INTERFACE TECNOLÓGICA

DESENVOLVIMENTO DE MÓDULO PARA INTERFACEAMENTO COM PLATAFORMA ROBÓTICA MÓVEL

DEVELOPMENT OF MODULE FOR INTERFACE WITH MOBILE ROBOTIC PLATFORM

Antonio Valerio Netto – antonio.valerio@pq.cnpq.br
Pesquisador DT CNPq – Brasília – DF – Brasil

DOI: 10.31510/infa.v15i1.323

RESUMO

O artigo descreve o desenvolvimento de um interfaceamento baseado no *framework* robótico ROS (*Robot Operating System*) com o objetivo de permitir a programação de uma plataforma robótica móvel desenvolvida no Brasil, chamada RoboDeck. Dessa forma, entende-se que as aplicações possam ser mais facilmente desenvolvidas para este robô permitindo acelerar o processo de criação de módulos e funções. Como resultado, a plataforma pode ser controlada por meio do *middleware* utilizando as bibliotecas clientes do ROS. Concluiu-se que as principais vantagens da criação desse *middleware* estão relacionadas à modularidade do *software* e abstração da arquitetura de *hardware* independência de plataforma, melhorando a portabilidade.

Palavras-chave: ROS. RoboDeck. Robótica móvel. Plataforma robótica.

ABSTRACT

The article describes the development of an interface based on the Robot Operating System (ROS) framework to allow the programming of a robotic mobile platform developed in Brazil, called RoboDeck. In this way, it is understood that the applications can be more easily developed for this robot allowing to accelerate the process of creating modules and functions. As a result, the platform can be controlled through the middleware using the ROS client libraries. It was concluded that the main advantages of the creation of this middleware are related to software modularity and hardware architecture abstraction platform independence, improving portability.

Keywords: ROS. RoboDeck. Mobile robotics. Robotic Platform.

1 INTRODUÇÃO

Aplicações envolvendo robôs móveis em ambientes dinâmicos são as mais interessantes, exigindo um sistema robusto e versátil, capaz de manter-se operacional diante de mudanças imprevisíveis na estrutura desses ambientes. Esse tipo de sistema deve possuir

mecanismos de percepção e de controle que, sistematicamente, adaptam os robôs às novas situações encontradas. Neste contexto, muitos desafios da área de Robótica têm ainda que ser superada no sentido de tornar um robô autônomo, dependendo, atualmente, mais de *software* do que do *hardware* (NETTO e GONÇALVES, 2016). O desenvolvimento de aplicações para plataformas robóticas 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. Isto acaba tornando uma tarefa muito especializada o que dificulta a popularização da plataforma RoboDeck (WEI, 2015) (ZANOLLA et al., 2017) junto aos desenvolvedores. Diante desse fato, faz se necessário um sistema operacional robótico que pudesse auxiliar e acelerar nas tarefas de desenvolvimento de aplicações, por exemplo, envolvendo análise de sinais e de processamento de imagens digitais.

Por isso, para este projeto foi escolhido um *frameworks* robótico, mundialmente popular entre os "robóticos", chamado ROS (*Robot Operating System*) (ROS, 2014a) para realizar sua integração com o Módulo de Alta Performance (MAP) (PISSARDINI, 2014) da plataforma RoboDeck. Dessa forma, entende-se que aplicações envolvendo visão computacional para a navegação dos robôs ou para outras aplicações, possam ser mais facilmente desenvolvidas. Como por exemplo, a utilização de um dispositivo Kinect da empresa Microsoft para capturar imagens e sinais de movimento corporal. O sistema de visão ativo baseado no Kinect pode ser capaz de desempenhar determinadas tarefas, tais como, reconhecer sinais manuais (realizados com a mão ou movimentos de outra parte do corpo) em uma imagem. O sistema pode também varrer a imagem em tempo real e procurar por regiões que foram modificadas, representando, por exemplo, objetos em movimento. Após a localização do objeto, o sistema pode identificá-lo como obstáculo ou alvo, e direcionar o comportamento da navegação do robô de acordo com o tipo do objeto, desviar ou se mover até ele. O artigo está dividido em mais quatro seções sendo que a seção dois fornece uma noção sobre o ROS (*Robot Operating System*), posteriormente na seção três é descrita a metodologia aplicada no desenvolvimento do projeto. Na seção quatro os resultados obtidos com a integração dos sistemas, e na seção cinco, as considerações finais conforme os resultados obtidos.

2 NOÇÕES SOBRE O ROS (*ROBOT OPERATING SYSTEM*)

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, 2014a). 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, 2014b). O ROS também permite que a colaboração seja distribuída por meio do suporte aos repositórios de código. Este modelo permite decisões independentes em relação ao desenvolvimento e à implementação. 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 é explicado a seguir:

- 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 micro-kernel, 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 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 pudessem realizar o desenvolvimento rápido de sistemas robóticos sem precisar "reinventar a roda" por meio da utilização das ferramentas do *framework* (QUIGLEY et al., 2009). 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*) (ROS, 2014b).

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 (ROS, 2014b).

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 por meio de um protocolo de conexão combinado. O protocolo usado no ROS mais comum é chamado de TCPROS, que usa *sockets* TCP/IP padrões (ROS, 2014c).

Esta arquitetura permite operações desacopladas, onde o nome é o principal meio pelos quais 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 "*cliente"* 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 (ROS, 2014c) e ele tem o intuito de ilustrar o 'remapeamento' de nomes. Por exemplo: para se controlar um medidor de distância a laser da empresa Hokuyo, pode-se iniciar o *driver* "hokuyo_node", o qual se comunica 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 pode 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. É possível 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, é possível iniciar o outro “hokuyo_node” para o novo medidor de distância a laser. Os conceitos do nível de comunidade são os recursos ROS que habitam comunidades separadas trocar *software* e conhecimento. Estes recursos incluem (ROS, 2014c):

- Distribuições: as distribuições do ROS são coleções de pilhas ‘versionadas’ que pode ser instalado. 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 (ROS, 2014c).
- 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 perguntar a respeito do *software*.

3 METODOLOGIA

Foi realizado um estudo aprofundado do ROS utilizando principalmente documentações do site "*www.ros.org*", 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 (MAP). 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 (*Software Development Kit*) em Java e a implementação do *middleware* (utilizando tanto o SDK quanto o ROS). A análise foi realizada a partir do estudo do *framework* ROS e dos resultados obtidos nos testes do *middleware*.

No caso do MAP, o mesmo fornece três categorias de métodos aos aplicativos robóticos: Para responder ao controlador; Para enviar comandos ao robô; e 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 Módulo de Controle de Sessão (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.

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 deu por meio da utilização do SDK como uma biblioteca, sendo que o *middleware* foi escrito em C++, utilizando a API (*Application Programming Interface*) em C++ do ROS. No entanto, o SDK da empresa XBot (*www.xbot.com.br*) 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. 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. 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 realizadas pelo cliente são traduzidos e compõem o *array* de *bytes* correspondente ao comando 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. O ROSJava é a primeira implementação em Java do *framework* ROS e foi desenvolvido pela Google em cooperação com a Willow Garage (ROSJAVA, 2014).

Para a implementação do *middleware* criou-se dois pacotes ROS. O primeiro contém dois subprojetos *gradle* (ferramenta de automação de *build*), 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 *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.

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 realizado em um arquivo com extensão “.srv”. A partir desse arquivo, o sistema de *build catkin* consegue gerar as mensagens correspondentes nas linguagens suportadas pelo ROS. A implementação do SDK em Java fornecida pela empresa 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 realizadas 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 possuía 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, GPS e chamadas aos comandos estendidos do robô. A documentação oferecida pela empresa XBot não continham informações suficientes para que fosse implementada a comunicação com o robô por meio de *Socket*, já que esta comunicação é realizada 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.

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 existam outras instâncias dessa classe, estas não conseguirão realizar uma conexão com o robô caso alguma outra já a tenha realizado. A classe “RoboDeck” é composta de outras classes que representam partes do robô, sendo que cada uma delas abstrai os comandos que são possíveis serem realizados por tal parte do robô.

- Robodeck: classe que centraliza as chamadas de envio de comandos a 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 RESULTADOS

Para testar a implementação do SDK, criou-se uma aplicação GUI (*Graphical User Interface*) semelhante a uma Giga de testes. Por meio dessa aplicação, foi possível testar todas as funcionalidades do SDK com maior facilidade. Na interface gráfica da aplicação, as abas foram divididas de acordo com as categorias de comandos do RoboDeck. A aba “Wi-Fi” 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.

Para verificar o funcionamento do *middleware* foram desenvolvidos programas “cliente” em diferentes linguagens para testar as chamadas aos serviços. Os programas

“clientes” realizam a abertura da conexão com o robô, seguida do comando de movimentação e, por fim, fecha a conexão com o robô. Esses programas 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 quanto com a requisição explícita (instanciando um objeto de requisição). Em Java, a requisição é instanciada pela 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.

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. 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”.

5 CONSIDERAÇÕES FINAIS

No desenvolvimento da integração do ROS com o MAP do RoboDeck, criou-se uma aplicação GUI no qual foi possível testar todas as funcionalidades do SDK com maior

facilidade. O desenvolvimento de *softwares* robóticos é muito complexo, pois a dimensão de código parte do nível mais baixo, como *drivers*, até a abstração de raciocínio. Neste âmbito, os *software*s normalmente são escritos em diversos módulos que interagem entre si por meio de mensagens ou passagem de dados. Em um *design* modular 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. No caso deste projeto, ele provê a comunicação entre os programas clientes e a plataforma RoboDeck por meio de uma API de serviços oferecida pelo *middleware*. Um problema recorrente ao *middleware* é o *overhead* de comunicação entre módulos, já que ocorre a criação e transmissão de mensagens no programa cliente, no *middleware* e no SDK.

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 suporta oficialmente três linguagens: C++, Python e LISP. No entanto, já existem versões iniciais de pacotes que dão suporte às linguagens Java (ROSJava), Javascript (ROS - *Bridge* e roslibjs) e Lua.

Importante salientar que foi o primeiro projeto a propor uma integração entre o MAP da plataforma RoboDeck e o *framework* ROS. A maior contribuição é permitir uma nova API para realizar a interação com o RoboDeck. Isso faz com que não seja preciso usar apenas o ambiente de programação fornecido pela empresa fabricante do RoboDeck (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.

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.

AGRADECIMENTO

Apoio do CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico), por meio do seu programa de Desenvolvimento Tecnológico e Extensão Inovadora (DT).

REFERÊNCIAS

BOHREN, J. **ROS Crash-course.** 2010. Disponível em: <http://www.cs.washington.edu/education/courses/cse466/11au/calendar/ros_cc_1_intro-jrsedit.pdf>. Acesso em: 13 jan. 2016.

NETTO, A. V.; GONÇALVES, L. M. G. **Robótica computacional e robótica educacional**: ferramentas para o conhecimento e inclusão tecnológica. Grandes desafios da SBC, Sociedade Brasileira de Computação. Disponível em: < http://www.xbot.com.br/wp-content/uploads/2012/10/Grandes_1-0.pdf>. Acesso em: 6 nov. 2016.

PISSARDINI, R. S. **Veículos autônomos de transporte terrestre**: proposta de arquitetura de tomada de decisão para navegação autônoma, 2014. Dissertação de mestrado, Universidade de São Paulo. Disponível em: <http://www.teses.usp.br/teses/disponiveis/3/3138/tde-26082015-161805/en.php>. Acesso em: 10 jan. 2016.

QUIGLEY, M. et al. **ROS**: an open-source robot operating system. In: ICRA Workshop on Open Source *Software*, 2009.

ROS, **Wiki**. 2014a. Disponível em: <http://www.ros.org/wiki/ROS/Concepts>. Acesso em: 10 nov. 2016.

ROS, **Wiki**. 2014b. Disponível em: <http://wiki.ros.org/catkin>. Acesso em: 10 nov. 2016.

ROS, **Wiki**. 2014c. Disponível em: <http://wiki.ros.org/catkin/CMakeLists.txt>. Acesso em: 10 nov. 2016.

ROSJAVA. **An implementation of ROS in pure java with android support.** 2014. Disponível em: <https://code.google.com/p/rosjava>. Acesso em: 13 nov. 2016.

WEI, D. C. M. **Método de desvio de obstáculos aplicado em veículo autônomo**, Dissertação de mestrado. Universidade de São Paulo, 2015. Disponível em: <http://www.teses.usp.br/teses/disponiveis/3/3138/tde-17062016-142254/en.php>. Acesso em: 5 nov. 2016.

ZANOLLA, L. et al. **Implementação com validação real de um controle proporcional, integral e derivativo na plataforma robótica roboDeck,** Disponível em: <http://www.xbot.com.br/wp-content/uploads/2012/10/artigo_Implementacao_ValidacaoReal_leandro.pdf>. Acesso em: 10 jan. 2017.