo só serão aceitos caso este esteja dentro de uma sessão de controle estabelecida entre o aplicativo e o robô.

Todo comando que chegue ao robô é encaminhado primeiramente para o MCS, independente de sua interface de origem. O MCS pode receber comandos provenientes do MAP ou do MCC (*WiFi* ou *Bluetooth* do MAP ou *Zigbee* do MCC) e ele que decide qual parte do robô irá executar os comandos.

A Figura 3 representa o MCS e seus submódulos. O submódulo IComm é, na verdade, uma biblioteca utilizada para realizar a comunicação serial entre o MCS e os módulos MAP e MCR. O submódulo Main contém o programa principal. Além destes dois, os seguintes submódulos compõem o MCS: TaskManager, SessionManager e Authenticator.

- **TaskManager:** tem a função de gerenciar a execução dos comandos. Ele escolhe o módulo que se encarregará de executar o comando, o encaminha, espera a resposta e a retorna

[1] conjunto de especificações para a comunicação sem-fio entre dispositivos eletrônicos

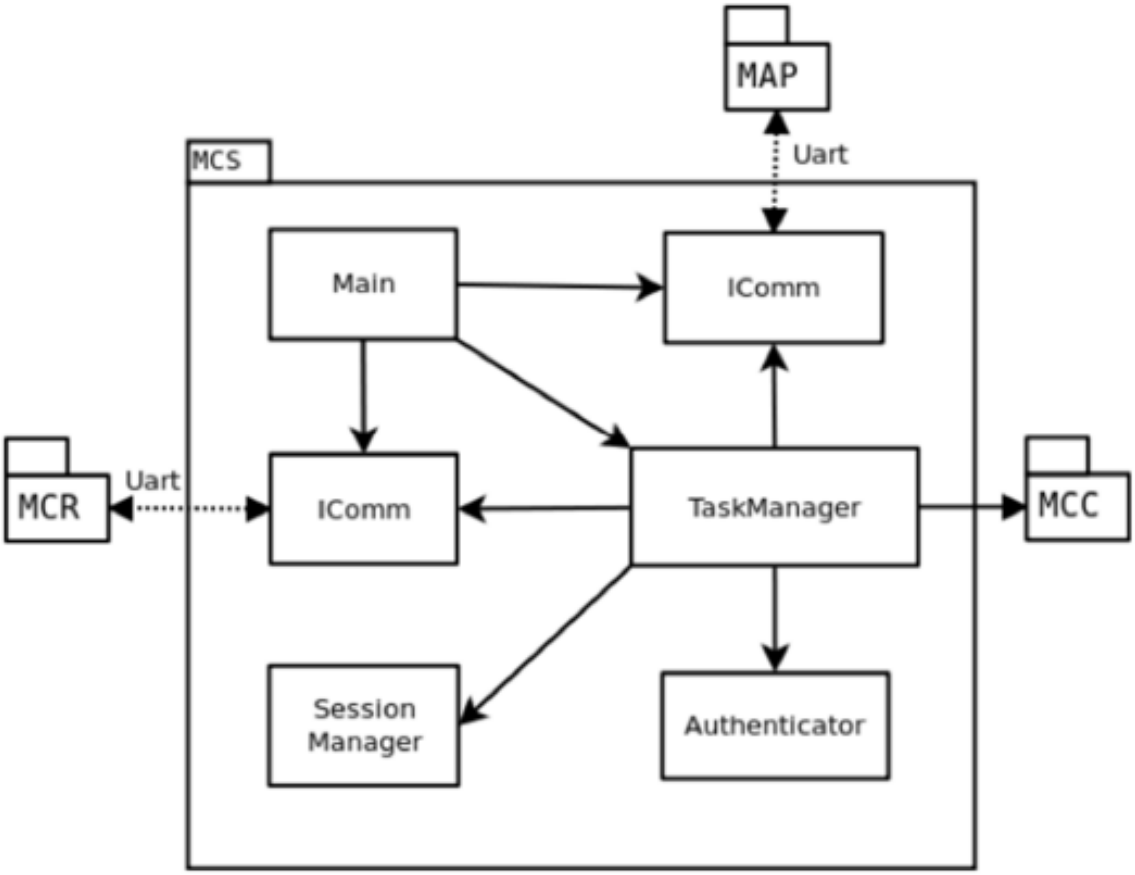

Figura 3 – Módulo de Controle de Sessão e seus submódulos. As setas cheias indicam as dependências entre os submódulos. As setas pontilhadas representam o fluxo de dados. Figura extraída de (MUñOZ, 2010).

ao controlador solicitante.

- **SessionManager:** O submódulo SessionManager é responsável por armazenar as informações da sessão corrente, assim como estabelecer as prioridades dos controladores para abrir uma sessão.

- **Authenticator:** Este submódulo é responsável pela autenticação dos controladores e por armazenar e fornecer a identidade do robô.

2.4 Módulo de Controle Robótico (MCR)

O Módulo de Controle Robótico (Figura 4) é responsável pelo controle do hardware do robô. Por meio da interface serial (UART), ele recebe pacotes de mensagens com comandos robóticos relacionados ao hardware. Este módulo executa os comandos, quando possível, e retorna ao MCS pacotes de mensagens, que representam as respostas aos comandos.

O MCR executa três categorias de comandos: comandos de baixo nível, de alto nível e de slots. Os comandos de baixo nível são diretamente relacionados ao hardware do robô (ex.

acionamento individual dos motores ou leitura direta dos sensores).

Os comandos de alto nível são construídos a partir da composição de comandos de baixo nível. Eles são disponibilizados para facilitar tarefas mais complexas. Por exemplo, para se fazer uma curva com o robô utiliza-se os quatro servomotores do robô. Assim, o comando de alto nível utilizado para virar o robô como um todo é construído por meio de comandos de baixo nível para virar cada um dos 4 servomotores de forma sincronizada.

Os comandos de slots são utilizados para realizar a comunicação com periféricos que podem estar conectados ao robô.

Este módulo (MCR) é dividido em duas camadas: a camada de controle e a camada de abstração do hardware, também chamada de Hardware Abstract Layer (HAL).

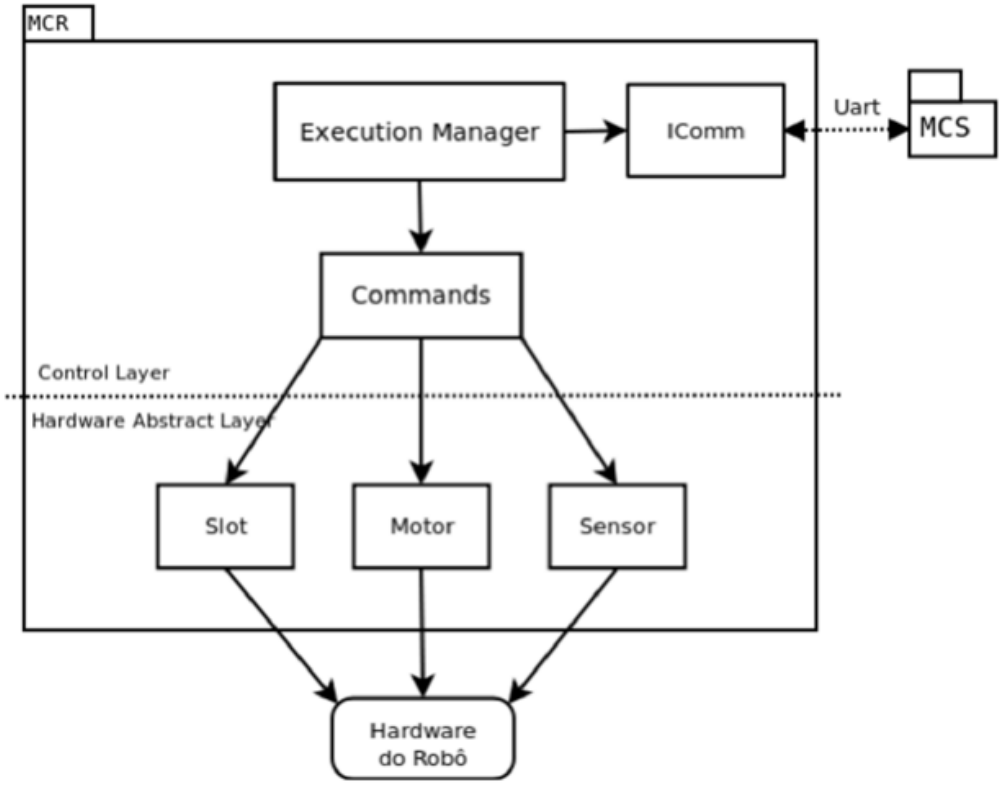

Figura 4 – Módulo de Controle Robótico e seus submódulos. As setas cheias indicam as dependências e as setas tracejadas indicam fluxo de dados. Figura extraída de (MUñOZ, 2010).

A camada de controle possui diversas funções. Sendo elas: receber pacotes que contêm comandos, 'deserializar' os comandos e seus parâmetros, gerenciar a execução dos comandos, serializar suas respostas, transformá-las em pacotes e enviá-las pela interface serial.

A camada de abstração do hardware (HAL) conhece os periféricos do robô. Ela também sabe os pinos e ports pelos quais os periféricos são acessados, além dos seus modelos e comandos de controle.

Os seguintes submódulos fazem parte do MCR: Execution Manager, Commands, Slot, Motor e Sensor.

- **Execution Manager:** este submódulo que recebe os pacotes provenientes da interface serial. Ele verifica se os comandos recebidos são válidos e os executa.
- **Command:** este submódulo implementa todos os comandos aceitos pelo MCR. Ele utiliza as funções implementadas na camada HAL.
- **Slot:** implementa as funções para a comunicação com os slots do robô.
- **Motor:** implementa todas as funções associadas ao controle dos motores do robô.
- **Sensor:** implementa as funções de leitura e configuração dos sensores do robô.

2.5 Módulo de Controle de Comunicação (MCC)

O Módulo de Controle de Comunicação é responsável pela comunicação de controle do robô e é implementado com base no protocolo para comunicação sem fio chamado *Zigbee*.

2.6 Módulo de Alta Performance (MAP)

O Módulo de Alta Performance é um software projetado para poder rodar em qualquer placa microprocessada que seja capaz de executar o sistema operacional Linux e que possua uma interface serial UART e duas interfaces USBs. Esta generalidade do MAP permite que o robô seja atualizado conforme a tecnologia de circuitos integrados se desenvolva e placas melhores possam utilizar o MAP. Os objetivos principais da placa de alta performance são: tornar o robô autônomo e permitir comunicação banda larga entre o robô e o meio externo.

O MAP permite que sejam feitos aplicativos robóticos que atuem como controladores do robô por meio do envio de comandos ao MCS. Estes aplicativos também podem ser usados para expandir os comandos robóticos.

Este módulo é executado como um *daemon* na placa de alta performance, sendo incumbido de centralizar as comunicações entre os controladores, o MCS e os aplicativos robóticos. A Figura 5 mostra a relação entre o MAP e outros componentes.

2.6.1 Estrutura do núcleo do MAP

A estrutura do MAP consiste de um núcleo e componentes (chamados de aplicativos robóticos) que são dinamicamente ligados a este núcleo.

O diagrama da Figura 6 mostra as principais classes que compõem o MAP.

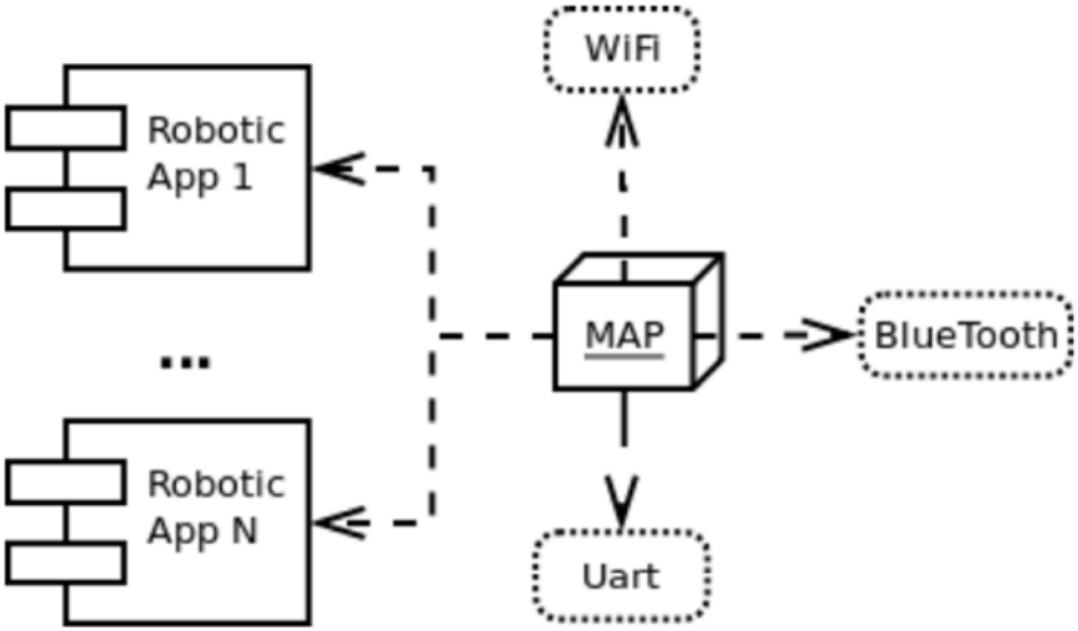

Figura 5 – Módulo de Alta Performance. Figura extraida de (MUñOZ, 2010).

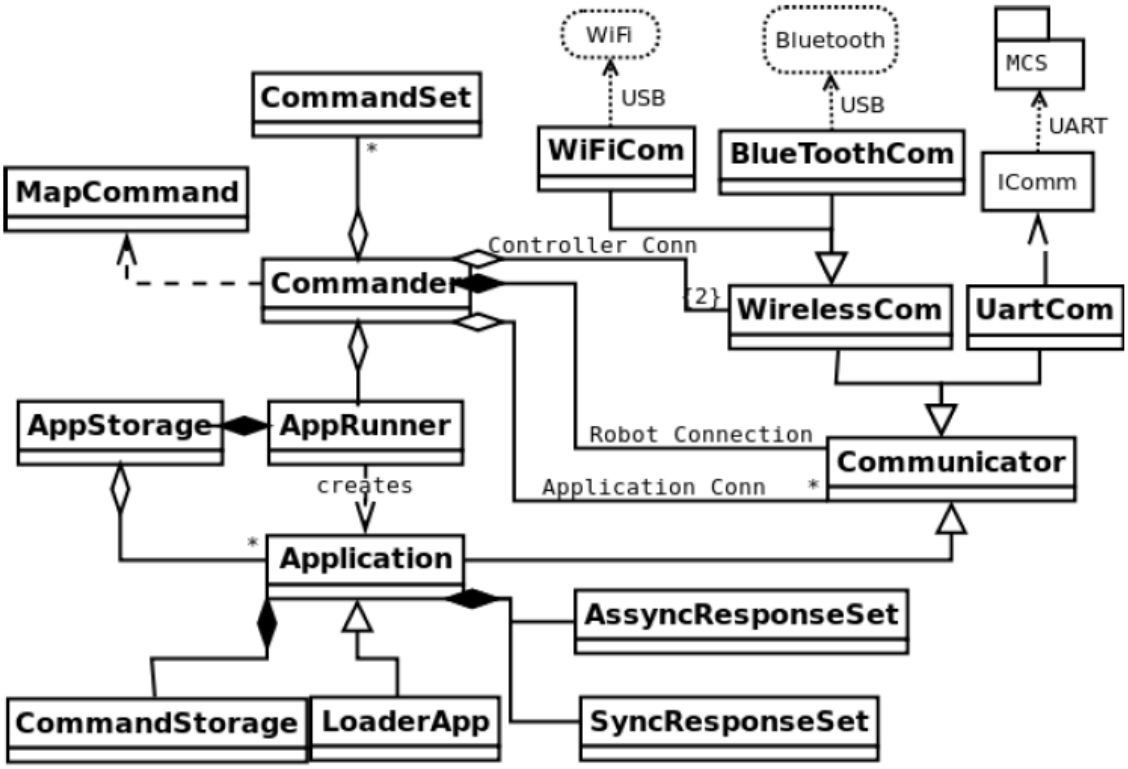

Figura 6 – Diagrama de classes do Módulo de Alta Performance. Figura extraída de (MUñOZ, 2010).

A seguir será feita uma descrição breve de cada classe que compõe o MAP.

- **Application:** classe base para todos os aplicativos robóticos. Ela implementa os mecanis-

mos de carregamento dinâmico dos aplicativos e os de comunicação com o controlador. Esta classe fornece também os métodos para que os aplicativos possam enviar os comandos para o MCS. Além disso, ela define a interface a ser implementada pelos aplicativos. Esta interface permite que o sistema possa recuperar informações a respeito da especificação dos comandos que o aplicativo implementa.

- **AppRunner:** responsável por iniciar, manter e finalizar os aplicativos robóticos. Esta classe também realiza a comunicação entre o sistema e os aplicativos, fornecendo uma interface de comunicação. Por meio desta interface o sistema consegue obter informações a respeito dos aplicativos, enviar comandos e receber suas respectivas respostas.
- **AppStorage:** armazena todas as informações dos aplicativos robóticos do sistema. Por meio desta classe podemos saber os aplicativos que estão sendo executados.
- **AssyncResponseSet:** armazena as informações dos comandos assíncronos sendo executados por um aplicativo robótico.
- **BlueToothCom:** herda da classe Wireless e lida com características específicas da comunicação via *Bluetooth* para se comunicar com controladores por meio do protocolo *Bluetooth*.
- **Commander:** esta classe implementa a coordenação geral do MAP. Ela é responsável por ler os pacotes que chegam pelas interfaces de comunicação, processar e tomar as devidas ações. Em suma, esta classe é responsável por executar os comandos estendidos do robô.
- **CommandSet:** armazena informações sobre os comandos que foram enviados ao robô.
- **CommandStorage:** armazena os estados dos comandos estendidos sendo executados de cada aplicativo e, também, é responsável pelos logs das execuções dos comandos.
- **LoaderApp:** esta classe herda de Application e é responsável por carregar os aplicativos robóticos.
- **MapCommand:** implementa todos os comandos não estendidos executados pelo próprio MAP.
- **SyncResponseSet:** armazena as informações dos comandos síncronos executados por um aplicativo robótico.
- **Communicator:** classe base para todas as classes de comunicação.
- **UartCom:** esta classe é um especialização da classe Communicator. É utilizada para realizar comunicação via interface serial UART.
- **WifiCom:** é uma especialização da classe Wireless e é responsável pela comunicação via protocolo WiFi.

- **WirelessCom:** é uma especialização da classe Communicator para redes sem fio.

2.6.2 Estrutura dos Aplicativos Robóticos

Para se implementar um aplicativo robótico para o MAP, deve-se utilizar os componentes e classes que compõem o RoboticsSDK, que é o software básico no qual os desenvolvedores devem se apoiar para desenvolver seus aplicativos.

Os aplicativos podem seguir qualquer arquitetura que seu desenvolvedor queira. Eles devem apenas ter a classe Application como base para que possam ser integrados ao MAP. Além disso, cada aplicativo robótico pode implementar apenas um comando robótico. Em contrapartida, este comando pode utilizar a quantidade de parâmetros que o desenvolvedor desejar.

A figura 7 mostra a estrutura do RoboticsSDK.

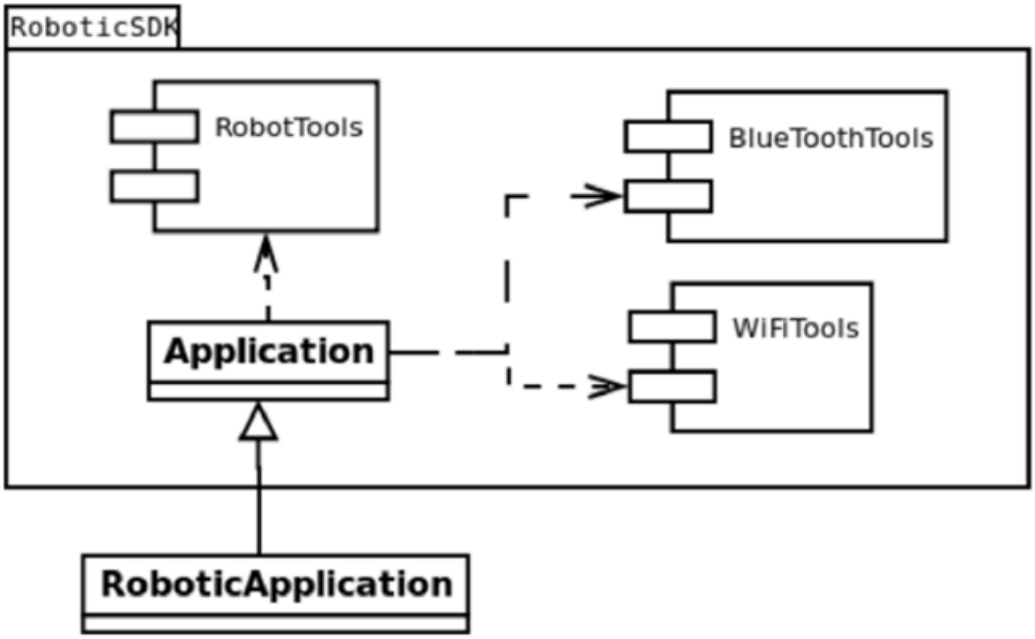

Figura 7 – Diagrama da estrutura do RoboticsSDK. Figura extraída de (MUñOZ, 2010).

Os componentes BlueToothTools e WiFiTools oferecem um conjunto de ferramentas para auxiliar os aplicativos na comunicação com agentes externos. Já o RobotTools auxilia na comunicação dos aplicativos com o MCS.

Como é mostrado na Figura 7, os aplicativos podem utilizar os métodos fornecidos pela classe Application para comunicar-se com os controladores via *WiFi* ou *Bluetooth*, além de também poderem se comunicar com o MCS do RoboDeck.

2.6.3 Métodos do MAP

O MAP fornece três categorias de métodos aos aplicativos robóticos:

- Para responder ao controlador;
- Para enviar comandos ao robô;
- Outros métodos (úteis ou necessários).

1. **Para responder ao controlador**

   Esta categoria contempla os métodos para a comunicação com o controlador e são restritos ao envio de respostas, que são enviadas ao controlador que originou o comando.

   No decorrer da execução de um comando estendido solicitado por um controlador, o aplicativo pode responder uma ou mais vezes ao controlador.

2. **Para enviar comandos ao robô**

   Nesta categoria estão os métodos que enviam comandos ao robô de forma síncrona ou assíncrona. Nos comandos assíncronos, quando a resposta ao comando chega, um método virtual da classe Application é chamado e envia a resposta ao aplicativo. No caso dos comandos síncronos, a thread que enviou o comando fica bloqueada até que a resposta chegue.

   Os métodos de abertura e fechamento de sessão com o MCS também se encontram nesta categoria.

3. **Outros Métodos fornecidos pelo MAP**

   Nesta última categoria se encontram os métodos utilitários ou necessários para a implementação dos aplicativos.

3 ROS (Robot Operating System)

Este capítulo apresenta os conceitos relacionados ao sistema operacional robótico ROS. A referência principal utilizada neste capítulo foi a documentação do próprio ROS, que se encontra disponível em: (ROS..., 2013a) e (ROS..., 2013b).

3.1 Introdução

O ROS é um meta sistema operacional *open source*, um *middleware* robótico e um *framework*. Ele provê os serviços de um sistema operacional, desde a abstração da camada de hardware, controle de dispositivos de baixo nível até comunicação entre processos e gerenciamento de pacotes. Além disso, ele também provê bibliotecas e ferramentas para se escrever, obter, realizar *build* ou executar código por meio de vários computadores (ROS..., 2013a). No entanto, vale ressaltar que o ROS não é um sistema operacional, pois ele não pode ser instalado diretamente em um equipamento robótico, já que ele trabalha em cima de um sistema operacional (BOHREN, 2010).

O ROS em tempo de execução é uma rede *peer-to-peer* de processos (também chamados de Nós) que são fracamente acoplados utilizando a infraestrutura de comunicação do ROS. Essa comunicação pode ser síncrona ou assíncrona, desde comunicação RPC síncrona a streaming de dados em barramentos e armazenamento de dados em um servidor de parâmetro. Além disso, O ROS é também considerado um *framework* que permite executáveis serem projetados individualmente (ROS..., 2013a).

O ROS também permite a colaboração ser distribuída por meio do suporte aos repositórios de código. Este modelo permite decisões independentes em relação ao desenvolvimento e à implementação (ROS..., 2013a).

O principal objetivo deste *framework* robótico é de dar suporte ao reuso de código em pesquisas e desenvolvimento robóticos. Além do reuso, o ROS possui outros objetivos: facilitar integração, ser *tools-based*, ser independente de linguagem, facilitar a realização de testes e ser escalável (QUIGLEY et al., 2009). Cada um destes objetivos serão explicados a seguir (ROS..., 2013a) (QUIGLEY et al., 2009):

- Facilitar integração: o ROS facilita a integração com outros *frameworks*. Ele já foi integrado com o OpenRave, Orocos e Player.
- *Tools-based*: segundo Quigley et al. (2009), os desenvolvedores do ROS optaram por projetar o ROS para ter um arquitetura de *microkernel*, ou seja, possuir diversas ferramentas pequenas para construir e rodar os vários componentes do ROS.

- Independente de linguagem: as especificações do ROS estão apenas na camada de passagem de mensagem. Além disso, o ROS usa uma linguagem de definição de interface (IDL) para descrever as mensagens que são passadas entre os módulos.
- Facilitar a realização de testes: O ROS possui um *framework* de teste de unidade e integração chamado *rostest*.
- Ser escalável: o ROS oferece escalabilidade para sistemas de grande porte.

Em suma, o intuito do ROS é permitir que pesquisadores possam realizar o desenvolvimento rápido de sistemas robóticos sem precisar "reinventar a roda" através da utilização das ferramentas do *framework* (QUIGLEY et al., 2009).

As próximas seções utilizam como referência a página da documentação em relação aos conceitos do ROS (ROS..., 2013b).

3.2 Conceitos do ROS

O ROS possui três níveis de conceitos: o nível do sistema de arquivos, o nível do Grafo de Computação e o nível da comunidade. Além desses conceitos, o ROS também define dois tipos de nomes: Nomes de Pacotes de Recursos (*Package Resource Names*) e Nomes de Recursos de Grafo (*Graph Resource Names*).

3.2.1 Nível de Sistema de Arquivos

Serão citados e descritos os conceitos relacionados ao nível de sistema de arquivos. Tais conceitos são:

- **Pacotes:** os pacotes são a unidade principal para a organização de software no ROS. Um pacote pode conter processos ROS em tempo de execução (nodes), biblioteca dependente do ROS, conjuntos de dados, arquivos de configuração entre outras coisas. O ROS possui mais de 2000 bibliotecas de software disponíveis para serem usadas.
- ***Manifests*:** *Manifests* (*manifest.xml*) provêem metadados a respeito de um pacote, incluindo informação a respeito da sua licença e suas dependências, bem como informações específicas de linguagens como flags de compilador.
- **Pilhas:** pilhas são coleções de pacotes que provêem funcionalidade em massa, como uma "pilha de navegação". As pilhas também são a forma como o software do ROS é lançado, sendo que as pilhas possuem um número de versão associado a elas.
- ***Manifests* de pilha:** *manifests* de pilha (*stack.xml*) provêem dados a respeito da pilha de forma semelhante aos manifests de pacotes.

- **Tipos de Mensagem (msg):** definem a estrutura de dados da requisição e da resposta para as mensagens enviadas no ROS.

  As descrições são armazenadas em *my_package/msg/MyMessageType.msg*.

- **Tipos de Serviço (srv):** definem as estruturas de dados da requisição e da resposta para os serviços no ROS.

  As descrições são armazenadas em *my_package/srv/MyServiceType.srv*.

3.2.2 Nível de Grafo de Computação

O Grafo de Computação é uma rede peer-to-peer de processos ROS que estão processando dados simultaneamente. Os conceitos do Grafo de Computação são: nós (nodes), Mestre (master), Servidor de Parâmetro (parameter server), mensagens (messages), tópicos (topics), serviços (services), e bolsas (bags). Todos estes elementos provêem dados ao grafo de formas diferentes (Todos estes conceitos citados são implementados na pilha *ros_comm*). Também será feita uma descrição da interação entre os componentes do ROS, além de citar um exemplo.

3.2.2.1 Nós

Os nós são processos que realizam computação. O ROS é projetado para ser modular em escala de granularidade fina. Um sistema de controle robótico normalmente abrange diversos nós. Podemos ter diversos nós executando processos diferentes. Por exemplo, um nó poderia receber os dados de um sensor de distância, outro poderia estar controlando os motores do robô e outro planejando o caminho que o robô vai realizar. Um nó ROS é escrito utilizando a biblioteca client do ROS, como o *roscpp* ou *rospy*.

3.2.2.2 Mestre

O ROS Mestre provê o registro de nomes e *lookup* para o resto do Grafo de Computação. Sem o Mestre, os nós não conseguem se encontrar, trocar mensagens e *arrays*.

3.2.2.3 Servidor de Parâmetro

Permite que dados sejam armazenados por uma chave em uma localização central. Atualmente, o servidor de parâmetro faz parte do Mestre.

3.2.2.4 Mensagens

Os nós se comunicam via passagem de mensagens. Uma mensagem é uma estrutura de dados, compreendendo campos 'tipados'. Primitivas padrão (inteiro, variável de ponto flutuante, booleano, etc.) são suportadas, assim como vetores de tipo primitivos. As mensagens podem incluir estruturas aninhadas (como as da linguagem C) e vetores.

3.2.2.5 Tópicos

As mensagens são roteadas por um meio de transporte com a semântica de '*publish / subscribe*'(publicação / inscrição). Um nó envia uma mensagem através da publicação em um dado tópico. O tópico é um nome que é usado para identificar o conteúdo da mensagem. Um nó que está interessado em um certo tipo de dado irá se inscrever ao tópico apropriado. Podem haver diversos publicadores e assinantes concorrentemente para um único tópico, e um único nó pode publicar e se inscrever a múltiplos tópicos. Normalmente, os publicadores e os assinantes não se conhecem. Com isso, ocorre o desacoplamento entre a produção de informação e o seu consumo. Pode-se dizer que um tópico é um barramento fortemente 'tipado' de mensagem. Cada barramento tem um nome e qualquer um pode se conectar ao barramento para enviar ou receber mensagens desde que eles sejam do tipo certo. A Figura 8 ilustra a comunicação via tópico.

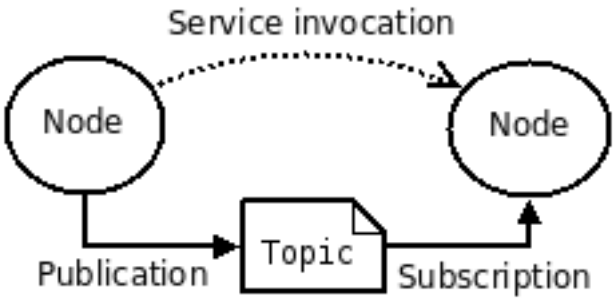

Figura 8 – publish subscribe entre nós por meio de um tópico. Figura extraída de http://www.ros.org/wiki/ROS/Concepts.

3.2.2.6 Serviços

O modelo '*publish / subscribe*' é um padrão muito flexível de comunicação, mas seu muitos-para-muitos e transporte de sentido único não é apropriado para interações de requisição e resposta, os quais são necessários em um sistema distribuído. As requisições e respostas são feitas via serviços, que são definidos por um par de estruturas de mensagens: uma para a requisição e uma para a resposta. Um nó provedor (ou servidor) oferece um serviço através de um nome e seu cliente usa o serviço por meio do envio de uma mensagem de requisição e aguarda a resposta. A biblioteca *client* do ROS normalmente oferece esta interação com o programador como uma chamada remota de procedimento (*remote procedure call*).

3.2.2.7 Bolsas

As bolsas são um formato para se salvar e recuperar os dados de mensagens do ROS. Elas são um mecanismo importante para o armazenamento de dados, como dados de sensores, os quais são difíceis de coletar mas necessários para se desenvolver e testar algoritmos.

3.2.3 Descrição da interação entre os componentes do ROS

O ROS Mestre age como um servidor de nomes no Grafo de Computação do ROS. Ele armazena as informações dos cadastros de tópicos e serviços para os nós do ROS. Os nós se comunicam com o Mestre para informar suas informações de cadastro. Com o decorrer da comunicação entre os nós e o Mestre, esses nós podem receber informação a respeito de outros nós cadastrados e fazer conexões conforme for apropriado. O Mestre também irá realizar *callbacks* para esses nós quando esta informação de cadastro mudar. Isto permite que os nós criem conexões dinamicamente ao mesmo tempo em que os nós estejam executando.

Os nós conectam a outros nós diretamente. O Mestre apenas providencia a informação necessária, como um servidor DNS(*Domain Name System*). Os nós que se inscrevem em um tópico requisitam conexões com os nós que publicam aquele tópico. Eles estabelecem a conexão através de um protocolo de conexão combinado. O protocolo usado no ROS mais comum é chamado de TCPROS, que usa *sockets* TCP/IP padrões.

Esta arquitetura permite operações desacopladas, onde o nome é o principal meio pelo qual sistemas mais complexos podem ser construídos. Os nomes têm um papel importante no ROS: tanto os nós como os tópicos, serviços e parâmetros possuem nomes. Toda biblioteca client do ROS suporta o 'remapeamento' de nomes via linha de comando, o que significa que um programa compilado pode ser reconfigurado em tempo de execução para operar em uma topologia diferente de Grafo de Computação.

O exemplo que será dado a seguir foi retirado da *Wiki* do ROS e ele tem o intuito de ilustrar o 'remapeamento' de nomes.

Por exemplo: para se controlar um medidor de distância a laser da Hokuyo, pode-se iniciar o driver *hokuyo_node*, o qual 'conversa' com o laser e publica mensagens no tópico scan. Para processar esses dados, pode-se criar um nó usando o driver *laser_filters* que se inscreve no tópico *scan*. Após a inscrição, o filtro poderia automaticamente começar a receber mensagens do laser.

Pode-se perceber como tanto o laser quanto o filtro são desacoplados. O *hokuyo_node* apenas publica as leituras do laser, sem saber se alguém está inscrito no seu tópico. E, o filtro apenas se inscreve no tópico para receber as leituras, sem saber se alguém irá publicá-las. Os dois nós podem ser iniciados, 'mortos' e reiniciados em qualquer ordem, sem induzir qualquer condição de erro.

Pode-se adicionar outro laser (utilizando o driver *hokuyo_node*) para o robô. Para isso deve-se reconfigurar o sistema. Essa reconfiguração consiste em 'remapear' os nomes usados. No exemplo dado, poderia 'remapear' o tópico scan para *base_scan*, e fazer a mesma coisa com o nó de filtro. Desta forma, os dois nós se comunicariam por meio do tópico *base_scan* ao invés de se comunicarem por meio do tópico *scan*. Assim, poderíamos iniciar o outro *hokuyo_node* para o novo medidor de distância a laser.

3.2.4 Nível de Comunidade

Os conceitos do Nível de Comunidade são os recursos ROS que habitam comunidades separadas trocar software e conhecimento. Estes recursos incluem:

- **Distribuições:** as distribuições do ROS são coleções de pilhas 'versionadas' que você pode instalar. As distribuições têm um papel semelhante às distribuições de Linux: elas facilitam a instalação de uma coleção de software, e também mantêm versões consistentes de um conjunto de software.
- **Repositórios:** ROS depende de uma rede federada de repositórios de código, onde diferentes instituições podem desenvolver e lançar seus próprios componentes de software.
- ***Wiki* do ROS:** a comunidade *Wiki* do ROS é o principal fórum para documentar informação sobre o ROS. Qualquer pessoa pode se inscrever para receber uma conta e contribuir com documentação, prover correções ou atualizações, escrever tutorial entre outras atividades.
- **Listas de e-mail:** é o principal canal de comunicação dos usuários do ROS, informando atualizações do ROS e também servindo de fórum para se fazer perguntas a respeito do software.

3.3 Nomes

3.3.1 Graph Resource Names

Os Nomes de Grafo de Recursos (*graph resource names*) são um mecanismo para fornecer encapsulamento no ROS. Cada recurso é definido dentro de um *namespace*, que pode ser compartilhado com muitos outros recursos. No geral, os recursos podem criar outros recursos dentro do seu *namespace*. Os recursos podem ter acessos a recursos dentro ou acima dos seus próprios *namespace*. As conexões podem ser feitas entre recursos em *namespaces* distintos, mas isso geralmente é feito por meio de integração de código acima dos dois *namespaces*. Este encapsulamento evita que diferentes porções do sistema acidentalmente peguem um recurso de nome errado ou invadam globalmente nomes.

Os nomes são resolvidos relativamente, portanto os recursos não precisam estar conscientes de qual *namespace* eles estão dentro. Isto simplifica a programação já que os nós que funcionam juntos podem ser escritos como se eles estivessem no namespace de nível mais alto. Quando esses nós estão integrados em sistemas maiores, eles podem ser 'empurrados' para baixo em um namespace que defina sua coleção de código. Por exemplo, pode-se pegar um demo de *Stanford* e um da *Willow Garage* e mesclá-los em um novo *demo* com os subgrafos de stanford e de wg (willow garage). Se os dois *demos* tiverem um nó chamado 'camera', não iria ocorrer conflito entre eles.

3.3.2 Package Resource Names

Os nomes de recursos de pacotes (*package resource names*) são utilizados no nível de sistema de arquivos do ROS para simplificar o processo de referência a arquivos e tipo de dados no disco. Os nomes de recursos de pacotes normalmente são: o nome do pacote que o recurso está dentro mais o nome do recurso. Por exemplo: o nome "*std_msgs/String*" se refere ao tipo de mensagem "String" no pacote "*std_msgs*".

Os nomes de recursos de pacotes são semelhantes a um caminho de um arquivo, mas mais curtos. Isso se deve à habilidade do ROS em localizar os pacotes em disco.

3.3.3 Resolução de nomes

Existem quatro tipos de Nomes de Grafo de Recursos no ROS: base, relativo, global e privado, os quais possuem a seguinte sintaxe:

- *base*
- *relative/name*
- */global/name*
- *~private/name*

Por padrão, a resolução é feita relativa ao namespace do nó. Por exemplo, o nó */wg/-node1* tem o namespace */wg*, então o nome *node2* será resolvido para */wg/node2*.

Nomes sem quaisquer qualificadores de namespace são nomes de base. Nomes de base são uma subclasse de nomes relativos e têm as mesmas regras para resolução. Nomes de base são utilizados para inicializar o nome do nó.

Nomes que começam com '/' são globais e são considerados resolvidos. Nomes globais devem ser evitados ao máximo pois eles limitam a portabilidade do código.

Nomes que começam com '~' são privados. Eles convertem o nome em um *namespace*. Por exemplo, *node1* em um *namespace /wg/* tem um namespace privado */wg/node1*. Nomes privados são úteis para a passagem de parâmetros para um nó específico por meio do servidor de parâmetro.

4 Desenvolvimento

Este capítulo organiza e apresenta o desenvolvimento da implementação da integração entre o sistema operacional robótico ROS e o Módulo de Alta Performance da plataforma robótica RoboDeck. Esta integração é feita por meio de um *middleware*, o qual serve como mediador da comunicação entre as aplicações robóticas e a plataforma robótica.

4.1 Ideias iniciais, empecilhos e comentários

A ideia inicial de implementação foi a de utilizar o SDK em C++ para funcionar como o módulo *core* da comunicação direta com o RoboDeck. A implementação do *middleware* se daria por meio da utilização deste SDK como uma biblioteca, sendo que o *middleware* seria escrito em C++, utilizando a API em C++ do ROS. No entanto, o SDK da XBOT foi implementado utilizando o C++/CLI, que é uma extensão da linguagem C++, a qual é utilizada em conjunto com o *framework* .NET da Microsoft. O C++/CLI não é portável para o Linux. Uma tentativa de contorno para este problema, foi realizar a reescrita do SDK em C++ puro. Porém os resultados não foram positivos e a ideia foi abandonada.

A solução definitiva foi a implementação do *middleware* utilizando tanto o SDK em Java, disponibilizado pela XBOT, quanto a implementação em Java do ROS (ROSJava). Como a linguagem Java é conhecida por ser multiplataforma, não existiu o esforço de migrar código do SDK de uma plataforma para outra, no caso, de Windows para Linux.

4.2 Visão Geral do *Middleware*

O *middleware* funciona como um intermediador da comunicação dos programas clientes com o RoboDeck e seu objetivo é oferecer uma API por meio do ROS para esta comunicação. Ele faz a tradução tanto da requisição do cliente para o robô quanto da resposta do robô para o cliente. A Figura 9 ilustra o funcionamento do *middleware*.

Como já foi mencionado anteriormente, tanto o *middleware* quanto as aplicações clientes utilizam o *framework* ROS para permitir a comunicação. Assim, eles devem ser executados como nós da rede do ROS, para que o processo *core* do ROS possa gerenciar a comunicação entre o nó do *middleware* e os nós clientes. Além disso, o *middleware* pode ser executado tanto na placa em que é executado o MAP quanto em um computador externo.

O SDK é utilizado pelo *middleware* para enviar os comandos à plataforma robótica, que possui um protocolo de comunicação por meio de *array* de bytes. Os argumentos das requisições feitas pelo cliente são traduzidos e compõem o *array* de bytes correspondente ao comando

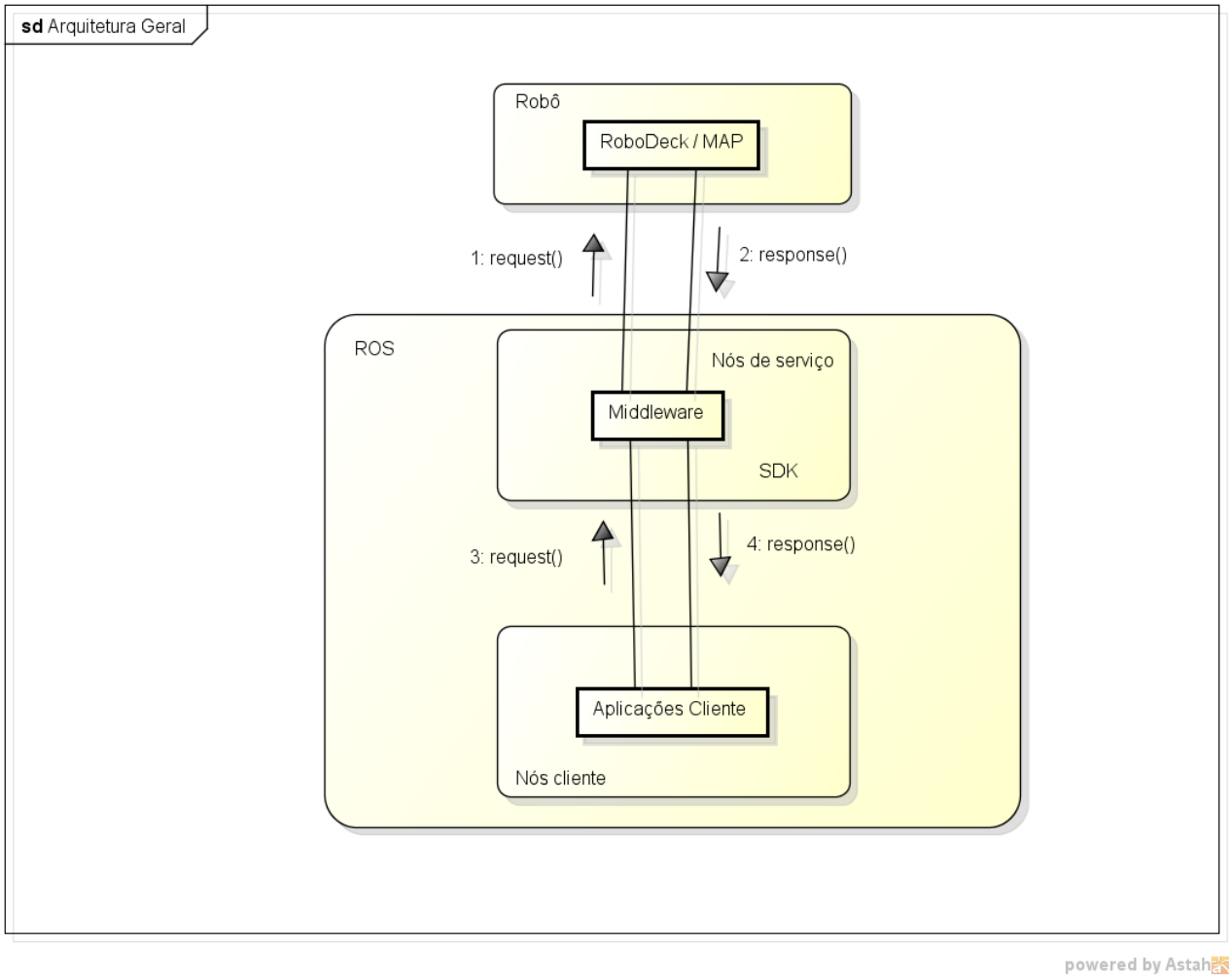

Figura 9 – diagrama de comunicação do *middleware*.

solicitado.

A resposta vinda do robô também é validada e traduzida para compor a resposta da requisição do cliente. O tipo da resposta varia de acordo com o comando solicitado.

4.3 ROSJava

O ROSJava é a primeira implementação em Java do *framework* ROS e foi desenvolvido pela Google em cooperação com a Willow Garage. Ele ainda se encontra em desenvolvimento e em qualidade *alpha* de software, apesar disso oferece todas as funcionalidades do ROS (ROS-JAVA, 2014).

Para a implementação do *middleware* criou-se dois pacotes ROS. O primeiro contém dois sub projetos gradle[1], um do *middleware* e outro do SDK. O segundo pacote define as mensagens dos serviços, as quais são utilizadas pelos clientes para se comunicarem com o

[1] ferramenta de automação de *build*

middleware.

A *build* dos pacotes ROS é gerenciada por um conjunto de macros pertencentes ao pacote *catkin*. Estas macros utilizam os arquivos *package.xml* e *CMakeLists.txt* para realizar a *build* dos pacotes (ROS..., 2014a).

O arquivo *package.xml* precisa ser incluso em qualquer pacote ROS. Este arquivo define propriedades do pacote como o nome e a versão do pacote, os autores, mantenedores e as dependências para outros pacotes (ROS..., 2014b).

Já o arquivo *CMakeLists.txt* é usado como input para sistema de build CMake, que constróis os pacotes. Este arquivo descreve como compilar o código e onde instalar o código compilado (ROS..., 2014c).

Os arquivos *build.gradle* e *settings.gradle* são utilizados pelo sistema de *build* gradle que gerencia a *build* dos pacotes ROSJava.

A estrutura de um pacote ROSJava é exposta na Figura 10.

```
+ pacote_ROSJava
    - package.xml
    - CMakeLists.txt
    - build.gradle
    - settings.gradle
    - gradlew
    + subprojeto_A
        - build.gradle
        + src/main/java
        + src/main/tests
    + subprojeto_B
        - build.gradle
        + src/main/java
        + src/main/tests
```

Figura 10 – estrutura de um pacote ROSJava.

No caso deste trabalho a estrutura do pacote que contém o *middleware* e o SDK é ilustrada na Figura 11.

```
+ robodeck
    - package.xml
    - CMakeLists.txt
    - build.gradle
    - settings.gradle
    - gradlew
    + middleware
        - build.gradle
        + src/main/java
        + src/main/tests
    + SDK
        - build.gradle
        + src/main/java
        + src/main/tests
```

Figura 11 – estrutura do pacote do *middleware*.

No caso do pacote de mensagens, a sua estrutura é exposta na Figura 12.

```
+ robodeck_msgs
    - package.xml
    - CMakeLists.txt
    + srv
        - connect.srv
        - disconnect.srv
        - move.srv
        - turn.srv
        - rotate.srv
        - strafe.srv
        - readInfraredSensors.srv
        - readUltrasonicSensorDistance.srv
        - readUltrasonicSensorLuminosity.srv
        - readOpticalSensor.srv
        - readCompass.srv
        - accelerometer.srv
        - batteryCharge.srv
        - gpsValidate.srv
        - altitude.srv
        - latitude.srv
        - longitude.srv
        - temperatureAndHumidityNorm.srv
        - temperatureAndHumidityRaw.srv
        - expansion.srv
```

Figura 12 – estrutura do pacote de mensagens.

O pacote *robodeck_msgs* contém as definições das mensagens utilizadas para se comunicar com os serviços do *middleware*. A definição da requisição e da resposta de um serviço é feita em um arquivo com extensão *.srv*. A partir destes arquivos, o sistema de *build* catkin consegue gerar as mensagens correspondentes nas linguagens suportadas pelo ROS.

4.4 SDK em Java

A implementação do SDK em Java fornecida pela XBOT não estava completa. A biblioteca não oferecia todas as funcionalidades necessárias para realizar a interação com o robô. Apenas a conexão era realizada com sucesso. Todas as outras funcionalidades foram implementadas (ou reimplementadas) para que funcionassem as chamadas remotas feitas ao robô.

As classes *Accelerometer*, *Battery*, *GPS*, *InfraredSensor*, *Robodeck*, *RobotMotor* e *RobotWifiNetwork* foram modificadas do SDK fornecido. O envio dos comandos da classe RobotMotor estavam errados e ela não possuia o método para o envio de comando de “strafe”. As outras classes do SDK também estavam com os envios de comandos errados. Já as classes *UltrasonicSensor* e *OpticalSensor* foram criadas.

As funcionalidades consertadas e implementadas foram: todas as chamadas de alto nível para a movimentação do robô, as leituras dos sensores ultrassônicos, infravermelhos, bússola, bateria e GPS e chamadas aos comandos estendidos do robô.

A documentação oferecida pela XBOT não oferecia informações suficientes para que fosse implementada a comunicação com o robô por meio de *Socket*, já que esta comunicação é feita por meio de *array* de bytes. Foi necessária a análise do código fonte em C++ do SDK para entender o funcionamento da comunicação com o robô.

4.5 Arquitetura do SDK

A arquitetura do SDK está centrada na classe Robodeck, a qual possui métodos para todos os comandos que podem ser enviados ao robô. Como o MCS do RoboDeck permite que apenas um controlador por vez possa enviar comandos ao robô, deve existir apenas uma instância da classe Robodeck. Caso exista outras instâncias da classe Robodeck, estas não conseguirão realizar uma conexão com o robô caso alguma outra já a tenha feito.

A classe *Robodeck* é composta de outras classes que representam partes do robô, sendo que cada uma delas abstrai os comandos que são possíveis ser feitos a tal parte do robô.

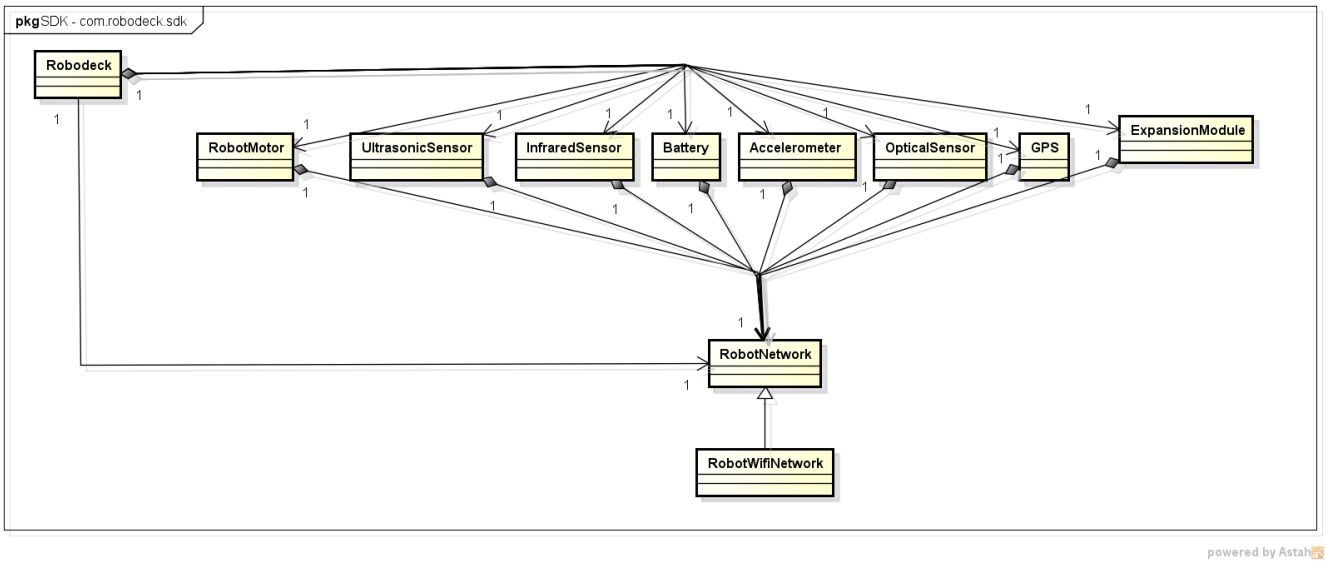

Figura 13 – diagrama de classes do SDK.

- **Robodeck:** classe que centraliza as chamadas de envio de comandos à qualquer parte do robô. É composta das classes RobotMotor, Ultrasonicsensor, Battery, Accelerometer, OpticalSensor, GPS, ExpansionModule e RobotNetwork;
- **RobotMotor:** classe responsável pelo envio de comandos de movimentação do robô;
- **UltrasonicSensor:** classe responsável pelo envio de comandos de leitura de sensores ultrassônicos;
- **InfraredSensor:** realiza o envio de comandos de leitura dos sensores infravermelhos;

- **Battery:** envia comandos de leitura das baterias do robô;
- **Accelerometer:** responsável pelo envio dos comandos de leitura do acelerômetro;
- **OpticalSensor:** responsável pelo envio dos comandos de leitura do sensor óptico;
- **GPS:** envia os comandos para realizar a validação do GPS e as leituras de altitude, latitude e longitude;
- **ExpansionModule:** responsável pelo envio de comandos expandidos ao robô;
- **RobotNetwork:** classe responsável pela comunicação com o robô. Ela define as propriedades e métodos comuns a qualquer comunicação, seja via *Wi-fi*, *bluetooth* ou *zigbee*;
- **RobotWifiNetwork:** implementa a comunicação pelo protocolo de comunicação *Wi-fi* e realiza o envio e o recebimento das mensagens;

4.6 Arquitetura do *middleware*

A arquitetura do *middleware* está centrada na implementação de serviços do ROS, os quais são disponibilizados aos programas clientes para poderem enviar comandos ao RoboDeck. Tanto o *middleware* quanto os programas clientes utilizam os recursos do ROS. Cada serviço representa um comando correspondente do RoboDeck. Os serviços representam APIs para os comandos do RoboDeck. Os programas fazem chamadas aos serviços do *middleware*, em sequência, tais serviços enviam os comandos ao RoboDeck, esperam sua resposta, traduzem-na e respondem ao cliente.

Para melhorar a visibilidade, dividiu-se o diagrama de classes do *middleware* em dois diagramas separados, os quais correspondem às Figuras 14 e 15.

No pacote *com.robodeck.middleware*, a classe *RobotMiddleware* estende a classe *AbstractNodeMain*, é composta de uma instância *Singleton* do *Robodeck* e possui uma relação de 1 para N com a classe RobodeckBaseResponseBuilder. Já a classe RobodeckBaseResponseBuilder implementa a interface ServiceResponseBuilder, que é responsável pela definição do comportamento dos serviços do *middleware*. Cada uma das classes que estendem a classe RobodeckBaseResponseBuilder é responsável pela implementação de um serviço do *middleware*.

4.6.1 Classes do pacote *org.ros.node*

- **AbstractNodeMain:** classe abstrata base que deve ser estendida para permitir que seja criado um nó na rede do ROS;
- **ConnectedNode:** classe abstrata que permite a criação de serviços e tópicos, além de permitir chamar serviços do nó Mestre do ROS e do servidor de parâmetros;

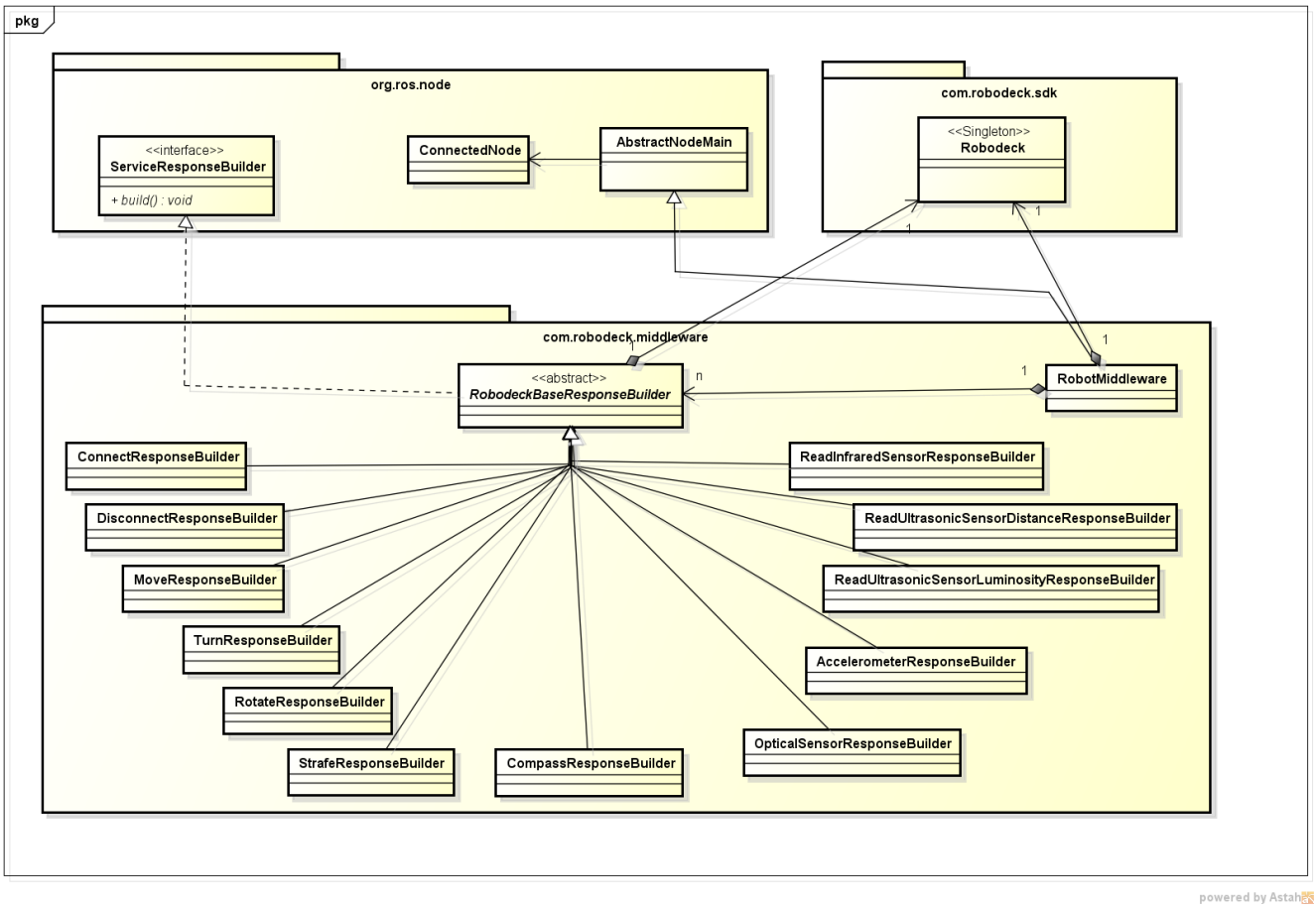

Figura 14 – diagrama de classes do *middleware*.

- **ServiceResponseBuilder (interface):** interface que deve ser implementada para definir o comportamento de um serviço.

4.6.2 Classes do pacote *com.robodeck.middleware*

- **RobotMiddleware:** esta classe estende a classe *AbstractNodeMain* e é executada como um nó da rede do ROS. Ela inicializa todos os serviços do *middleware*;

- **RobodeckBaseResponseBuilder:** classe abstrata que define o método build, o qual é responsável pela resposta das requisições aos serviços do *middleware*;

- **Outros ResponseBuilders:** cada um dos ResponseBuilders definem o comportamento de um serviço correspondente. Por exemplo, a classe ConnectResponseBuilder responde à requisição do serviço de conexão do *middleware*, enviando o comando ao robô e respondendo ao cliente após receber a resposta do robô.

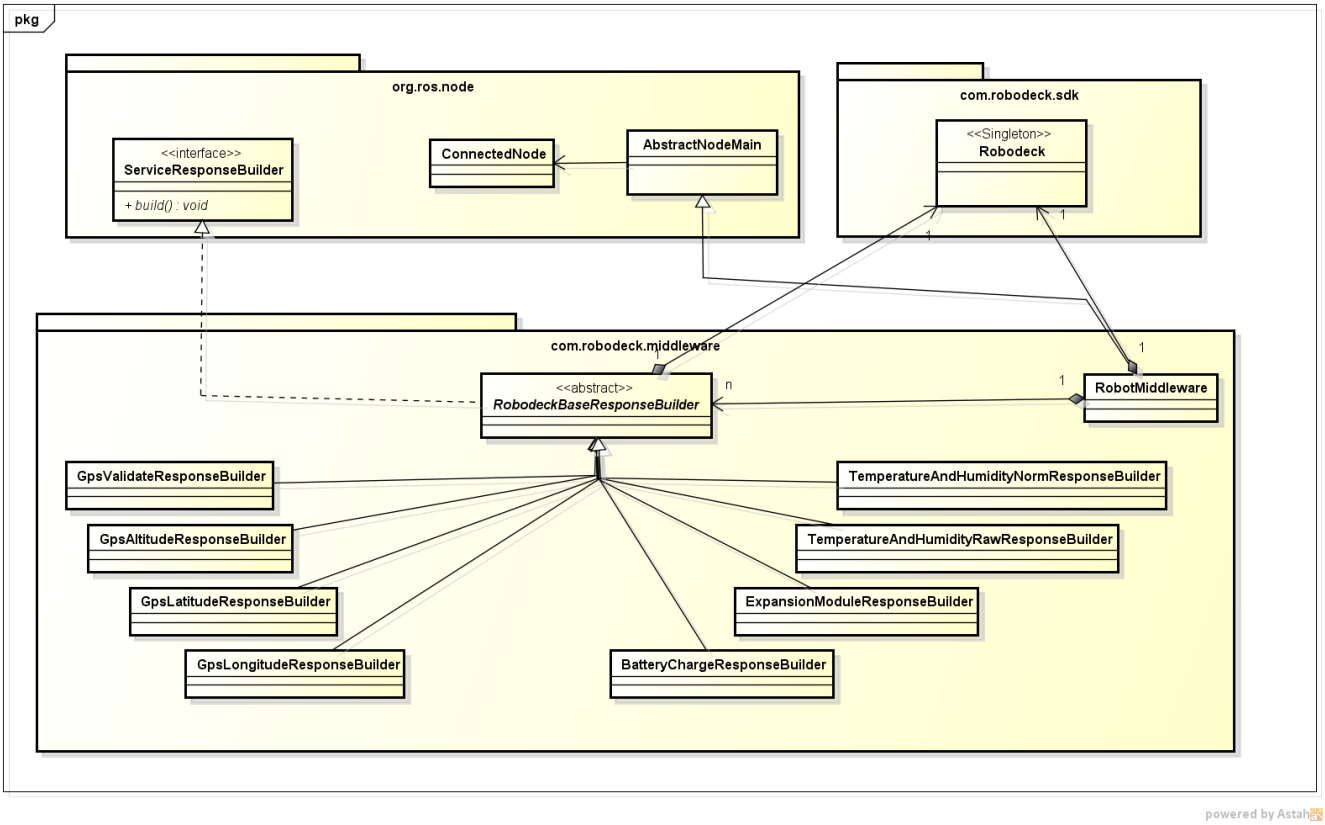

Figura 15 – diagrama de classes do *middleware*.

4.7 Serviços do *middleware*

No pacote *com.robodeck.middleware*, cada um dos ResponseBuilders é responsável pela implementação de um serviço do *middleware*. Na Tabela 1 estes serviços são listados e descritos.

4.8 Diagramas de Sequência de Funcionamento

Os diagramas de sequência a seguir mostram o funcionamento do *middleware* conjuntamente aos programas clientes e ao robô.

O primeiro diagrama mostra como é feita a conexão com o robô por meio do *middleware* a partir de um nó cliente e o segundo diagrama mostra como é feita uma chamada para movimentar o robô.

No primeiro diagrama, um nó cliente envia uma requisição para o serviço "robodeck/connect", o qual é responsável pela conexão com o robô, passando como argumentos o IP e a porta de conexão. Em seguida, a instância da classe *ConnectResponseBuilder* recebe a requisição e instancia um objeto da classe *Robodeck*. Após ser instanciado o objeto *Robodeck*, o *ConnectResponseBuilder* chama o método *connectToRobodeck* da instância *Robodeck*. O método *connectToRobodeck* chama o método *connect* do *RobotNetwork* , que instancia um *socket*

Nome do Serviço	**Descrição**
robodeck/connect	Envia o comando de conexão com o robô
robodeck/disconnect	Envia o comando de desconexão do robô
robodeck/move	Envia o comando de movimentação do robô
robodeck/turn	Envia o comando para o robô realizar uma curva
robodeck/rotate	Envia o comando para o robô realizar uma rotação no próprio eixo
robodeck/strafe	Envia o comando de "strafe" ao robô
robodeck/infraredSensors/read	Envia o comando de leitura dos sensores infravermelhos
robodeck/ultrasonicSensor/distance/read	Envia o comando de leitura de distância dos sensores ultrassônicos
robodeck/ultrasonicSensor/luminosity/read	Envia o comando de leitura de luminosidade dos sensores ultrassônicos
robodeck/opticalSensor/read	Envia o comando de leitura dos sensores ópticos
robodeck/compass/read	Envia o comando de leitura da bússola
robodeck/batteryCharge/read	Envia o comando de leitura das baterias do robô
robodeck/accelerometer/read	Envia o comando de leitura do acelerômetro do robô
robodeck/gps/validate	Envia o comando para validar o GPS
robodeck/gps/altitude/read	Envia o comando de leitura da altitude do GPS
robodeck/gps/latitude/read	Envia o comando de leitura da latitude do GPS
robodeck/gps/longitude/read	Envia o comando de leitura da longitude do GPS
robodeck/temperatureAndHumidity/norm/read	Envia o comando de leitura de temperatura e umidade normalizados
robodeck/temperatureAndHumidity/raw/read	Envia o comando de leitura de temperatura e umidade crus
robodeck/expansion	Envia o comando para realizar a execução de um comando expandido

Tabela 1 – Tabela de Serviços do *middleware*

de conexão com o robô e envia a mensagem de conexão. O robô responde ao *RobotNetwork* caso a conexão tenha sido bem sucedida ou não, e tal resposta é propagada até chegar ao nó cliente.

Analogamente, no segundo diagrama, um nó cliente envia uma requisição para o serviço "robodeck/move" passando como argumento a intensidade que o robô deve se mover. Em seguida, a instância da classe *MoveResponseBuilder* recebe a requisição e chama o método move do *Robodeck*, o qual chama o método *move* da instância da classe *RobotMotor*. A instância da classe *RobotMotor* monta o *array* de bytes correspondente à mensagem de movimentação e chama o método *send* da instância da classe *RobotNetwork* passando como argumento o *array* de bytes. Através do método *send*, a instância do *RobotNetwork* envia o *array* de bytes ao robô por meio do *socket* criado na conexão. Por fim, a resposta do robô é recebida e propagada até o nó cliente.

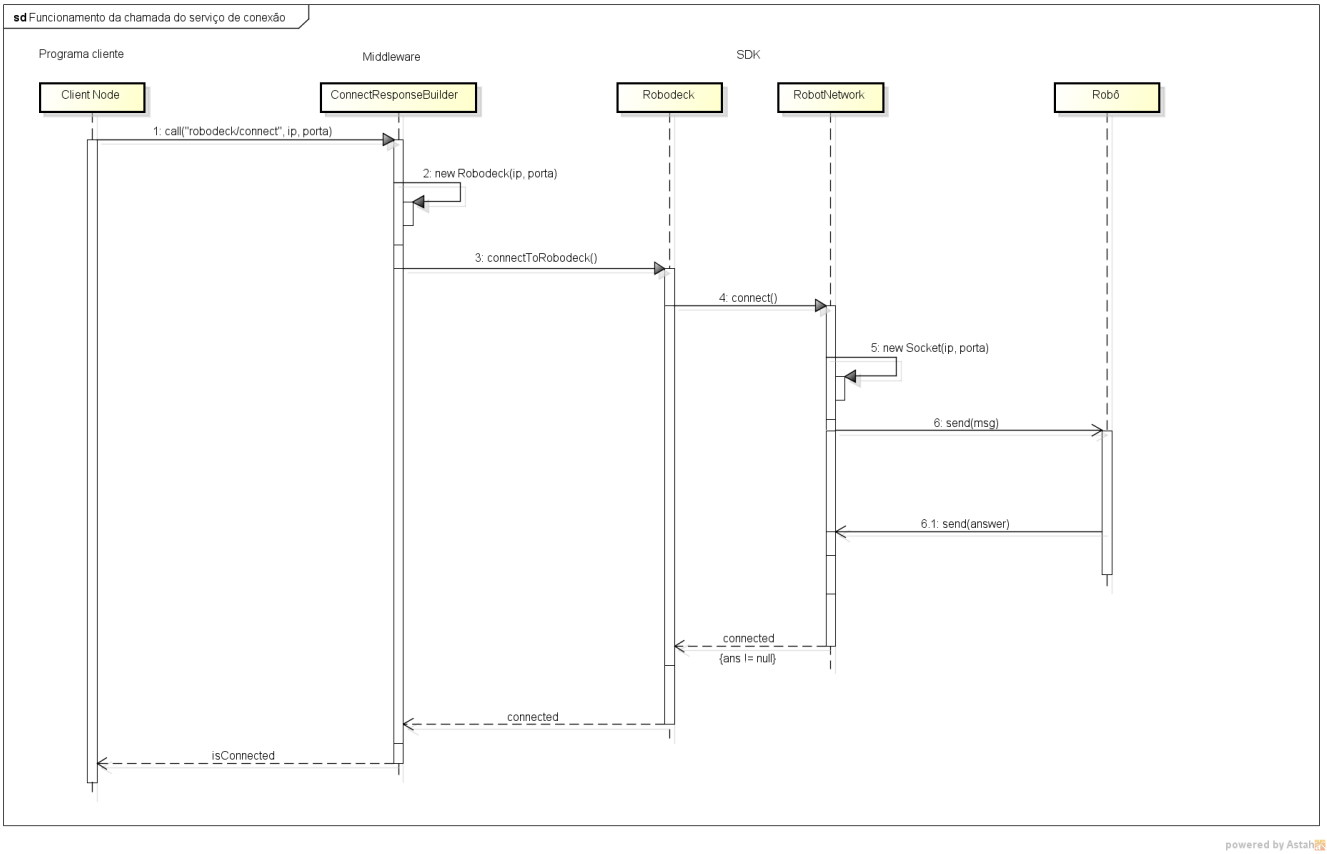

Figura 16 – diagrama de sequência do serviço de conexão.

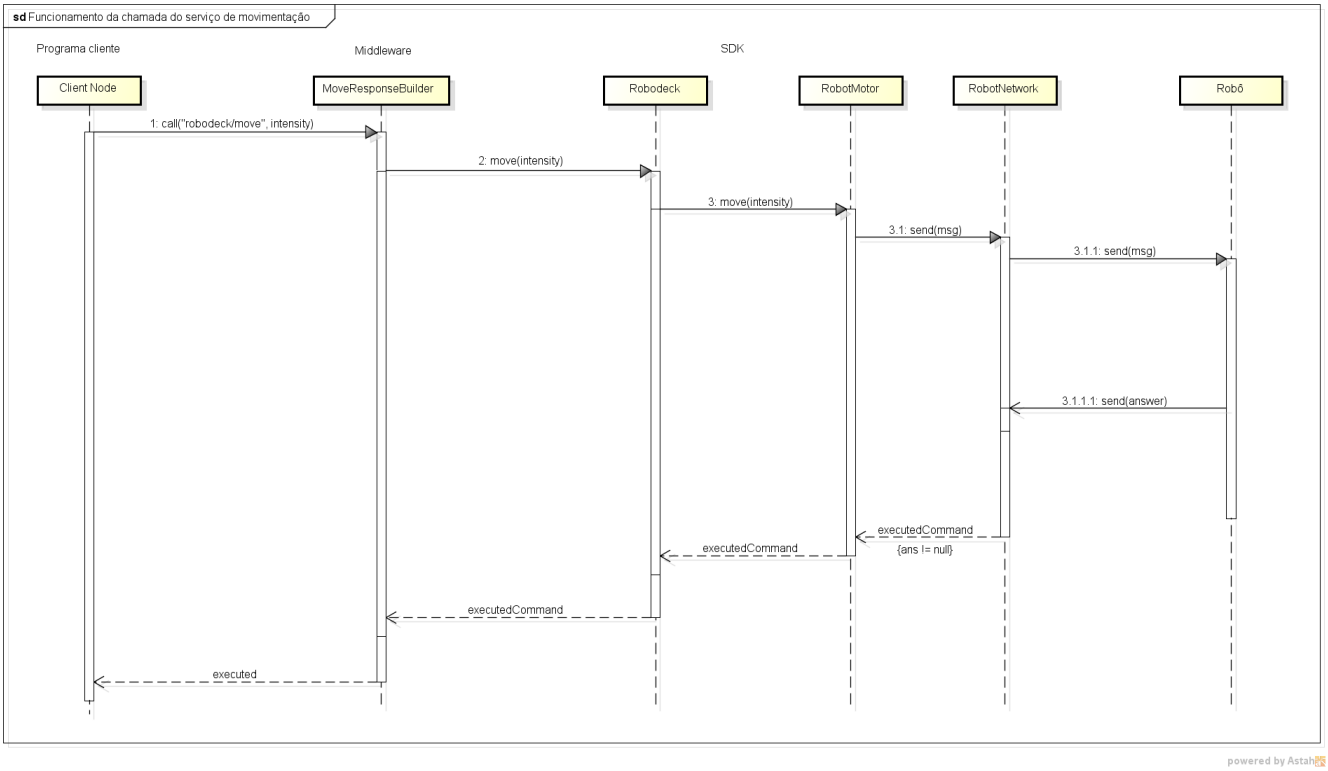

Figura 17 – diagrama de sequência do serviço de movimentação.

5 Analise e Discussão de Resultados

Este capítulo evidência os resultados obtidos e realiza uma análise a partir destes.

5.1 Teste do funcionamento do SDK da XBOT

Primeiramente, foi testado o SDK fornecido pela XBOT (implementação em C++/CLI) por meio do Giga de Testes, que é um programa em C#. Observou-se que aleatoriamente o robô não respondia corretamente aos comandos enviados. Principalmente, em relação à conexão ao robô. Observou-se que a própria plataforma robótica não conseguia executar o comando ou responder ao comando enviado.

5.2 Teste do funcionamento do SDK em Java

Para testar a implementação do SDK, criou-se uma aplicação GUI semelhante ao Giga de Testes da XBOT. Por meio desta aplicação, foi possível testar todas as funcionalidades do SDK com maior facilidade. A Figura 18 mostra a interface gráfica da aplicação.

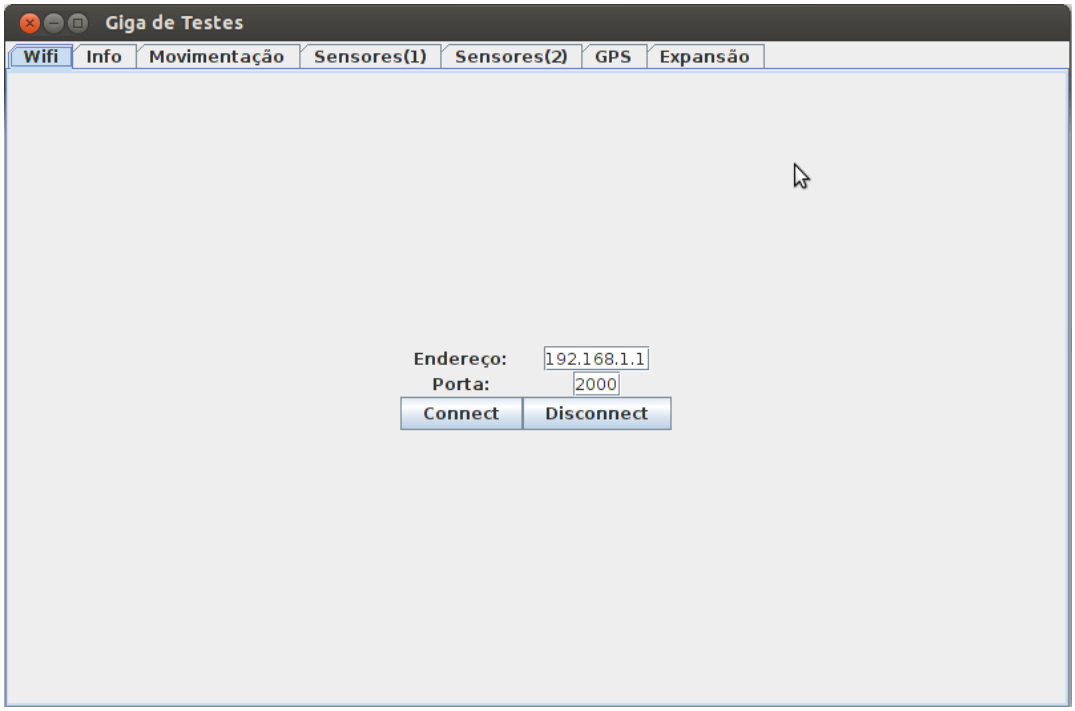

Figura 18 – aplicação GUI de teste.

As abas foram divididas de acordo com as categorias de comandos do RoboDeck. A aba “Wifi” permite realizar a conexão e desconexão com o robô. A aba “info” informa se o

robô está conectado e faz o log de todas as mensagens enviadas e recebidas do robô. A aba "Movimentação" permite realizar todos os movimentos do robô. As abas de sensores contêm botões para realizar as leituras dos sensores do RoboDeck. As abas GPS e expansão, enviam comandos ao GPS e ao módulo de comandos expandidos, respectivamente.

Foram testados todos os comandos do RoboDeck por meio dessa aplicação GUI em Java e assim como o SDK da XBOT, ocasionalmente ocorria do robô não responder corretamente ao comando enviado, assim como no SDK em C++.

5.3 Análise do *middleware*

O desenvolvimento de softwares robóticos são muito complexos, pois a dimensão de código parte do nível mais baixo, como drivers, até a abstração de raciocínio (QUIGLEY et al., 2009). Neste âmbito, os softwares normalmente são escritos em diversos módulos que interagem entre si por meio de mensagens ou passagem de dados (EINHORN et al., 2012). Em um design modular, Einhorn et al. (2012) afirmam que o *middleware* possui um papel importante. Ele serve como uma "cola" que junta os módulos e provê os mecanismos necessários para a comunicação entre os módulos (EINHORN et al., 2012). No caso deste trabalho, ele provê a comunicação entre os programas clientes e a plataforma robótica RoboDeck por meio de uma API de serviços oferecida pelo *middleware*.

Um problema recorrente ao *middleware* é o *overhead* de comunicação entre módulos (EINHORN et al., 2012), já que ocorre a criação e transmissão de mensagens no programa cliente, no *middleware* e no SDK.

5.4 *Framework* ROS

As vantagens oferecidas pelo *middleware* desenvolvido são herdadas pela utilização do *framework* robótico ROS, sendo elas:

- Reutilização de código a partir de pacotes e pilhas disponibilizados pela comunidade do ROS. O ROS é desenvolvido partindo do princípio de desenvolvimento colaborativo de software robótico, assim cada desenvolvedor ou grupo de desenvolvimento pode contribuir para melhorar as bibliotecas e *drivers* disponíveis. Esta reutilização de código ajuda a reduzir o tempo de desenvolvimento.

- Por meio do *middleware* desenvolvido é possível utilizar a plataforma robótica RoboDeck em conjunto com outros robôs que já possuem integração com o ROS. Utilizando um nó mestre, é possível permitir a comunicação com diversos robôs conectados a ele. Isto é, pode-se trabalhar com diversos robôs simultaneamente utilizando a rede do ROS.

- Programação de aplicações clientes independente de linguagem. Atualmente, o ROS dá suporte oficialmente a três linguagens: C++, Python e LISP. No entanto, já existem versões iniciais de pacotes que dão suporte às linguagens Java (ROSJava), Javascript (ROSBridge e roslibjs) e Lua.

5.5 Teste do funcionamento do *middleware*

Para verificar o funcionamento do *middleware* foram feitos programas cliente em diferentes linguagens para testar as chamadas aos serviços. Os programas clientes (apêndices A, B e C) realizam a abertura da conexão com o robô, seguida do comando de movimentação e, por fim, fecha a conexão com o robô. Com exceção do programa do apêndice C, o qual realiza uma chamada a mais para o comando de movimentação antes de se desconectar. Pode-se observar que os programas clientes possuem semelhanças notáveis de código. Os três utilizam a API do ROS para chamar e instanciar os serviços. A API também é usada para construir as requisições e seus respectivos argumentos. No entanto, por se tratar de linguagens diferentes, a API do ROS acaba diferindo entre elas.

No código em C++, utiliza-se o objeto da classe *NodeHandle* para instanciar o serviço, o qual corresponde à classe *ServiceClient*. Já no código em Python, utiliza-se o método "ServiceProxy" do objeto "rospy" para instanciar o serviço. Em relação ao código em Java, utiliza-se a classe *ConnectedNode* para instanciar um serviço.

Os objetos de requisições utilizados nos programas provêm do pacote "robodeck_msgs". Como já foi mencionado, este pacote define o formato das mensagens utilizadas para se comunicar com os serviços do *middleware*. Em C++, tanto a requisição quanto a resposta ficam contidos no mesmo objeto (ex: robodeck_msgs::connect). Já em Python, é possível realizar chamadas tanto com a requisição implícita (ex: chamadas feitas no apêndice B) quanto com a requisição explícita (instanciando um objeto de requisição). Em Java, a requisição é instanciada pelo chamada do método *newMessage* do serviço correspondente.

As respostas, assim como as requisições, provêm do pacote "robodeck_msgs". Em C++, esta resposta já esta contida no objeto da mensagem. Já em Python, ela é retornada pela chamada do serviço. Em Java, ela é passada como parâmetro no método *onSuccess* de um objeto de uma classe que implemente a interface org.ros.service.ServiceResponseListener, sendo que este objeto é passado como parâmetro na chamada do serviço.

5.5.1 Programa cliente em Java

O programa cliente em Java ainda não permite realizar chamadas síncronas aos serviços. Por isso, criou-se uma arquitetura de classes para poder realizar as chamadas aos serviços, de forma que o próximo serviço seja chamado apenas após a chegada da resposta do serviço anterior. No diagrama de classes da Figura 19.

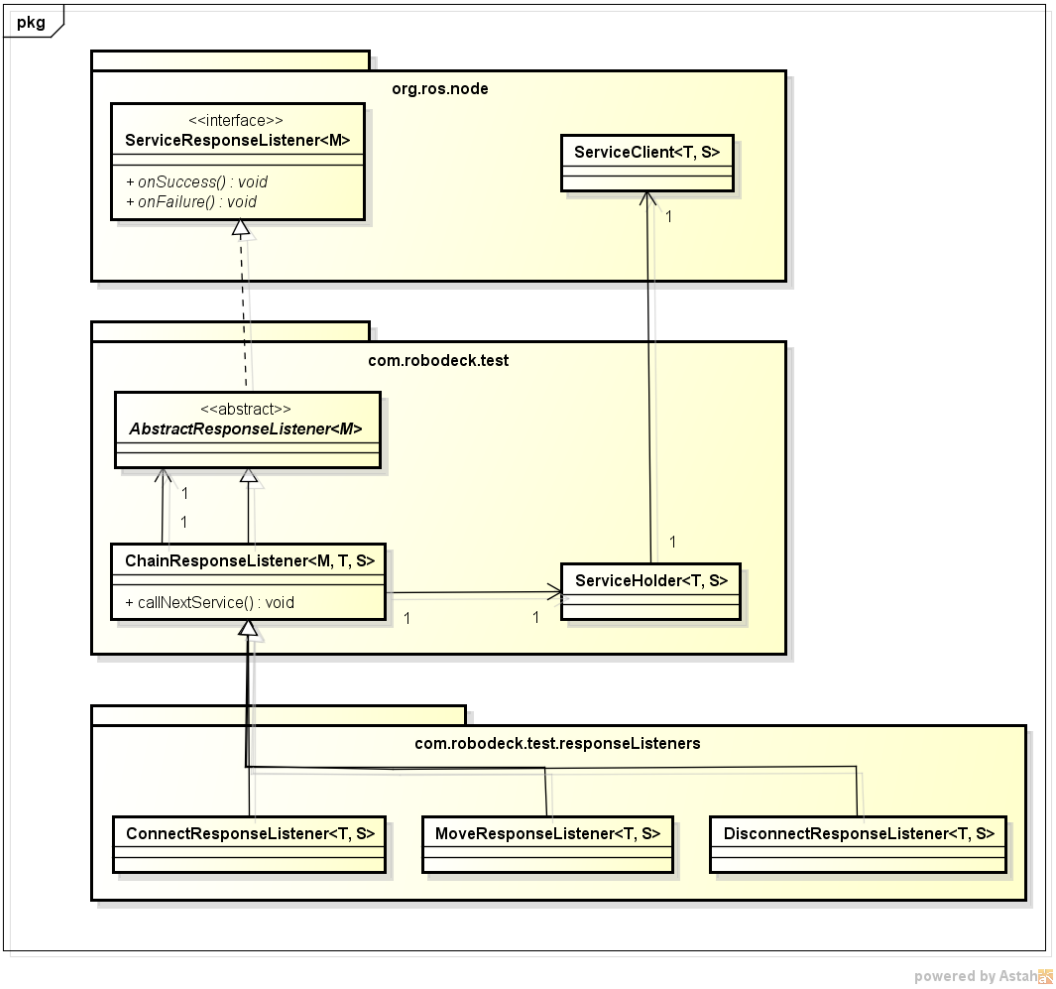

Figura 19 – arquitetura para simular chamadas síncronas em Java.

A classe *ChainServiceResponseListener* permite o encadeamento de chamadas de serviços. O método *callNextService* chama o serviço presente no *ServiceHolder* passando como argumento o ResponseListener contido no *ChainServiceResponseListener*. Este ResponseListener pode ser tanto um *AbstractResponseListener* quanto um *ChainServiceResponseListener*.

6 Conclusão

Este capítulo apresenta as principais contribuições deste trabalho conjuntamente com ideias para a sua continuação.

6.1 Contribuições

Este é o primeiro trabalho a propor uma integração entre o MAP da plataforma robótica RoboDeck e o *framework* ROS. A maior contribuição deste trabalho é permitir uma nova API para realizar a interação com a plataforma robótica RoboDeck. Isso faz com que não seja preciso usar apenas o ambiente fornecido pela XBOT (Giga de Testes e SDK em C++) para se comunicar com o robô. Além disso, é possível construir programas clientes em qualquer linguagem suportada pelo ROS, tornando assim os aplicativos independentes de linguagem e de plataforma.

Outra contribuição foi terminar e consertar a implementação do SDK em Java. Por meio dele é possível interagir com o robô programando em Java puro. No entanto, perde-se as vantagens da utilização dos pacotes e bibliotecas do ROS.

6.2 Trabalhos Futuros

Uma possível melhoria para o *middleware* seria projetar uma forma de utilizar as leituras de sensores e GPS como tópicos do ROS. Ou seja, receber em intervalos regulares a leitura dos valores dos sensores por meio de um tópico.

Outra possível melhoria seria modificar a forma com que o MCS realiza a autenticação do controlador. Atualmente, ele permite que seja aberto um *stream* apenas na porta 2000. Isso limita os envios de comandos ao robô.

Referências

BOHREN, J. *ROS Crash-Course*. 2010. http://www.cs.washington.edu/education/courses/cse466/11au/calendar/ros_cc_1_intro-jrsedit.pdf. Accessed: 2013-07. Citado na página 35.

EINHORN, E. et al. Mira - middleware for robotic applications. In: *IROS*. [S.l.: s.n.], 2012. p. 2591–2598. ISBN 978-1-4673-1737-5. Citado na página 54.

ELKADY, A.; SOBH, T. Robotics middleware: A comprehensive literature survey and attribute-based bibliography. *Journal of Robotics*, v. 2012, p. 15, 2012. Citado na página 22.

KERR, J.; NICKELS, K. Robot operating systems: Bridging the gap between human and robot. In: *System Theory (SSST), 2012 44th Southeastern Symposium on*. [S.l.: s.n.], 2012. p. 99–104. ISSN 0094-2898. Citado na página 21.

MUñOZ, M. E. de S. *Projeto do Software do RoboDeck versão 0.2*. [S.l.], 2010. Documentação técnica. Citado 8 vezes nas páginas 21, 25, 26, 27, 28, 29, 31 e 33.

QUIGLEY, M. et al. Ros: an open-source robot operating system. In: *ICRA Workshop on Open Source Software*. [S.l.: s.n.], 2009. Citado 4 vezes nas páginas 21, 35, 36 e 54.

ROS Wiki. 2013. http://www.ros.org/wiki/ROS/Introduction. Accessed: 2013-07. Citado na página 35.

ROS Wiki. 2013. http://www.ros.org/wiki/ROS/Concepts. Accessed: 2013-07. Citado 2 vezes nas páginas 35 e 36.

ROS Wiki. 2014. http://wiki.ros.org/catkin. Accessed: 2014-06. Citado na página 45.

ROS Wiki. 2014. http://wiki.ros.org/catkin/package.xml. Accessed: 2014-06. Citado na página 45.

ROS Wiki. 2014. http://wiki.ros.org/catkin/CMakeLists.txt. Accessed: 2014-06. Citado na página 45.

ROSJAVA. 2014. https://code.google.com/p/rosjava. Accessed: 2014-06. Citado na página 44.

WOLF, D. F. et al. Robotica inteligente: Da simulação às aplicações no mundo real. In: *Congresso da Sociedade Brasileira de Computação - CSBC, Jornadas de Atualização em Informática - JAI (SBC JAI)*. [S.l.: s.n.], 2009. p. 1–51. Citado na página 21.

XBOT. *Apostila de Software do RoboDeck Versão 1.1*. [S.l.], 2011. Citado 4 vezes nas páginas 21, 25, 26 e 27.

Apêndices

APÊNDICE A – Código cliente cm C++ para a movimentação do robô

```
#include "ros/ros.h"
#include "robodeck_msgs/connect.h"
#include "robodeck_msgs/move.h"
#include "robodeck_msgs/disconnect.h"
#include <cstdlib>

int main(int argc, char **argv) {
  ros::init(argc, argv, "move_test");
  ros::NodeHandle n;

  ros::ServiceClient connectService = n.serviceClient<robodeck_msgs::
    connect>("robodeck/connect");
  robodeck_msgs::connect connectSrv;
  connectSrv.request.ip = "192.168.1.1";
  connectSrv.request.port = 2000;

  ros::ServiceClient moveService = n.serviceClient<robodeck_msgs::move>("
    robodeck/move");
  robodeck_msgs::move moveSrv;
  moveSrv.request.intensity = 32767;

  ros::ServiceClient disconnectService = n.serviceClient<robodeck_msgs::
    disconnect>("robodeck/disconnect");
  robodeck_msgs::disconnect disconnectSrv;

  if (connectService.call(connectSrv)){
    if (connectSrv.response.isConnected) {
      ROS_INFO("Conectou-se ao robo!");

      if (moveService.call(moveSrv)) {
        if (moveSrv.response.executed) {
          ROS_INFO("Robo movimentou!");
        }
      }

      if (disconnectService.call(disconnectSrv)) {
        if (disconnectSrv.response.isDisconnected) {
```

```
35            ROS_INFO("Desconectou-se do robo com sucesso!");
            }
37          else {
              ROS_INFO("Problema ao tentar desconectar");
39          }
        }
41    }
  }else{
43    ROS_ERROR("Falha ao tentar conectar-se ao robo");
      return 1;
45  }

47  return 0;
}
```

move.cpp

APÊNDICE B – Código cliente em Python para a movimentação do robô

```
#!/usr/bin/env python
import rospy
from robodeck_msgs.srv import connect
from robodeck_msgs.srv import move
from robodeck_msgs.srv import disconnect

def connectFunction():
  rospy.wait_for_service('robodeck/connect')
  try:
    connect_proxy = rospy.ServiceProxy('robodeck/connect', connect)
    ans = connect_proxy('192.168.1.1', 2000)
    return ans.isConnected
  except Exception, e:
    raise e

def moveFunction(intensity):
  rospy.wait_for_service('robodeck/move')
  try:
    move_proxy = rospy.ServiceProxy('robodeck/move', move)
    ans = move_proxy(intensity)
    return ans.executed
  except Exception, e:
    raise e

def disconnectFunction():
  rospy.wait_for_service('robodeck/disconnect')
  try:
    disconnect_proxy = rospy.ServiceProxy('robodeck/disconnect', disconnect
    )
    ans = disconnect_proxy()
    return ans.isDisconnected
  except Exception, e:
    raise e

if __name__ == '__main__':
  try:
    if(connectFunction()):
```

```
      print 'conectou-se ao robo'

      if(moveFunction(32767)):
        print 'O robo movimentou-se'

      if(disconnectFunction()):
        print 'Desconectou-se do robo'
      else:
        print 'Problema ao tentar desconectar-se'
    else:
      print 'problema ao tentar enviar o comando de conexao'
  except Exception, e:
    raise e
```

move.py

APÊNDICE C – Código cliente em Java para a movimentação do robô

```
package com.github.robodeck.test;

import org.ros.exception.RosRuntimeException;
import org.ros.exception.ServiceNotFoundException;
import org.ros.namespace.GraphName;
import org.ros.node.AbstractNodeMain;
import org.ros.node.ConnectedNode;
import org.ros.node.service.ServiceClient;

import robodeck_msgs.robodeckSrv;

import com.robodeck.test.responseListeners.ConnectResponseListener;
import com.robodeck.test.responseListeners.DisconnectResponseListener;
import com.robodeck.test.responseListeners.MoveResponseListener;
import com.robodeck.test.responseListeners.ServiceHolder;

public class MoveTest extends AbstractNodeMain {

  private ServiceClient<robodeck_msgs.connectRequest, robodeck_msgs.
    connectResponse> connectService;
  private ServiceClient<robodeck_msgs.moveRequest, robodeck_msgs.
    moveResponse> moveService;
  private ServiceClient<robodeck_msgs.moveRequest, robodeck_msgs.
    moveResponse> stopService;
  private ServiceClient<robodeck_msgs.disconnectRequest, robodeck_msgs.
    disconnectResponse> disconnectService;

  private MoveResponseListener<robodeck_msgs.moveRequest, robodeck_msgs.
    moveResponse> moveResponseListener;
  private MoveResponseListener<robodeck_msgs.disconnectRequest,
    robodeck_msgs.disconnectResponse> stopResponseListener;

  private DisconnectResponseListener disconnectResponseListener;

  private ConnectResponseListener<robodeck_msgs.moveRequest, robodeck_msgs.
    moveResponse> connectResponseListener;
```

```
31    @Override
      public GraphName getDefaultNodeName() {
33      return GraphName.of("robodeckTest/move");
      }
35
      @SuppressWarnings({ "rawtypes", "unchecked" })
37    @Override
      public void onStart(ConnectedNode connectedNode) {
39
        // inicializando servico de conexao
41      try {
          this.connectService = connectedNode.newServiceClient("robodeck/
        connect", robodeck_msgs.connect._TYPE);
43      } catch (ServiceNotFoundException e) {
          throw new RosRuntimeException(e);
45      }

47      try {
          this.moveService = connectedNode.newServiceClient("robodeck/move",
        robodeck_msgs.move._TYPE);
49      } catch (ServiceNotFoundException e) {
          throw new RosRuntimeException(e);
51      }

53      try {
          this.stopService = connectedNode.newServiceClient("robodeck/move",
        robodeck_msgs.move._TYPE);
55      } catch (ServiceNotFoundException e) {
          throw new RosRuntimeException(e);
57      }

59      try {
          this.disconnectService = connectedNode.newServiceClient("robodeck/
        disconnect", robodeck_msgs.disconnect._TYPE);
61      } catch (ServiceNotFoundException e) {
          throw new RosRuntimeException(e);
63      }

65      robodeck_msgs.connectRequest connectRequest = this.connectService.
        newMessage();
        connectRequest.setIp("192.168.1.1");
67      connectRequest.setPort(2000);

69      robodeck_msgs.moveRequest moveRequest = this.moveService.newMessage();
        moveRequest.setIntensity((short) 32767);
71
        robodeck_msgs.moveRequest stopRequest = this.stopService.newMessage();
```

```
73      stopRequest.setIntensity((short) 0);

75      robodeck_msgs.disconnectRequest disconnectRequest = this.
    disconnectService.newMessage();

77      this.disconnectResponseListener = new DisconnectResponseListener();

79      this.stopResponseListener = new MoveResponseListener(connectedNode,
    disconnectRequest,
            "robodeck/disconnect", robodeck_msgs.disconnect._TYPE, this.
    disconnectResponseListener,
81          new ServiceHolder<robodeck_msgs.disconnectRequest, robodeck_msgs.
    disconnectResponse>(this.disconnectService));

83      this.moveResponseListener = new MoveResponseListener(connectedNode,
    stopRequest,
            "robodeck/move", robodeck_msgs.move._TYPE, this.
    stopResponseListener,
85          new ServiceHolder<robodeck_msgs.moveRequest, robodeck_msgs.
    moveResponse>(this.stopService));

87      this.connectResponseListener = new ConnectResponseListener(
    connectedNode, moveRequest,
                        "robodeck/move", robodeck_msgs.move._TYPE, this.
    moveResponseListener,
89                      new ServiceHolder<robodeck_msgs.moveRequest,
    robodeck_msgs.moveResponse>(this.moveService));

91      System.out.println("Tentando conectar...");
        // so eh preciso chamar o servico de conexao, pois os outros sao
    chamados em cadeia
93      // connectResponseListener => moveResponseListener =>
    stopResponseListener => disconnectResponseListener
        this.connectService.call(connectRequest, this.connectResponseListener);
95    }

97  }
```

MoveTest.java