UNIVERSIDADE FEDERAL DE ITAJUBÁ

PROGRAMA DE PÓS-GRADUAÇÃO EM
CIÊNCIA E TECNOLOGIA DA COMPUTAÇÃO

ROSRemote: Utilizando ROS para acesso remoto a robôs

Alyson Benoni Matias Pereira

Itajubá, março de 2018

UNIVERSIDADE FEDERAL DE ITAJUBÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM
CIÊNCIA E TECNOLOGIA DA COMPUTAÇÃO

Alyson Benoni Matias Pereira

ROSRemote: Utilizando ROS para acesso remoto a robôs

Dissertação submetida ao Programa de Pós-Graduação em Ciência e Tecnologia da Computação como parte dos requisitos para obtenção do Título de Mestre em Ciência e Tecnologia da Computação

Área de Concentração: Matemática da Computação

Orientador: Prof. Dr. Guilherme Sousa Bastos

março de 2018
Itajubá - MG

Dedico aos meus pais, minha irmã e minha noiva, que sempre me incentivaram e me deram força para concluir esse trabalho.

Agradecimentos

Agradeço, primeiramente à Deus, que me deu força, capacidade e sabedoria suficientes para concluir esse trabalho.

Agradeço aos meus pais, David e Cida, que não mediram esforços e sempre investiram em minha educação me permitindo ter condições de atingir todos os meus objetivos.

Agradeço a minha irmã Bárbara e ao meu cunhado César, que auxiliaram na conclusão desse trabalho sempre me dando ideias de quais caminhos seguir.

Agradeço a minha noiva Ana Paula, que comemorou comigo os momentos bons e me suportou e apoiou nos momentos de dificuldade, sempre me dando força pra seguir em frente quando a vontade era desistir.

Agradeço ao meu orientador, que me mostrou sempre o melhor caminho, me cobrou e me forçou a procurar a melhor solução, principalmente quando eu mesmo achava ser impossível.

Agradeço, por fim, aos meus colegas e amigos, que me ajudaram a aprender as bases necessárias para desenvolver esse projeto e me mostraram que as dificuldades enfrentadas eram comuns para todos.

Quer você acredite que consiga
fazer uma coisa ou não,
você está certo.
(Henry Ford)

Resumo

Aplicações em nuvem estão são o foco de muitas pesquisas devido ao fato de ser fácil obter recursos baratos e praticamente ilimitados utilizando-se servidores de internet. Isso permite que usuários possam manipular qualquer quantidade de dados. Com a robótica não é diferente e utilizando a ferramenta ROS (*Robot Operating System*) é possível controlar robôs de forma simples e fazê-los realizar qualquer trabalho necessário. Porém, o ROS possui a desvantagem de somente funcionar em uma rede local e não na internet. A forma mais comum de combinar robótica e aplicações na nuvem é a chamada "Robótica na nuvem", que permite a realização de cálculos e armazenamento de dados em servidores, porém não permite o controle de robôs por meio da internet. Com base nesse fato, esse trabalho propõe o ROSRemote, uma ferramenta que possibilita a utilização das funcionalidades do ROS em acesso remoto a robôs, permitindo que comandos sejam enviados e recebidos. Utilizando o *framework SpaceBrew* para o envio de informações, os usuários podem realizar tarefas com os comandos implementados no ROSRemote ou criar suas próprias aplicações que funcionarão utilizando um recurso que se encontra na nuvem. Assim, o ROSRemote pode ser classificado como um *Framework* que possibilita a criação de aplicações com o ROS que funcionem de forma remota. Para verificar a viabilidade do ROSRemote foram utilizados computadores do laboratório de robótica da Unifei e um robô *AmigoBot* para realizar os testes. A coleta de tempo de tráfego de dados foi feita diretamente no terminal do Ubuntu e com a utilização do software *WireShark* e a velocidade da internet foi medida utilizando-se ferramentas especializadas para essa tarefa. Ao fim o ROSRemote se mostrou mais rápido do que outras ferramentas, podendo não só ser utilizado para controle, mas também para monitoramento de robôs, permitindo que o usuário crie aplicações novas e divida seus dados com outros usuários.

Abstract

Cloud applications are the focus of many researches due to the fact that it is easy to get cheaper and practically unlimited resources using internet servers. This allow users to be able to handle any amount of data. The same thing happens with robotics and using ROS (Robot Operation System) toolkit it is possible to control robots and make them perform any necessary job. However, ROS has the disadvantage that it only works in a local network and not over the internet. The most common form of combining robotics and cloud applications is the "Cloud Robotics", which allows performing calculations and storage data into servers, but do not helps users to control robots through internet. Based on this fact, this work introduces ROSRemote, a tool that helps users to use ROS functionalities when accessing remote robots, making possible to send and receive commands. Using *SpaceBrew* framework to send information, users can work with commands implemented in ROSRemote or use it to create their own applications that will work remotely. So ROSRemote can be classified as a Framework that allows the creation of application with ROS that works remotely. To verify ROSRemote viability it was used computers at Robotics laboratory at Unifei and an AmigoBot robot to perform tests. Data traffic time collection was done directly on Ubuntu terminal and using WireShark software, internet speed was measured using tools developed for this task. At the end, ROSRemote proved itself to be faster than other tools and helps users to control and monitor robots, allowing users to create new applications and share their data with others users.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Lista de Abreviaturas e Siglas

CERN	*European Organization for Nuclear Research*
DNS	*Domain Name System*
GPS	*Global Positioning System*
HTML	*HyperText Markup Language*
HTTP	*HyperText Transfer Protocol*
IETF	*Engineering Task Force*
IPv4	*Internet Protocol version 4*
IPv6	*Internet Protocol version 6*
JSON	*JavaScript Objetc Notation*
MD5	*Message-Digest algorithm 5*
M2C	*Machine to Cloud*
M2M	*Machine to Machine*
NASA	*National Aeronautics and Space Administration*
OSI	*Open Systems Interconnection*
RAM	*Random Access Memory*
ROS	*Robotic Operating System*
SLAM	*Simultaneous Localization And Mapping*
TCPROS	*Transmission Control Protocol for Robot Operating System*
TCP/IP	*Transmission Control Protocol/Internet Protocol*
UDP	*User Datagram Protocol*
UDPROS	*User Datagram Protocol Robot for Operating System*
URL	*Uniform Resource Locator*
VPN	*Virtual Private Network*

1 INTRODUÇÃO

1.1 Contexto e Relevância da Pesquisa

A Robótica Móvel atrai a atenção de muitos pesquisadores hoje em dia, porém a evolução dessa área já não depende somente de um único robô, mas sim da comunicação e colaboração entre vários deles. O carro autônomo do Google (LITMAN, 2014), os drones que já fazem entrega de compras (AMAZON, 2017) e robôs em uma linha de montagem necessitam de interação para realizar o trabalho com mais segurança e agilidade. Com isso em mente, foi criada a Robótica na Nuvem que une a Robótica Móvel com o processamento na nuvem, externo aos robôs. Retirar o processamento dos robôs e enviá-lo para a nuvem facilita a comunicação entre eles, pois permite um processamento central enquanto todos os robôs se comunicam entre si (LORENCIK; SINCAK, 2013).

Juntamente com a Robótica na Nuvem surgiu a necessidade de controlar e manipular robôs a distância, levando em consideração que algumas tarefas poderiam ser realizadas de casa. Nem sempre o deslocamento que o usuário necessita realizar para ir ao local onde o robô se encontra é vantajoso. Na verdade, esse deslocamento possui várias desvantagens pois pode demandar muito tempo ou até mesmo ser geograficamente inviável. Para tanto, é necessário uma aplicação simples e que permita que um robô possa ser controlado a distância.

Para auxiliar os usuários a criarem aplicações de monitoramento e controle de robôs foi desenvolvida a ferramenta ROS (*Robot Operating System*). Um de seus benefícios é ser modular, ou seja, é possível utilizar somente o necessário para a aplicação desejada. Além de diminuir o espaço ocupado em disco, isso evita que todo o sistema pare de funcionar ao mesmo tempo, pois seu funcionamento é baseado em nós independentes. Além disso o ROS permite que seja realizada a integração entre vários dispositivos diferentes, é grátis e possui uma comunidade grande e colaborativa, vantagens que levaram o ROSRemote a ser criado baseado em ROS.

A maior desvantagem do ROS é não permitir que aplicações sejam utilizadas na internet. Baseado nesse problema foi criado o ROSRemote, uma ferramenta que permite controlar robôs pela internet de forma simples e rápida. O ROSRemote é considerado um *framework*, pois permite tanto que o usuário utilize as funções já implementadas quanto desenvolva suas próprias aplicações que irão funcionar utilizando um recurso que se encontra na nuvem. Com a utilização do ROSRemote o usuário não necessita realizar quase nenhuma configuração, apenas instalar algumas dependências simples. Essas dependências, na maioria dos casos, podem já estar instaladas na máquina onde o ROSRemote será executado, pois são utilizadas para várias outras aplicações.

Antes do desenvolvimento do ROSRemote já existiam algumas ferramentas de propósitos gerais utilizadas para realizar essa tarefa, como o SSH e a VPN, porém elas não foram desenvolvidas especificamente para o controle de robôs. Apesar de serem bastante utilizadas, elas possuem algumas limitações e necessitam que o usuário realize uma pré configuração, o que nem sempre é possível. Isso se deve ao fato que são necessários acessos a recursos que nem sempre estão disponíveis ao usuário comum, como senhas de roteadores. Além disso, não é possível utilizar o SSH e a VPN caso o usuário esteja trabalhando com uma rede 3G/4G, pois nesse tipo de rede não é possível que realizar o redirecionamento de portas, necessário para que ambas as ferramentas funcionem (KOUBAA *et al.*, 2017).

Além de ferramentas de propósitos gerais também foram criadas algumas de propósitos específicos, como o *Rapyuta* (MOHANARAJAH *et al.*, 2015a) e o *ROSLink* (KOUBAA *et al.*, 2017), que permitem que o usuário utilize o ROS para controle e monitoramento de robôs através da internet. Apesar de úteis, essas ferramentas são de difíceis instalação e utilização. Primeiramente, para poder instalá-las é necessário possuir algumas dependências que acarretam em vários erros quando são acessadas. Além disso é necessário que o usuário memorize algumas estruturas de mensagens bastante complexas para que possa desenvolver suas próprias aplicações.

Além dos problemas já citados, realizar as conexões nas ferramentas similares ao ROSRemote é uma tarefa complicada e, no caso do ROSLink, ainda não é abordada de maneira eficiente, dificultando a utilização. Para facilitar essa conexão entre robôs e máquinas que irão se comunicar pelo ROSRemote é utilizado o *framework SpaceBrew*. Com ele é possível que o usuário conecte vários dispositivos por meio de uma interface gráfica. Essa interface contém o nome das aplicações bem como o endereço IP de onde elas se encontram e o usuário conecta os clientes com a utilização do mouse. Assim, é possível afirmar que o ROSRemote é um *framework* que além de auxiliar o usuário a enviar dados para *Masters* remotos, permite que as conexões sejam realizadas rapidamente.

Após a criação do ROSRemote, ele e o *framework SpaceBrew* serão avaliados buscando responder as seguintes perguntas chave para a pesquisa: 1. A ferramenta ROSRemote pode ser utilizada para controle e monitoramento de robôs de forma remota? 2. A utilização do *framework SpaceBrew* é interessante nesse caso? 3. A ferramenta ROSRemote é melhor do que as opções já existentes? A seção 1.2 aborda os objetivos principais desse trabalho.

1.2 Objetivos

1.2.1 Objetivo Geral

O principal objetivo deste trabalho é criar uma aplicação que permita o controle e monitoramento de robôs através da rede com o auxílio do ROS (*Robot Operating System*).

1.2.2 Objetivos Específicos

Os objetivos específicos do trabalho são:

- Avaliar o desempenho e a viabilidade do ROSRemote, indicando as vantagens e desvantagens de se utilizar o servidor gratuito do *SpaceBrew* (que se encontra na nuvem) e de se utilizar um servidor criado dentro da rede da UNIFEI;
- Descrever ferramentas existentes e utilizadas para realizar a mesma função do ROSRemote;
- Comparar com o ROSRemote as ferramentas mais comuns utilizadas para o controle de robôs, apontando as vantagens e desvantagens de cada uma delas;
- Verificar se o *SpaceBrew* é uma ferramenta rápida e viável que pode ser utilizada para conectar elementos no ROSRemote e em outras aplicações. Além disso serão descritas as vantagens e desvantagens do *SpaceBrew*.

1.3 Contribuições

A contribuição desse trabalho é desenvolver um pacote denominado ROSRemote que permita a qualquer usuário controlar e monitorar um dispositivo ou robô a distância. Além disso o ROSRemote permite utilizar o ROS com uma estrutura *multimaster*, com isso é possível a comunicação de múltiplas plataformas, dispositivos e aplicações diferentes de forma simples. Por fim, o ROSRemote possui outras vantagens: (1) não é necessário realizar configurações as quais o usuário não possua recursos suficientes para completar; (2) realizar e desfazer conexões em tempo de execução; (3) permite que o sistema seja facilmente escalável.

Outra contribuição importante deste trabalho é indicar se o *SpaceBrew* é um *framework* simples e rápido como promete ser. Apesar de existirem alguns trabalhos que o utilizam, nenhum deles possui nenhuma conclusão sobre essa ferramenta.

1.4 Estrutura do Trabalho

A fim de cumprir os objetivos propostos, este trabalho foi dividido em cinco capítulos. A fundamentação teórica é abordada no capítulo 2, tratando das principais ferramentas utilizadas neste trabalho, mostrando onde esse trabalho se encaixa tanto na área da robótica móvel quanto na área de robótica na nuvem e descrevendo algumas ferramentas similares ao ROSRemote.

O capítulo 3 descreve como a aplicação foi criada, descrevendo o código fonte utilizado para ela, mostrando o funcionamento da mesma e do pacote ROS criado a partir dela.

O capítulo 4 demonstra como foram feitos os testes para validar a pesquisa e responder as perguntas citadas na seção 1.1. Além disso mostra também um comparativo entre os testes com o ROSRemote e as ferramentas similares a ele, indicando as vantagens e desvantagens de cada uma delas. Por fim, o capítulo 4 também mostra os resultados dos testes e verifica a viabilidade da aplicação.

O capítulo 5 retoma as vantagens e desvantagens de cada ferramenta e, em seguida, apresenta as considerações finais e algumas sugestões para trabalhos futuros relacionados a esse tema.

2 FUNDAMENTAÇÃO TEÓRICA

Esse capítulo apresenta uma fundamentação teórica para a realização do trabalho, indicando onde este se encaixa na literatura e qual problema ele busca resolver. Além disso, o capítulo também detalha as ferramentas necessárias para o desenvolvimento do trabalho. Por fim, são descritas algumas ferramentas que realizam o mesmo trabalho que o ROSRemote e como elas funcionam.

As seções 2.1, 2.2 e 2.3 mostram onde esse trabalho se encaixa na literatura, descrevendo alguns campos de pesquisas e porque essas áreas vêm sendo muito difundidas. As seções 2.4.1, 2.5.1 e 2.6.1 mostram todas as ferramentas utilizadas no trabalho e o motivo delas terem sido escolhidas. Por último, as seções 2.7.1.1, 2.7.1.2, 2.7.2.1 e 2.7.2.2 mostram algumas ferramentas similares a esse trabalho e como elas são inseridas no contexto da robótica.

2.1 Computação na Nuvem (*Cloud Computing*)

A Computação na Nuvem é um área que tem se difundido ultimamente, principalmente pelo fato de garantir recursos quase ilimitados e baixo custo (ZHANG *et al.*, 2010).

Segundo Mell *et al.* (2011) a computação na nuvem é um modelo que permite acesso de de qualquer local e sob demanda a uma reserva de recursos compartilhados e configuráveis que podem ser rapidamente fornecidos e liberados com um esforço de gerenciamento mínimo. Isso acontece devido ao fato de que os recursos são alocados sob demanda, quando um usuário necessita deles, eles são reservados e quando a utilização é finalizada, os recursos podem ser transferidos para outro usuário. Dessa forma, um mesmo recurso é alocado várias vezes por vários usuários diferentes, diminuindo custos.

Geralmente, a arquitetura da computação na nuvem pode ser dividida em quatro camadas de acordo com (ZHANG *et al.*, 2010): hardware, infraestrutura, plataforma e aplicação.

A camada de hardware é responsável pelos recursos físicos da nuvem, como os data centers e servidores. É nessa camada que se encontram os processadores, memórias e sistemas de resfriamento. Dentre os vários problemas que essa camada pode gerar, os principais são: a quantidade de recursos que devem e podem ser oferecidos, tolerância a falhas (um servidor deve assumir o trabalho no caso de outro parar de funcionar) e gerenciamento dos recursos para resfriamento, já que se deve levar em conta que um processador com maior carga de trabalho necessita de mais resfriamento que um ocioso.

A camada de infraestrutura é responsável por criar o armazenamento de recursos chamado de *pool* ou piscina, pois simula uma piscina que contém todos os recursos que

são acessados pelos usuários sempre que necessário. Para dividir os recursos físicos desse *pool* entre os usuários, são utilizadas tecnologias de virtualização.

A camada de plataforma é montada sobre a de infraestrutura e consiste no sistema operacional e *frameworks*. Ela é criada exclusivamente para que as aplicações não sejam diretamente feitas sobre a camada de infraestrutura, melhorando o desempenho.

A última das camadas consiste nas aplicações em si. Uma característica importante desse nível é permitir que ocorra, automaticamente, um aumento de recursos para obter melhor desempenho a um menor custo. O servidor do *SpaceBrew* funciona sobre essa camada. A distribuição das camadas pode ser visualizada na Figura 2.1.1.

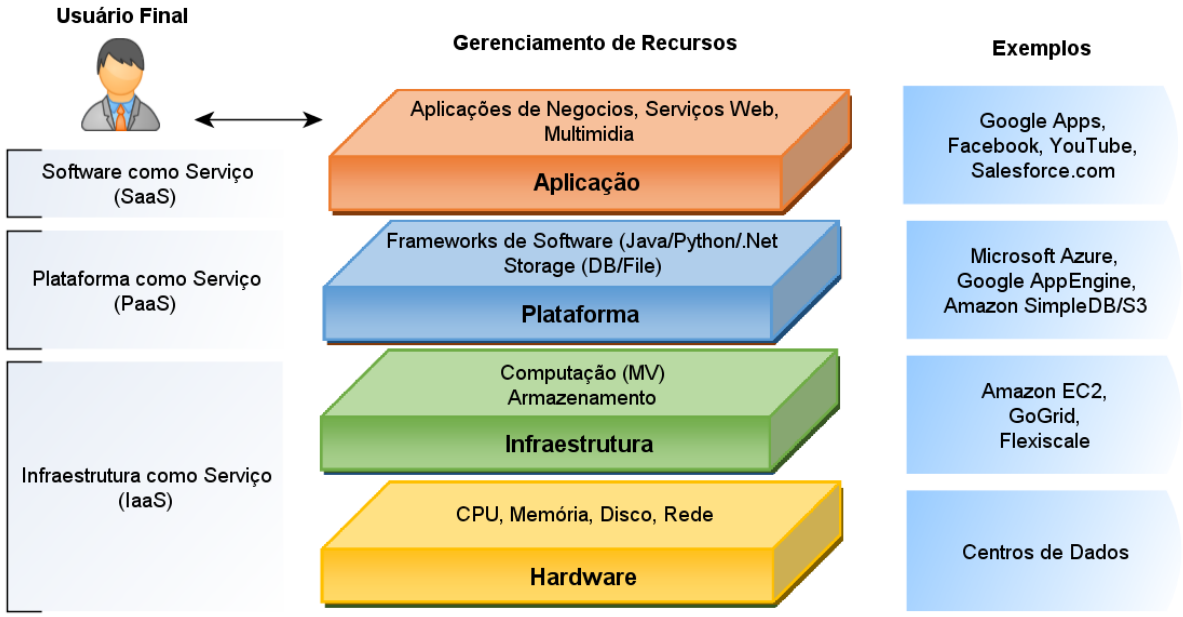

Figura 2.1.1 – Camadas de uma Aplicação na Nuvem. Fonte: (OTERO, 2013).

A computação na nuvem é divida em quatro áreas:

- *Software* como Serviço (*Software as a Service* (SaaS));
- Plataforma como Serviço (*Platform as a Service* (PaaS));
- Infraestrutura como Serviço (*Infraestructure as a Service* (IaaS)) (BYRNE *et al.*, 2017);
- *Framework* como Serviço (*Framework as a Service* (FaaS)) (RIMAL *et al.*, 2009).

Mesmo levando em conta que Byrne *et al.* (2017) descreve o FaaS como *Function as a Service* e Gunawi *et al.* (2011) utiliza essa sigla como *Failure as a Service*, no contexto desse trabalho ela é tratada como *Framework as a Service*.

A utilização de *Software* como Serviço é a mais comum na nuvem e consiste em usar a infraestrutura desta para criar aplicações que podem ser acessadas por usuários. Essas aplicações podem ser utilizadas por vários dispositivos através de uma interface leve, como

navegadores. O usuário não consegue gerenciar ou controlar a infraestrutura da nuvem e o número de conexões simultâneas é limitado em alguns casos.

A Plataforma como Serviço não é utilizada por usuários comuns, mas sim por desenvolvedores de sites ou por quem deseja disponibilizar aplicações on-line para outras pessoas. Nesse caso, o consumidor não consegue gerenciar a infraestrutura física da nuvem, porém é possível controlar quais aplicações estão disponíveis e ajustar algumas configurações.

Na Infraestrutura como Serviço, o usuário apenas não consegue gerenciar e controlar o *hardware* do serviço na nuvem, porém é possível controlar todo o resto, incluindo sistemas operacionais, portas e *firewalls*.

Por último, uma nova categoria foi adicionada recentemente na computação na nuvem para preencher uma lacuna que havia entre o SaaS e o PaaS. Essa nova categoria é denominada FaaS, *Framework* como Serviço e é a área na qual o ROSRemote se encaixa.

Como apresenta Cyfronet (2012), o *Framework* como Serviço descreve uma abordagem que não é *SaaS*, nem *PaaS*. O primeiro não possui flexibilidade suficiente e o segundo exige um esforço grande ao se criar uma nova aplicação. No caso do *FaaS*, o usuário pode criar plataformas de serviço de maneira mais rápida. O *framework* auxilia na criação da aplicação e realiza o serviço mais complexo, permitindo que o usuário se preocupe somente com a customização da aplicação. Além disso, o FaaS também é considerado um *software* pois é possível que o usuário o utilize sem ser necessário realizar configurações extras.

Seguindo a classificação descrita, esse trabalho deve ser classificado entre duas formas: no caso do usuário utilizar o servidor do *SpaceBrew* e enviar mensagens através dele, a aplicação pode ser classificada como *Software* como Serviço; no caso do usuário criar seu próprio servidor e utilizá-lo para enviar os dados, pode ser uma Plataforma como Serviço. Portanto é possível inferir que a aplicação descrita neste trabalho se encaixa na categoria de um *FaaS*.

Existem algumas características que devem ser levadas em consideração ao se disponibilizar um serviço de Computação na Nuvem e elas devem ser obrigatórias para o bom funcionamento das aplicações. De acordo com as recomendações de Mell *et al.* (2011), o modelo de Computação na Nuvem é composto por cinco características principais e todas elas são desejáveis neste trabalho:

- Serviço sob demanda: quaisquer recursos devem ser disponibilizados ao usuário sempre que ele desejar e sem a necessidade de interação humana ;
- Amplo acesso: os recursos devem estar disponíveis para vários dispositivos como *smartphones*, *tablets*, notebooks, entre outros, sempre e onde o usuário precisar;
- Reserva de recursos: os recursos computacionais são deixados em um reservatório (*pool*) e ficam disponíveis para todos os usuários. Quando aqueles são solicitados, são retirados desse *pool* e fornecidos ao usuário. Ao terminar a utilização, o recurso é devolvido e fica a espera de uma nova requisição. O usuário não sabe onde esses

recursos estão localizados, mas é importante que estejam disponíveis independentemente de onde se encontrem;

- Rápida elasticidade: a quantidade de recursos deve ser facilmente aumentada ou reduzida, caso necessário. Porém, é necessário que o usuário não perceba que os recursos são escaláveis e limitados. Para o usuário, os recursos são ilimitados e utilizáveis a qualquer momento;

- Serviço medido: o serviço na nuvem controla e otimiza os recursos que devem ser disponibilizados ao usuário. Caso o usuário não necessite de muitos recursos, ele recebe menos do que alguém que está realizando um processamento maior.

Além dessas características, existem outros assuntos que devem ser levados em consideração quando se trata de computação na nuvem, como segurança dos dados, confiabilidade e custos. Baseado nesses requisitos, as nuvens foram separadas em vários tipos, cada um dlses possui vantagens e desvantagens (MELL *et al.*, 2011).

As Nuvens Públicas são as que oferecem seus recursos como serviços para o público em geral (ARMBRUST *et al.*, 2010). Dentre todos os benefícios desse tipo de nuvem pode-se destacar que não é necessário investimento inicial em infraestrutura e todos os riscos são transferidos para os fornecedores do serviço. Como a rede não possui custos para o usuário do serviço, ela não permite controle sobre os dados nem oferece muita segurança, o que acaba prejudicando sua utilização.

Da mesma forma que existem as Nuvens Públicas também existem as Privadas, que geralmente são específicas para uso exclusivo de uma única organização (ARMBRUST *et al.*, 2010). Essas nuvens podem ser criadas e gerenciadas pelo próprio dono ou por fornecedores externos, mas independente do caso, ela é muito custosa. Nesse tipo de rede, o controle sobre todas as variáveis como desempenho e segurança é alto, sendo mais indicado para empresas que querem manter seus dados compartilhados seguros.

Visando buscar o que há de melhor entre os dois tipos de nuvens citadas, foi criada a Nuvem Híbrida. A combinação dos dois tipos de nuvens oferece flexibilidade, maior controle e segurança sobre as aplicações, enquanto facilita a escalabilidade dos serviços e o fornecimento sob demanda dos recursos. Por outro lado, a criação desse tipo de nuvem requer estudos para decidir como a divisão entre as Nuvens Públicas e Privadas deve ser feita (ZHANG *et al.*, 2010).

Além desses tipos, ainda existe a Nuvem Virtual Privada, que é uma alternativa à Nuvem Híbrida. A VPC (*Virtual Private Cloud*) é uma plataforma que funciona sobre as Nuvens Públicas e possui como principal vantagem permitir aos fornecedores criar suas próprias regras de segurança, como *firewalls*. Com isso, as Nuvens Virtuais Privadas são mais seguras, fáceis de implementar e baratas. Uma VPC é considerada uma combinação da infraestrutura de uma VPN (*Virtual Private Network*) com os recursos da computação

na nuvem, isto significa que os recursos na nuvem são alocados dinamicamente para os usuários através de uma conexão VPN (WOOD *et al.*, 2009).

Para a utilização do ROSRemote quaisquer tipos de nuvens são indicados, porém é aconselhável que sejam utilizadas as privadas por dois motivos: (1) a nuvem pública permite que qualquer pessoa consiga interceptar os dados enviados; (2) como o *SpaceBrew* não necessita de muito processamento, um servidor privado de baixo custo é suficiente para suprir as necessidades da aplicação.

Apesar de todos os estudos e avanços nessa área, ainda existem muitos desafios que requerem mais pesquisas e melhorias. Algumas das preocupações e oportunidades de pesquisas são mencionados por (ARMBRUST *et al.*, 2010):

- Disponibilidade do serviço: é esperado que o serviço esteja disponível sempre que necessário. Para solucionar o problema de disponibilidade é indispensável a realização de novas pesquisas sobre múltiplos servidores oferecendo o mesmo serviço;
- Confidencialidade dos dados: muitas empresas guardam segredos importantes em seus servidores e independentemente do tipo de dado, é necessário assegurar que eles não serão vistos por qualquer pessoa. Portanto a segurança é um tópico extremamente importante, tanto no âmbito acadêmico quanto industrial;
- Gargalos em transferência de dados: o volume de dados transferidos a cada segundo é muito alto e podem gerar gargalos. Isso ocorre devido ao fato de que muitas informações acabam sendo enviadas ao mesmo tempo gerando uma espécie de "engarrafamento" e causando atrasos nos envios. Uma solução é melhorar o hardware para o aumento do fluxo de dados, porém isso resulta em custos excessivos, visto que deve ser realizado um grande investimento em dispositivos e tecnologias de armazenamento;
- Escalabilidade: a quantidade de dados que deve ser armazenada cresce rapidamente e portanto, é necessário criar maneiras de melhorar a capacidade de armazenamento sempre que necessário;
- Problemas em sistemas distribuídos (*bugs*): a dificuldade de se remover erros nos códigos criados para serem utilizados em servidores na nuvem é enorme. Isso se deve ao fato de não ser possível que eles sejam testados em uma escala menor do que a que serão utilizados e que os problemas só sejam encontrados ao se colocar a aplicação em um ambiente real. Esse é um tópico que atrai muitas pesquisas, como o WiDS Checker (LIU *et al.*, 2007) e o PIP (REYNOLDS *et al.*, 2006) que são ferramentas capazes de localizar problemas para que eles possam ser consertados, essas ferramentas são classificadas como *debuggers*.

Apesar de todas as dificuldades que ainda devem ser levadas em consideração ao se criar uma aplicação na nuvem, esse tipo de sistema vem crescendo ultimamente e é a

tendência para o futuro da computação. Para comprovar isso, foram feitas buscas em alguns dos principais bancos de dados utilizando as palavras chave *"cloud computing"* e foi constatada a grande utilização dessa tecnologia. Na base Science Direct essa busca retornou mais de 84 mil artigos, na IEEEXplore foram encontrados 47 mil, a Scopus retornou 60 mil resultados e o Springer Link foi a que retornou mais resultados, cerca de 110 mil. Entre esses resultados se encontram artigos, periódicos e livros. Na área da robótica a computação na nuvem também está presente e é explicada na seção 2.3.

Entre os trabalhos mais novos e/ou mais citados está o trabalho de Pandey *et al.* (2010) que descreve que a forma como o usuário contrata os serviços na nuvem hoje em dia não é baseada em transferência de dados e custo de execução. Devido a esse fato ele apresenta uma ferramenta que permite medir o custo de execução e de transmissão de dados. Além desse trabalho relacionado a custos da computação na nuvem também foram encontrados trabalhos sobre desempenho de nuvens computacionais como o Aneka (VECCHIOLA *et al.*, 2009) e o CloudSim (CALHEIROS *et al.*, 2011). Ainda existem trabalhos sobre segurança, que é um ponto bastante importante e apresentado em alguns trabalhos como os de Gai *et al.* (2017) e Li *et al.* (2017).

Entre as várias pesquisas sobre computação na nuvem, uma das mais novas áreas que cresce bastante e é a que possui maior quantidade de pesquisas atualmente, é a *Big Data*, visto nos estudos de Assunção *et al.* (2015) e de Yang *et al.* (2017). Além destas áreas, existem várias outras que utilizam a computação na nuvem. Como exemplos podem ser citados educação, que pode ser vista em Sultan (2010), jogos, como no trabalho descrito por Chen (2015) e em ciências e no auxílio a pesquisas como nos estudos de Keahey *et al.* (2008) e de Sadooghi *et al.* (2017). A próxima seção disserta sobre a robótica móvel e como ela vem sendo utilizada, pois, da combinação entre computação na nuvem e a robótica móvel, surgiu a computação na nuvem, que é a área na qual este trabalho se encaixa e é explicada na seção 2.3.

2.2 Robótica Móvel

A robótica já é conhecida e utilizada há muito tempo. Desde muito tempo atrás a humanidade passou a tentar desenvolver formas mecânicas que pudessem reproduzir trabalhos humanos, no entanto o termo "robô" e "robótica" apareceram em uma parte recente da história, há cerca de 100 anos. Nehmzow (2012) e Sánchez-Martín *et al.* (2007) dissertam sobre o fato do termo "robô" ter sido utilizado pela primeira vez por Karel Capek em 1921 em sua peça *Rossum's Universal Robot* (CAPEK *et al.*, 1920). Porém, o termo "robótica" só foi utilizado pela primeira vez 20 anos depois, no livro "Eu, Robô" de Isaac Asimov, mais especificamente no conto "Andando em circulos" ou *"Runaround"*, (ASIMOV, 1942). Apesar da robótica já ser utilizada antes dos contos de Asimov, seu apogeu coincidiu com a publicação desse conto.

Durante o mesmo período em que "Andando em Círculos" foi lançado alguns autores dizem que foi criado o primeiro robô móvel (WALTER, 1950), chamado de *"Tortoise"* ou *"Machina Speculatrix"*. Por outro lado, existem autores que acreditam que o primeiro robô móvel criado foi o chamado *"Shakey"* (NILSSON, 1984), que apesar de ser documentado somente em 1984 já existia em 1966 e foi utilizado para testes a partir dessa data até 1972.

A definição de robôs móveis feita por Nehmzow (2012) diz que são robôs que podem se movimentar através de locomoção e utilizando-a, é possível inferir que Walter (1950) foi o criador do primeiro robô móvel, com o *"Speculatrix"* que se movia como uma tartaruga. No entanto, *"Shakey"* foi o primeiro robô móvel criado para propósitos gerais, capaz de se movimentar, desviar de obstáculos, mover objetos e abrir portas.

Apesar de todo o avanço na construção de robôs, ainda existem muitos pontos chave que precisam ser estudados e melhorados. Siegwart *et al.* (2011) descrevem alguns deles, principalmente com relação a locomoção de robôs: estabilidade, forma de contato com o solo (rodas ou pés) e ambiente no qual é possível a movimentação (terra, água ou ar). De acordo com eles, existem várias formas de um robô se locomover, seja com rodas, pés, patas ou caudas.

Entre os robôs que se movimentam com rodas, existem alguns tipos de acordo com (SIEGWART *et al.*, 2011), os de duas, três e quatro rodas. Além disso, eles também podem ser subdivididos em duas categorias básicas, omnidirecional e unidirecional, o primeiro pode se movimentar em todas as direções, incluindo para os lados e o segundo se movimenta somente para a frente. É possível encontrar mais informações sobre os tipos de robôs com rodas, nos trabalhos dos autores Siegwart *et al.* (2011) e Tzafestas (2013). Ambos os artigos dão vários detalhes sobre esse tipo de movimentação, como o tipo de roda, se são móveis ou fixas, se possuem tração ou servem apenas para direcionar e apoiar o robô. Apesar dos autores não descreverem, ainda existem robôs que possuem seis rodas, como os robôs da Missão *Mars Rover*. Além da quantidade de rodas, esses robôs possuem uma suspensão bastante avançada que permite maior estabilidade e capacidade de escalar obstáculos, como pedras. Essa suspensão é chamada de *Rocker-Bogie* e pode ser utilizada para melhorar a estabilidade de quaisquer robôs (HARRINGTON; VOORHEES, 2004).

Robôs com rodas possuem uma estabilidade muito alta, porém a maioria deles possui várias dificuldades ao escalar obstáculos (RAIBERT, 1986). Para resolver esses problemas, cientistas iniciaram pesquisas com robôs que possuem pernas e que são divididos em robôs de duas, quatro e seis pernas. A maioria das pesquisas são feitas com a utilização de robôs com duas pernas, os chamados humanoides, porém também existem muitas pesquisas com robôs de quatro e seis pernas, pois eles são capazes de ultrapassar obstáculos que outros tipos não conseguem, como subir e descer escadas (TODD, 2013).

Ainda existem dois tipos menos citados de robôs que são os "rastejantes", que se movimentam como uma cobra e os "aquáticos", capazes de nadar imitando o movimento

de peixes e sapos (CHERNOUSKO, 2017). A vantagem do primeiro é que ele é dividido em pequenos módulos e caso ocorra a falha de um deles, o robô ainda continuará com sua movimentação normalmente. Além disso, é possível que ele alcance locais que outros robôs não conseguem, como pequenos buracos na terra (WRIGHT *et al.*, 2007). Já os robôs aquáticos têm a vantagem de se locomover rapidamente embaixo da água e auxiliam na detecção de poluição (MCGOVERN *et al.*, 2008; HIRATA *et al.*, 2000).

Utilizando esses tipos de robôs várias aplicações tem sido criadas tanto para estudos de Inteligência Artificial quanto para lazer, educação e serviços domésticos. O primeiro caso era o uso mais comum para robôs em meados da década de 80 (BRADY, 1985a), porém, com os recentes avanços na área, começaram a ser criados robôs de baixo custo para auxiliar em tarefas domésticas, lazer e educação de crianças e adolescentes.

Lund e Nielsen (2002) criam e utilizam em seu trabalho a palavra *edutainment*, mistura das palavras educação (*education*) e entretenimento (*entertainment*) e mesmo o artigo sendo relativamente antigo, essa prática ainda é uma tendência nos dias de hoje. Ela consiste em criar brinquedos divertidos que auxiliam na educação de crianças. Nesse mesmo artigo, Lund e Nielsen (2002) citam como exemplo os famosos *Tamagotchis* que não são considerados robôs móveis, mas que ensinavam as crianças a cuidar de um animal de estimação. Além disso também existem outros brinquedos como o *Robota* (BILLARD, 2003a) e recentemente, vários brinquedos produzidos pela empresa WowWee (WOWWEE, 2018).

Vários outros brinquedos menos conhecidos também foram criados, alguns deles utilizados para auxiliar crianças com algum tipo de incapacidade motora ou mental (VALADÃO *et al.*, 2017). De acordo com (BILLARD, 2003b) existe uma classe de brinquedos que foi estudada em 2003 chamada "Robota" capazes de imitar seres humanos e aprender a realizar algumas tarefas pequenas, além disso auxiliavam crianças a desenvolver algoritmos de aprendizado.

Contudo a robótica não vem sendo utilizada somente para diversão, já existem robôs que auxiliam em tarefas domésticas, como o robô aspirador de pó que já é comum em muitas casas e faz bastante sucesso em vários países (BOGUE, 2017) apesar de ainda não ser muito comum no Brasil. No entanto essa ideia não é nova, desde 2000 já haviam pesquisas nessa área, como o *Mr. Helper*, capaz de manusear alguns objetos com o auxílio de um ser humano (KOSUGE *et al.*, 2000).

Uma busca por artigos sobre robótica móvel mostra que a maioria das pesquisas nessa área são relacionadas à inteligência artificial. Isso se deve ao fato de que a robótica é um desafio a essa área, pois como descreve (BRADY, 1985b), envolve objetos no mundo real e leva em consideração que uma inteligência artificial mal desenvolvida pode colocar em risco a vida de pessoas que necessitarem dos robôs. Essas pesquisas se iniciaram há muito tempo e existem até hoje em várias áreas como na indústria (LU *et al.*, 2017) e na medicina (HAMET; TREMBLAY, 2017). Esses estudos vem sendo realizados não somente

na inteligência lógica, mas também na emocional, como é o caso do robô que auxilia as compras (BERTACCHINI *et al.*, 2017). Com as pesquisas nessa área de inteligência é possível melhorar a percepção das emoções humanas pelos robôs, além de permitir que eles também exprimam as suas.

A robótica móvel não abrange somente pesquisas sobre emoções. Uma área que vem evoluindo muito é a criação de armas para auxiliarem exércitos e evitarem perdas de soldados e/ou civis. Outro ponto que leva à criação desses tipos de armas é que um robô autônomo possui um tempo de reação menor do que um ser humano treinado (KASTAN, 2013). O fato é que essas armas são cada vez menores e mais letais e novas tecnologias e estudos apontam para a utilização de enxames de robôs para ser utilizado em uma guerra, ao invés de um único robô. Nesse caso, vários mini robôs são enviados em conjunto para realizar uma tarefa, isso os torna mais letais e precisos, pois caso alguns deles sejam destruídos, é possível que os restantes completem a missão. Esse é um assunto bastante controverso, alguns autores defendem essa prática, porém outros a abominam.

Existem vários tipos de robôs e esta pesquisa do ROSRemote pode ser utilizada para qualquer um que consiga trabalhar em conjunto com o ROS. A união das duas áreas descritas, computação na nuvem e robótica móvel, deu origem a outra área que ficou conhecida como robótica na nuvem, e que é descrita na próxima seção.

2.3 Robótica na Nuvem (*Cloud Robotics*)

Atualmente o conceito de Robótica na Nuvem (*Cloud Robotics*) vem sendo amplamente utilizado. Essa tecnologia consiste em utilizar recursos da nuvem para comunicação e processamento, permitindo que os robôs utilizados possuam menos recursos internos, diminuindo custos e facilitando a configuração.

De acordo com Hu *et al.* (2012), as aplicações de robótica podem ser classificadas em duas categorias: robôs tele operados ou multi-robôs. No primeiro caso, o robô é operado a distância por um ser humano, como exemplo pode ser citado o Mars Rover (NASA, 2016), veículo não tripulado para exploração de Marte. Mas nos dias de hoje, a robótica na nuvem é mais comumente utilizada para auxiliar o processamento em sistemas de multi-robôs, pois como é necessário que todos tomem decisões em conjunto, é mais vantajoso utilizar um sistema central para coordenar toda a comunicação e as enviar informações. Inaba (1997) já havia tido essa ideia de separar o corpo e o cérebro do robô a fim de que pudessem ser utilizadas técnicas de Inteligência Artificial que antes só eram possíveis em simulações.

No modelo proposto por Hu *et al.* (2012) existem dois tipos de comunicação, a M2M (*Machine-to-machine*) de máquina para máquina, e a M2C (*Machine-to-Cloud*) de máquina para nuvem. A comunicação M2M pode ser realizada dentro de uma rede local com todas as máquinas conectadas a um mesmo roteador e compartilhando a mesma

rede. Essa comunicação em rede local não pode ocorrer no caso da comunicação M2C, nela ocorre uma reserva (*pool*) de recursos computacionais que são alocados de forma dinâmica em tempo de execução. Um diagrama explicando o modelo é apresentado na Figura 2.3.1.

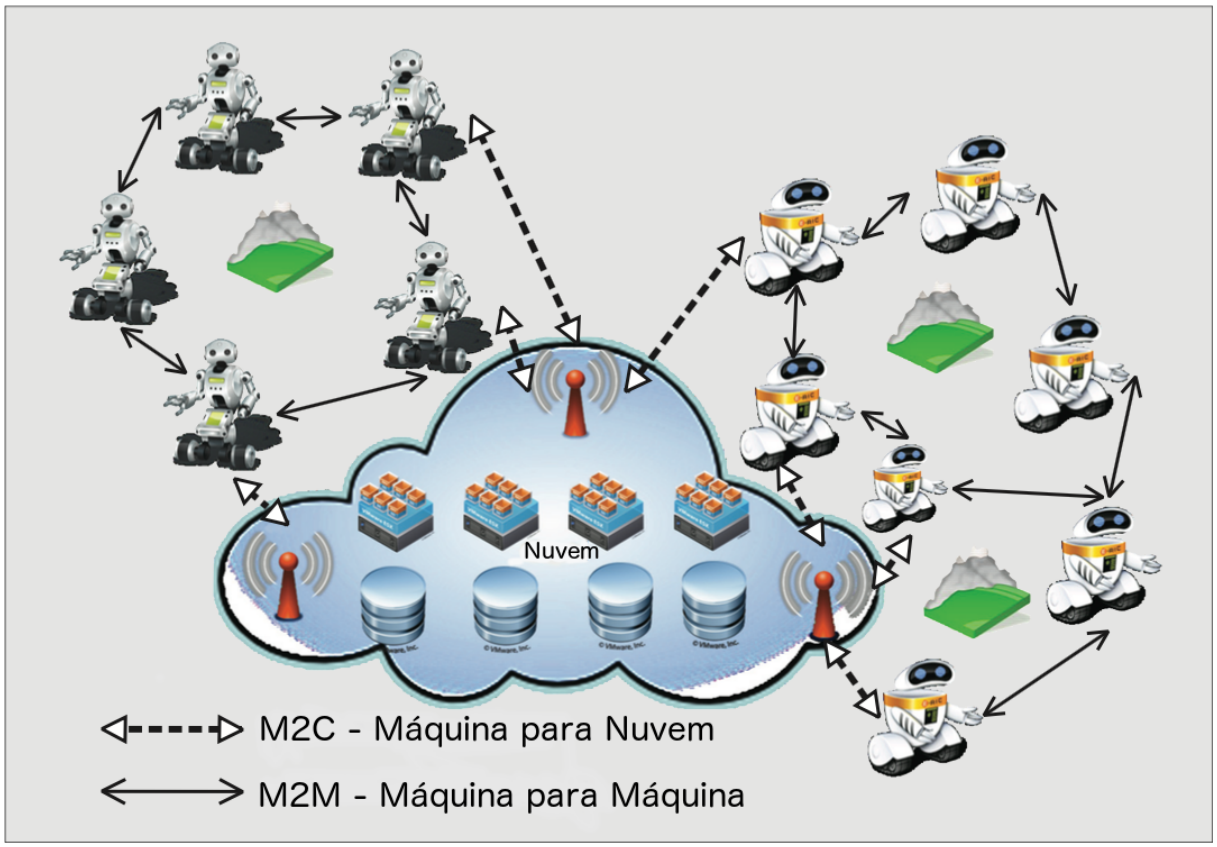

Figura 2.3.1 – Robôs interconectados dividindo recursos locais e remotos. Fonte: Adaptado de (HU *et al.*, 2012).

Na comunicação M2C o processamento é realizado na nuvem. Dessa forma os dispositivos recuperam os dados necessários para a tarefa e os enviam para os computadores remotos. Como esses possuem melhores recursos (processadores e memória disponível) realizam os cálculos necessários rapidamente e enviam a resposta com os comandos necessários para que seja possível a conclusão do objetivo. Para tanto, é possível utilizar serviços de hospedagem disponíveis online ou criar um servidor próprio capaz de suprir essas necessidades. Essa tecnologia de robótica na nuvem pode ser utilizada de várias formas diferentes e para diferentes aplicações, contudo as mais comuns são para robôs autônomos móveis e robôs industriais.

O carro autônomo do Google (MARKOFF, 2010) faz uso da robótica na nuvem em toda sua comunicação, pois ele precisa traçar a rota, desviar de obstáculos e localizar outros veículos a fim de evitar acidentes. Além disso, ele utiliza sensores, GPS (*Global Positioning System* ou Sistema de Posicionamento Global) e câmera para monitorar sua posição. Todo esse processamento deve ser feito em tempo hábil para que o comando seja

dado a tempo suficiente de evitar colisões, então é mais vantagem utilizar o processamento em nuvem. Apesar de ser uma aplicação muito interessante da robótica na nuvem, esse carro ainda precisa de muita pesquisa até chegar ao ponto de poder ser comercializado (POCZTER; JANKOVIC, 2014).

De acordo com Almada-Lobo (2016), acredita-se que hoje as indústrias estejam a beira da quarta revolução industrial e com a chegada da chamada internet das coisas (termo utilizado para a era atual em que os dispositivos eletrônicos estão sempre conectados à internet) o mundo real e virtual estão cada vez mais se unificando. É possível que em um futuro próximo os clientes possam personalizar seus produtos diretamente, permitindo que cada produto seja único e aproximando o cliente do robô que o fabrica. Não haverá um mediador entre o que o cliente deseja e a linha de montagem, o cliente personaliza o produto e a aplicação comanda o robô montador automaticamente. Como descrito em Zhou *et al.* (2015), a ideia é transformar produtos populares em personalizados e permitir que os usuários possuam a sensação de como pode ser divertido criar seus produtos, aumentando as vendas.

Outro uso que tem se tornado muito comum para a robótica na nuvem são os drones que entregam compras utilizando apenas as coordenadas de entrega (WELCH, 2015; PANDIT; POOJARI, 2014). O drone leva a compra e volta para o local de origem sem ser necessário algum tipo de interação humana. Caso isso se torne um hábito, será necessário a comunicação entre eles para evitar colisões.

A robótica na nuvem é uma tecnologia que tende a crescer muito e ser muito utilizada no futuro, porém no momento em que se encontra possui muitos desafios. O principal e talvez o mais difícil de lidar, seja a velocidade das conexões, pois o robô precisa confiar na aplicação na nuvem e essa deve enviar os comandos rapidamente. Se a conexão não transferir os dados na velocidade desejada podem ocorrer gargalos e o resultado será uma aplicação mais lenta e menos inteligente (REN, 2011).

Para uma aplicação que não necessite de resposta em poucos milissegundos, a robótica na nuvem é uma alternativa, porém caso seja necessário que uma decisão seja tomada em um curto período de tempo, o atraso nas conexões pode colocar em risco toda a operação. Basta supor que um robô seja controlado remotamente por um ser humano e este observa algum obstáculo que possa danificar a máquina. Podem ocorrer alguns contratempos, como o comando enviado ser recebido somente após o acidente já ter ocorrido. Nesse sentido, ainda é necessário muito estudo para que essa tecnologia seja 100% viável. Além disso também existem outros problemas e são descritos em Wan *et al.* (2016a) como:

- Alocação de recursos: é necessário que o processamento seja feito na máquina mais próxima do robô, com isso o desempenho é melhorado;
- Interação entre robô e nuvem: dispositivos e sensores de diferentes fabricantes podem gerar dados diferentes, obrigando que a interface da nuvem saiba interpretar uma

gama de estruturas de dados (WAN *et al.*, 2016b);

- Segurança: caso os dados sejam confidenciais, é necessário que a segurança dos mesmos seja primordial.

Nesse trabalho não serão descritas formas de lidar com os problemas citados, portanto todos os desafios enfrentados pela robótica na nuvem em geral também ocorrerão aqui. Além disso a robótica na nuvem é, dentre todas as tecnologias estudadas, a que mais se assemelha ao trabalho descrito nesse documento e por esse motivo é a base para o desenvolvimento da dissertação.

Uma busca na literatura revelou várias aplicações que utilizam a robótica na nuvem, porém as mais importantes são destacadas nos próximos parágrafos. Todos os autores descrevem que ocorreram problemas com a internet durante os estudos como *delays*, problemas com segurança e recursos. Com isso é possível inferir que esses problemas são enfrentados em todos os trabalhos e não é uma desvantagem específica deste.

Entre todas as aplicações encontradas a mais completa e utilizada é conhecida por *RoboEarth*. Sob a premissa de que os seres humanos são mais inteligentes quando compartilham informações, desenvolvedores criaram uma plataforma que permite que robôs também possam realizar essa partilha (WAIBEL *et al.*, 2011). A ideia é criar uma base de dados acessível por todos através da internet que mantém todo o conhecimento adquirido pelos robôs que a utilizam (ZWEIGLE *et al.*, 2009). Esse conhecimento pode ser tanto do ambiente em que o robô se encontra (como mesas, cadeiras e outros obstáculos) quanto sobre a execução de uma tarefa. O robô que adquire esse conhecimento previamente salvo na base de dados, é capaz de realizar a ação de forma mais rápida e efetiva do que o robô anterior a ele, mesmo que ele nunca tenha realizado a tarefa. Após essa ação é possível enviar novos dados sobre o local para que o próximo robô seja sempre capaz de realizar a tarefa de forma mais eficaz que o último.

De acordo com Zweigle *et al.* (2009), a melhor maneira de descrever o *RoboEarth* é como uma ferramenta capaz de dividir o conhecimento entre robôs em todo o mundo ao criar uma base de dados de conhecimento sobre o mundo e seus objetos. As descrições das ações necessárias para se completar uma tarefa são chamadas de receitas e em geral, funcionam para vários robôs mesmo que eles possuam diferentes *hardwares*. Isso significa que independentemente do dispositivo que o usuário esteja utilizando, existe a chance dele poder utilizar o conhecimento adquirido por outro dispositivo para realizar o trabalho que necessita.

Waibel *et al.* (2011) descreve um exemplo muito interessante da utilização do *RoboEarth*, eles assumem que exista um robô que trabalha em um hospital e que precisa realizar a tarefa de levar um copo de água para um dos enfermos. A fim de realizar esse processo, ele necessita completar uma série de tarefas sequenciais.

1. Encontrar a garrafa com água;

2. Chegar perto da garrafa e pegá-la;

3. Localizar o paciente;

4. Entregar a garrafa.

Durante a realização das tarefas, o robô mantém uma descrição (*log*) de tudo que executou, do mapa do local onde ele se encontra e onde se encontram os objetos. Além disso, caso o robô não possua sucesso em suas tarefas, ele pode pedir ajuda para um ser humano e armazenar todo o novo conhecimento. O robô realiza um *upload* de todo o conhecimento armazenado para uma base de dados compartilhada ao completar as tarefas. Caso outro robô necessite realizar a mesma tarefa em um ambiente diferente, mesmo que o conhecimento do local em que o primeiro robô se encontrava não seja suficiente para o segundo completar a tarefa, o reconhecimento da garrafa de água e da cama onde se encontra o paciente já fornece um auxílio ao robô. Basicamente esse é o funcionamento do *RoboEarth*, que surgiu em 2010, mas somente a partir de 2015 começou a ser utilizado na comunidade acadêmica.

Além do *RoboEarth* outras aplicações vêm sendo criadas utilizando a robótica na nuvem. Como já descrito, a robótica na nuvem é a base para que as empresas avancem para a chamada quarta revolução industrial com a Indústria 4.0 (ALMADA-LOBO, 2016). Ela se caracteriza pela troca da produção de produtos padronizados para todos os consumidores, para produtos que podem ser personalizados de acordo com as características e desejos de cada comprador. Vale ressaltar que a princípio, a indústria 4.0 foi implementada na Alemanha, mas já vem sendo utilizada em outros países, pois é uma área bastante promissora.

Além das duas grandes e importantes utilizações da robótica na nuvem citadas até aqui, a *RoboEarth* e a indústria 4.0, também existem outras aplicações simples que utilizam essa poderosa ferramenta. Como exemplo é possível citar Kamei *et al.* (2012) e Manzi *et al.* (2017) que descrevem dois trabalhos distintos para auxiliar pessoas idosas a terem uma vida mais independente, sem a necessidade de um acompanhante humano a todo momento. Esses robôs são capazes de reconhecer comandos simples como ir a algum lugar e lembrar eventos, como a hora de administrar algum medicamento. Nesses casos, a robótica na nuvem é utilizada simplesmente para facilitar as tarefas do robô, seja de forma simples como utilizar a API do Google para ter acesso a agendas ou reconhecimento de voz, até a utilização de bases de dados complexas para deslocamento e identificação de objetos.

Outro uso bastante comum da robótica na nuvem é a cooperação entre robôs, também conhecido por sistemas multi-robôs. Alguns estudos já foram feitos nessa área e em diferentes locais (indústrias, casas, ruas), como os estudos de Turnbull e Samanta (2013) e mais recentemente Cardarelli *et al.* (2017). Em casos onde o robô precisa executar a tarefa em uma região pequena, apenas um robô é suficiente para ter êxito rapidamente,

porém em grandes áreas são necessários mais de um robô e eles devem trabalhar em conjunto.

Cardarelli *et al.* (2017) descrevem um sistema de cooperação que recupera dados de diferentes fontes. Esses dados permitem que os AGVs (*Automated Guided Vehicles* ou Veículos Automaticamente Guiados) sejam capazes de coordenar seus movimentos de maneira otimizada, evitando congestionamentos e obstáculos. Para isso, toda a informação recuperada dos sensores é enviada para um sistema na nuvem que pode ser acessado por quaisquer robôs. Quando uma tarefa é criada, esse sistema também é responsável por realizar o cálculo e o planejamento do caminho mais rápido para alcançar o destino.

Outra aplicação bastante explorada é a de localização e mapeamento de robôs, conhecido por SLAM (Simultaneous Localization And Mapping) (HU *et al.*, 2012). Existem várias ferramentas e pesquisas que auxiliam na localização de robôs, porém grande parte delas funciona somente para um robô. Para aumentar a velocidade e capacidade de processamento e tomada de decisão é necessária a utilização de vários robôs, portanto é importante que uma central de processamento capaz de concentrar todos os dados recebidos dos diversos robôs seja criada. Para tanto, Doriya *et al.* (2017) descreve algumas ferramentas capazes de permitir a utilização de SLAM em sistemas multi-robôs, todos com a utilização de robótica na nuvem. Nesse estudo foram comparados três *frameworks* e foi concluído que o *Rapyuta* permite um rápido aumento de suas funcionalidades por ser *open source*, o DAVinci (ARUMUGAM *et al.*, 2010) permite maior paralelismo e o C2TAM apresentado por Riazuelo e onthers (2014) é melhor utilizado em condições onde é necessário armazenar grande quantidade de dados e a largura de banda é baixa.

Outra aplicação bastante interessante encontrada na literatura é a de robôs que auxiliam clientes em suas compras (BERTACCHINI *et al.*, 2017). Essa aplicação tenta fazer com que o robô seja capaz de perceber as emoções e interesses dos clientes e tente transmitir suas próprias emoções a fim de auxiliar os clientes a tomarem decisões. Nesse projeto foi utilizado o robô NAO juntamente com um sistema de aprendizado que permite que os robôs adquiram noos conhecimentos a cada interação com um novo cliente. Dessa forma, com os próximos clientes, o robô é capaz de identificar as emoções e auxiliá-los de forma mais rápida e precisa.

Apesar de todas as aplicações já existentes, ainda falta na literatura algo que permita que o usuário crie suas aplicações da maneira que preferir e só utilize o *framework* para enviar e receber as informações. Além disso, é importante que o desenvolvedor sinta segurança no tráfego de seus dados. Nesse ponto o ROSRemote juntamente com o *SpaceBrew* podem auxiliar o usuário, pois há a possibilidade de criar seu próprio servidor com suas próprias regras de segurança e utilizá-lo sem a interferência de terceiros. Na seção 2.5.1 é descrito o *SpaceBrew*, uma ferramenta que utiliza WebSockets e auxilia no desenvolvimento desse trabalho ao permitir que o usuário possa realizar conexões de forma simplificada.

2.4 WebSockets

2.4.1 Definição de WebSockets

O *WebSocket* foi inserido como forma de comunicação a partir do HTML5 e funciona como um canal de duas vias, permitindo o envio e recebimento de dados pela web por meio de um único *socket*. Um *socket* deve conter um endereço IP e um número de porta para que a camada de transporte possa identificar a aplicação para qual os dados são destinados. De acordo com Lubbers (2011), o WebSocket não é somente uma melhoria na comunicação convencional feita pelo HTTP mas representa um avanço enorme, especialmente para aplicações que necessitam enviar e receber grandes volumes de dados.

Quando o usuário visita uma página na internet, uma solicitação HTTP é enviada para o servidor que contém a página e o servidor retorna a resposta. O grande problema é que, em alguns casos, até que o navegador receba o retorno a informação já pode estar desatualizada. Para resolver esse problema é necessário atualizar manualmente a página e realizar uma nova solicitação HTTP com todos os seus cabeçalhos.

Segundo Pimentel e Nickerson (2012) até a criação dos WebSockets, para o desenvolvimento de aplicações em rede era utilizado o chamado *HTTP Polling*. Esse método soluciona um problema e cria outro pois é necessário o envio de cabeçalhos de requisição e de resposta que contêm informações desnecessárias, resultando no aumento da latência. Diferentemente do *Polling*, no WebSocket cliente e servidor trocam seus cabeçalhos HTTP fazendo um upgrade de protocolo durante seu "aperto de mãos" (*handshakes*) inicial e após essa saudação, é possível a troca de mensagens a qualquer momento.

A Figura 2.4.1 mostra a diferença entre os dois métodos e é possível notar que no *Polling*, para cada atualização é necessário realizar uma requisição. Por outro lado no caso do WebSocket o servidor pode enviar inúmeros dados sem que o cliente faça o pedido. No protocolo WebSocket, o *handshake* que ocorre entre cliente e servidor se dá por meio de cabeçalhos que indicam que foi realizado um *upgrade* de uma conexão HTTP simples para uma conexão WebSocket.

Para efeitos de comparação entre o HTTP e o WebSocket, Lubbers (2011) utiliza como exemplo o envio de cabeçalhos HTTP de 871 bytes, apenas para mostrar o que ocorre quando uma aplicação que necessita *pollings* frequentes é utilizada por vários usuários ao mesmo tempo.

A Figura 2.4.2 mostra a comparação de *overheads* desnecessários enviados caso haja 1.000, 10.000 e 100.000 clientes conectados ao mesmo tempo ao servidor. No Caso de Uso C, o pior entre os casos, foi necessário o tráfego de 696.800.000 bytes para o HTTP contra apenas 1.600.000 com WebSocket. Devido ao ganho de desempenho, a aplicação criada nesse trabalho utilizará apenas WebSockets para envio e recebimento de mensagens e dados.

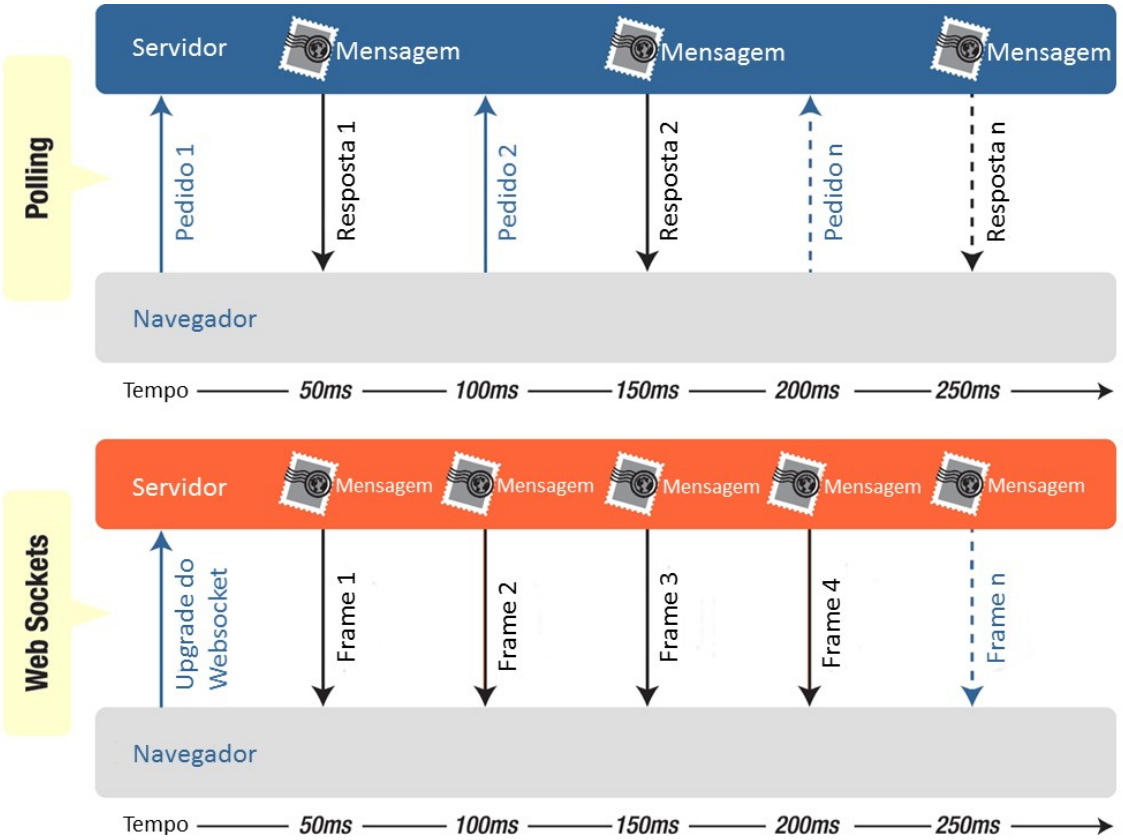

Figura 2.4.1 – Comparação de latência entre uma aplicação com HTTP *Polling* e outra com WebSocket. Adaptado de Lubbers (2011).

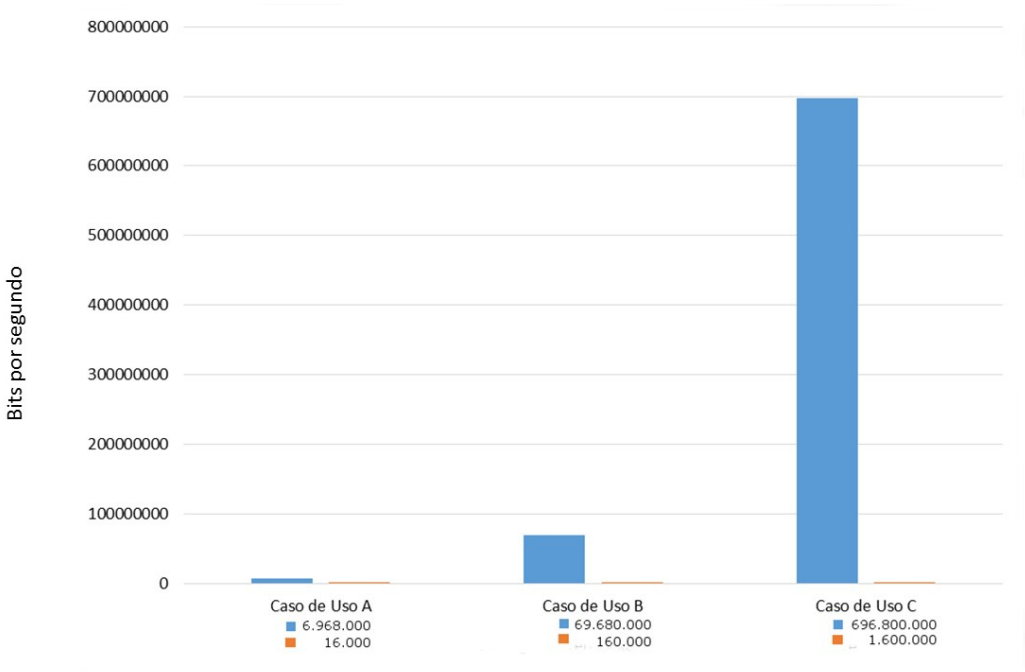

Figura 2.4.2 – Comparação de *overhead* desnecessário entre *Polling* e WebkSocket. Fonte: Adaptado de Lubbers (2011).

2.5 Spacebrew

2.5.1 Definição

O *SpaceBrew* é um *framework* que foi uma das principais ferramentas utilizadas para a realização desse trabalho, pois facilita o envio de informações entre elementos que podem se interagir. De acordo com Group (2014a), o *SpaceBrew* é um *framework* que permite realizar a conexão, de maneira simples, de elementos que podem interagir entre si.

Apesar da descrição utilizada no site dos desenvolvedores da ferramenta, Desolda *et al.* (2017) a descreve de uma maneira diferente. Para os autores, o *Spacebrew* é um conjunto de ferramentas que permitem desenvolver serviços na web utilizando um sistema de "evento-condição-ação". Quando um evento é disparado, sua condição é avaliada e uma ação é tomada em resposta. Ainda segundo os autores, a configuração ocorre conectando os serviços que publicam os eventos aos serviços que fornecem a ação em resposta a esses eventos.

Apesar de Desolda *et al.* (2017) descreverem o *framework* de uma forma complexa, o funcionamento do *SpaceBrew* ocorre da mesma maneira que o do ROS (ROS, 2007), utilizando o conceito de assinante e inscritor (do inglês, *Publisher* e *Subscriber*), sendo o primeiro o responsável por enviar informações e o segundo por recebê-las. Essas informações podem ser dos tipos: (1) **boolean** que pode receber os valores de *true* ou *false*; (2) **number range** que é uma faixa de números que varia entre 0 e 1023; (3) ou **string** que contém um texto comum. Esses tipos de dados são os mais básicos, mas além deles estão sendo implementados outros tipos como som, imagem, alguns tipos de dados que podem ser especificados pelo usuário como o "accel", que envia dados de um acelerômetro, e dados do tipo JSON, amplamente utilizados nesse trabalho.

Uma busca pelas principais bases de dados na internet (Scopus, ScienceDirect, Springer Link e ACM digital Library) revelou que poucos trabalhos utilizam o *SpaceBrew* e a grande maioria deles apenas possuem uma descrição rápida da ferramenta, mas não realizam um estudo sobre suas vantagens e desvantagens. Além disso, grande parte dos artigos encontrados mostram o trabalho de desenvolvimento de uma ferramenta para comunicação pela internet que se assemelha ao *SpaceBrew*, portanto ele somente é citado para efeitos de comparação, mas sem fornecer maiores detalhes.

Entre as aplicações que citam o *SpaceBrew* se encontra o trabalho de Diakopoulos e Kapur (2013), o Netpixl. Nele o *SpaceBrew* é descrito como uma ferramenta que merece atenção e que possui uma abordagem amigável para conectar dispositivos por meio de uma interface gráfica. De forma diferente do *SpaceBrew*, o Netpixl pode trocar dados utilizando outros protocolos, ao invés de focar apenas no WebSocket. Já Estrany *et al.* (2017), em seu trabalho sobre dispositivos que auxiliam na ligação entre humanos e máquinas, apenas escreve que o *SpaceBrew* é uma ferramenta de código aberto e com uma qualidade extraordinária. Merizalde *et al.* (2016), além de citar que o *SpaceBrew* é

uma ferramenta de código aberto, também escreve que é possível conectar dispositivos de forma simples, é baseado em um modelo cliente/servidor e utiliza WebSockets. Em seu trabalho, Blackstock e Lea (2014) descrevem uma ferramenta capaz de permitir ao usuário trabalhar com aplicações distribuídas, porém apenas mostra como o *SpaceBrew* é similar ao seu trabalho. Ghosh (2014) cria uma ferramenta denominada *JADE* que auxilia os desenvolvedores a implementarem aplicações que resolvam uma atividade computacional em particular, porém também apenas descreve as semelhanças entre seu trabalho e o *SpaceBrew*.

Existem outros autores que utilizam o *SpaceBrew* em suas aplicações, mas não fornecem nenhum tipo de conclusão sobre ele. O trabalho de Segrera *et al.* (2013) fornece um contexto para a pesquisa de um Avatar que pode se comunicar com outros como um jogo de video game. Ainda assim ele apenas menciona que quando o jogador realiza um movimento, o jogo envia um sinal para o *SpaceBrew*. Murer *et al.* (2015) descrevem uma aplicação que utiliza um objeto voador que é capaz de impor seu momento angular em um tablet. Nesse trabalho, o autor cita o motivo de ter escolhido o *SpaceBrew* como ferramenta de comunicação e se deve ao fato dele permitir a comunicação entre vários componentes diferentes de forma rápida. Por último, Lankes *et al.* (2016) utilizam o *SpaceBrew* em seu estudo, no qual é descrito uma possível aplicação que permite que um espectador que esteja observando outra pessoa jogar um video game, possa interagir com o jogador apenas com o olhar, a fim de melhorar a experiência do jogo. Porém assim como outros autores, também não há uma conclusão sobre a viabilidade da utilização do *SpaceBrew* na aplicação. Além esses trabalhos também existem um último que utiliza essa ferramenta, porém o autor a descreve de forma errônea. Hemmert e Joost (2016) disserta sobre sua utilização do "protocolo *SpaceBrew*" que, como já descrito anteriormente, não é um protocolo mas sim um *framework*.

O *SpaceBrew* possui algumas vantagens, uma delas é utilizar uma abordagem amigável com uma interface gráfica criada em HTML5 que permite conectar publicadores (*publishers*) e assinantes (*subscribers*) ao servidor. À esquerda dessa interface se encontram os publicadores e à direita, os assinantes e quando ambos são conectados é possível enviar e receber dados de quaisquer aplicações (BLACKSTOCK; LEA, 2014).

Outra vantagem de se utilizar o *SapceBrew* que segundo Arévalo *et al.* (2017) é a mais importante, é que a aplicação oferece permite que seja realizada uma reconfiguração de forma dinâmica e em tempo de execução. Para a aplicação descrita neste trabalho essa característica é vantajosa já que o usuário pode receber informações de vários dispositivos simultaneamente e em algum momento, necessitar que essa recepção seja cessada. Desse modo, ele pode desconectar esse dispositivo remoto e quando necessário, reconectá-lo e continuar recebendo as informações normalmente.

Existem algumas ferramentas citadas na literatura que podem realizar um trabalho semelhante ao do *SpaceBrew*, como:

- *Xively* (https://www.xively.com/);
- *Temboo* (https://temboo.com/) Merizalde *et al.* (2016);
- *Twine* (http://supermechanical.com/twine/);
- *WigWag* (http://www.wigwag.com) Cabitza *et al.* (2015)

Porém todos esses softwares são pagos, podendo ser utilizado gratuitamente por um período de teste. O único software gratuito encontrado foi o *Rapyuta* (http://rapyuta.org/), cuja utilização se encontra no início mas já possui alguns trabalhos publicados, como Mohanarajah *et al.* (2015b), Gherardi *et al.* (2014) e Shinde *et al.* (2016) que são detalhados na seção 2.7.2.1.

Durante testes realizados, foram encontradas algumas limitações do *SpaceBrew*, entre elas as duas mais importantes são: a impossibilidade de conectar dois dispositivos que se encontram na mesma rede e o envio de muitos dados em um curto período de tempo. O primeiro problema não necessariamente afeta esta pesquisa, pois o objetivo é conectar *masters* que se encontram distantes um do outro. Além disso, para conectar dois *masters* que se encontram em uma mesma rede não é necessário o *SpaceBrew*, pois o ROS funciona quando conectado em uma rede local e de forma muito mais rápida do que em uma rede externa.

Outro problema é que ao se utilizar alguns comandos que enviam dados em grande quantidade, o *SpaceBrew* desconecta seus clientes. Isso se deve ao fato de que o servidor entende essa grande quantidade de dados como um ataque DoS (*Denial of Service* ou Negação de Serviço). Um ataque DoS ocorre quando um cliente envia muitas requisições para um servidor em um curto período de tempo, fazendo com que esse fique ocupado as atendendo e não forneça o serviço a outros usuários que desejam se conectar. Devido a esse fato, o servidor cancela a conexão e para resolver esse problema foi necessário utilizar uma taxa de frequência de publicação de dados que é apresentada na seção 3.2.

Na literatura também foi encontrada uma terceira restrição para se trabalhar com o *SpaceBrew*, que é a necessidade de se ter uma pequena noção de programação. Para criar uma aplicação que contenha um cliente é necessário possuir um conhecimento de programação básica. Porém para esse trabalho que utiliza ROS, acredita-se que todos os usuários do ROSRemote saibam programar o suficiente para conseguirem trabalhar com o *SpaceBrew*.

O *SpaceBrew* funciona sobre WebSockets, eles são usados para realizar a conexão entre o cliente e o servidor e é necessário instalá-lo pelo cliente antes do *framework* poder ser utilizado. Além disso, existem algumas regras que necessitam ser seguidas para a aplicação funcionar corretamente, entre elas:

- As aplicações só podem ser conectadas com aplicações que possuam valores semelhantes. Por exemplo, se uma aplicação envia um dado do tipo *boolean*, só poderá enviar e receber informações de outra que forneça o mesmo tipo de dado;
- Um publicador só pode ser conectado a um assinante e não é possível realizar a comunicação entre dois elementos semelhantes;
- É possível conectar um publicador a vários assinantes ao mesmo tempo e vice-versa.

A conexão é realizada utilizando-se a interface gráfica *Web Admin Tool* e é bastante intuitiva. O usuário apenas necessita selecionar o publicador e o assinante que deseja conectar e os dois dispositivos já estão configurados para se comunicar. Caso o usuário não necessite mais dessa conexão, ele pode selecionar um dos dois elementos e em todos os dispositivos conectados a ele surgirá um botão (X com fundo escuro) para desfazer a conexão. A Figura 2.5.1 mostra a interface gráfica do *SpaceBrew* e dois objetos conectados, essa conexão se dá, visualmente, pela linha preta vinculada a um publicador e a um assinante.

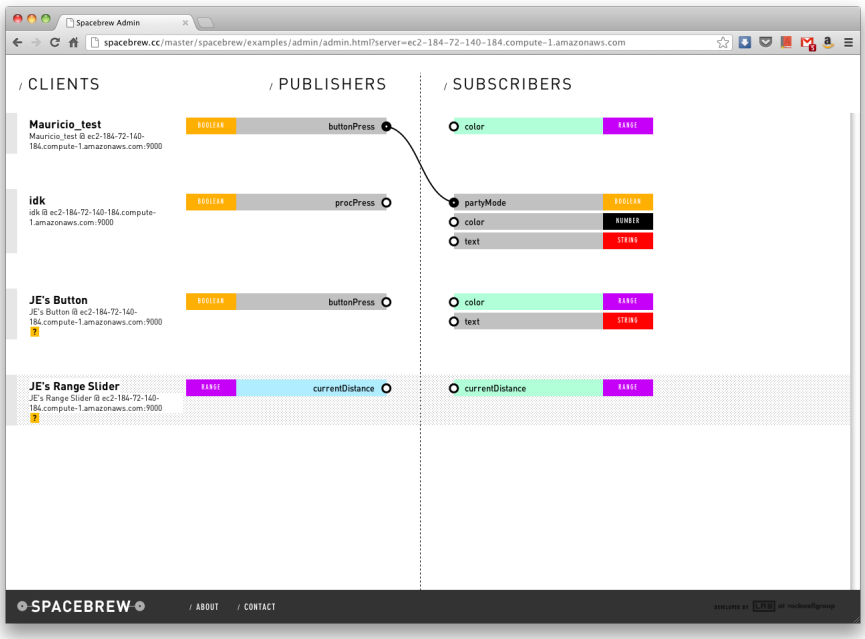

Figura 2.5.1 – Interface gráfica do *Web Admin Tool*, implementada no servidor do *SpaceBrew*, mostrando uma ligação realizada entre aplicações. Fonte - (GROUP, 2014b).

2.6 ROS (*Robot Operating System)*

2.6.1 Definição

Existem vários sistemas capazes de auxiliar no desenvolvimento de software para controle de robôs, como demonstra Kerr e Nickels (2012) em sua pesquisa que descreve e avalia os principais sistemas para se operar robôs. Entre todos os mencionados na pesquisa, o ROS foi um dos que mais se destacou. Essa pesquisa levou em consideração que os estudantes da Universidade de Trinity, local onde ela foi criada e conduzida, não possuíam muitos conhecimentos em programação, fato que resultou em uma nota mais baixa para o ROS nesse estudo. Outros sistemas utilizados nessa pesquisa, como o *Microsoft Robotics Developer Studio* (JACKSON, 2007), permitem desenvolver aplicações sem que seja necessário ter um nível alto de conhecimento em programação. Por outro lado, na mesma pesquisa, o autor diz que o ROS é a ferramenta mais utilizada atualmente. Apesar da necessidade de se possuir conhecimentos em programação para utilizar o ROS, ele possui muitas outras vantagens que foram responsáveis pela escolha dessa ferramenta para a realização desse trabalho, entre elas podem ser destacadas:

- Várias plataformas: Permite a utilização e comunicação de vários dispositivos diferentes (QUIGLEY *et al.*, 2009);
- Modularidade: Minimiza a dificuldade de realizar *debugging* e evita que o sistema falhe como um todo. Caso um módulo falhe, todo o resto se mantém em funcionamento;
- Facilidade: O ROS pode ser facilmente incorporado às bibliotecas mais utilizadas para tarefas específicas, como o OpenCV (BRADSKI, 2000).

O ROS é um sistema de código aberto que auxilia o controle de componentes robóticos por meio de um computador. É composto por uma quantidade de Nós (do inglês, *nodes*) que comunicam entre si pelo envio e recebimento de mensagens, essas mensagens são trocadas através de Tópicos (*Topics*) (ROBOTICS, 2014).

Para que o ROS funcione é necessária a criação de um *Master* e ela pode ser realizada em um computador ou diretamente dentro de um robô. O *Master* fornece serviços de nomes e registros para todos os Nós do sistema e permite que Nós consigam localizar outros Nós, principal objetivo do *Master* (ROS, 2016a). Em resumo, o *Master* serve simplesmente para facilitar a comunicação entre Nós (O'KANE, 2014).

O *Master* funciona como um servidor que conecta dois computadores. Em redes, quando um usuário envia informações para outro cuja localização é desconhecida, ele envia uma requisição com o nome para o servidor DNS que busca o destino com o qual deseja se conectar. Ao encontrar seu destino, o servidor faz a conexão entre os dois clientes e isso permite que eles possam trocar mensagens diretamente. Na internet, essa conexão

é realizada entre dois clientes, no ROS, é feita entre Nós. Pela própria definição do ROS, Nós são "processos que realizam um processamento" (ROS, 2016b) ou, segundo O'Kane (2014), Nós são uma instância em execução de um programa do ROS e ainda podem ser descritos como um "módulo de um software" (QUIGLEY *et al.*, 2009).

Os Nós, ao serem combinados, formam uma espécie de grafo e as informações entre eles são trafegadas através de tópicos e chamadas remotas à procedimentos (RPCs). A figura 2.6.1 mostra um grafo criado em ROS através do comando rqt_graph (ROS, 2016c). As elipses mostram os Nós criados e as setas, os Tópicos através dos quais os dados são transferidos.

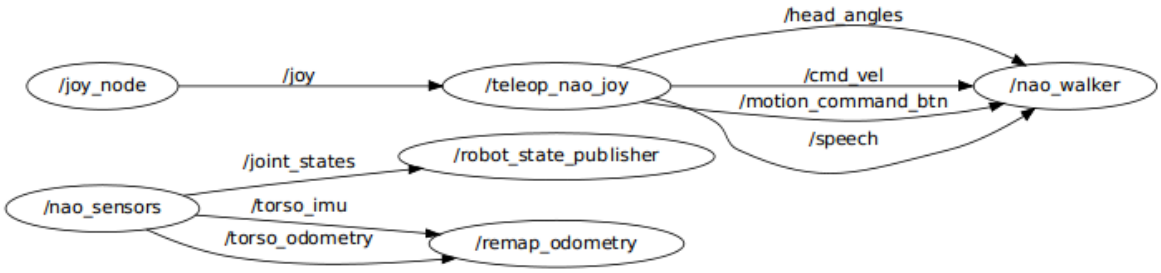

Figura 2.6.1 – Exemplo de um grafo do ROS, mostrando alguns dos Nós utilizados para controle do robô NAO. Fonte: (ROS, 2017).

Existem várias vantagens ao se utilizar Nós para comunicação e controle no ROS. O principal benefício é a prevenção de falhas, visto que cada Nó trabalha isoladamente e caso ocorra alguma falha, apenas um deles para de funcionar, não comprometendo o restante do sistema. Os Nós se comunicam entre si através de mensagens, que são simplesmente uma estrutura de dados.

Os Tópicos possuem uma semântica de publicadores (*publishers*) e assinantes (*subscribers*) e os Nós não sabem de quais Nós estão recebendo a informação. Eles simplesmente se inscrevem nos Tópicos os quais desejam receber os dados e publicam nos Tópicos quando desejam distribuir seus dados.

De acordo com Ros (2016f), "Tópicos são barramentos nomeados, através dos quais os Nós trocam mensagens". Os Nós que estão interessados em um dado se inscrevem em um Tópico relevante e os Nós que querem gerar dados publicam no Tópico desejado, podendo haver vários *publishers* e/ou vários *subscribers* para um mesmo Tópico (QUIGLEY *et al.*, 2009). Cada Tópico no ROS publica ou se inscreve para receber um tipo de mensagem e os Tópicos só podem recebê-las se forem do mesmo tipo que eles esperam. Essas mensagens são utilizadas pelo ROS para facilitar a geração de código fonte para diferentes linguagens.

Existem inúmeros tipos de mensagens, cada uma utilizada para enviar um tipo específico de dado, no entanto todas possuem a mesma função que é facilitar a comunicação entre várias linguagens diferentes, permitindo que uma aplicação criada em Python possa se comunicar com outra em C++, por exemplo.

O único problema dos Tópicos é que eles servem apenas para comunicação unidimensional, uma operação que necessita a realização do envio de algum dado e espera uma resposta não pode ser executada com a utilização de Tópicos, para isso o Nó deve se inscrever em um Serviço (ROS, 2016d). Um Nó que fornece um Serviço o faz utilizando-se de um nome único e um cliente o requere enviando uma mensagem. Esse Serviço é bloqueante, ou seja, o cliente fica impossibilitado de realizar novas ações até que receba a resposta do Nó fornecedor.

Os serviços são definidos utilizando-se arquivos *srv* (ROS, 2016e), que funcionam da mesma forma que os *msg* para mensagens, porém o *srv* deve ser definido pelo usuário e após serem criados, também são compilados no código fonte. Os *srv* são divididos em duas partes e separados por três traços. A primeira parte indica os dados necessários para chamar o serviço e a segunda parte o que será enviado como resposta.

Algoritmo 1: Exemplo de um serviço que recebe uma string como parâmetro e devolve um inteiro como resposta

```
1 string str
2 ---
3 int8 num
```

O ROS pode transportar suas mensagens utilizando dois protocolos, um baseado em TCP (o TCPROS), mais confiável e lento, e outro baseado em UDP (o UDPROS), mais rápido porém com maior taxa de perda de dados, sendo recomendado para *streaming* de vídeos e tele operações (ROS, 2016f). Até o momento o UPDROS somente é suportado pelo roscpp, aplicações criadas utilizando C++. Como esse trabalho foi desenvolvido em Python e essa linguagem só possui suporte ao TCP, será utilizado apenas o TCPROS.

O ROS é uma ferramenta cuja utilização vem crescendo exponencialmente (BOREN; COUSINS, 2011) e uma busca por algumas bases de dados comprova que existem várias pesquisas relacionadas a esse conjunto de ferramentas e bibliotecas. Utilizando as palavras chave "Robot Operating System" entre aspas, foram encontradas 434 publicações na bases de dados IEEEXplore, 464 no ScienceDirect, 1004 no Springer Link e somente 53 na ACM Digital Library, entre livros, artigos e conferências. O ROS permite trabalhar com qualquer tipo de aplicação e isso contribui para que existam várias publicações utilizando esse assunto.

O ROS é utilizado em várias áreas diferentes, desde robôs industriais até veículos aéreos não tripulados. Entre algumas das principais pesquisas realizadas com essa ferramenta podem ser citadas o FIT IoT-LAB (ADJIH *et al.*, 2015), uma plataforma do tipo banco de ensaio (*testbed*) que permite para que o usuário possa realizar experimentos em grande escala que vão desde testes com protocolos até serviços avançados pela internet. Todos os testes podem ser realizados com tecnologias relacionadas a Internet das Coisas. Os autores apenas citam que os robôs móveis existentes nessa plataforma utilizam o ROS, mas não fornece maiores detalhes sobre a utilização dessa ferramenta.

Outra aplicação interessante que pode ser destacada é descrita por (WEAVER *et al.*, 2013). Nesse trabalho é descrito uma maneira de realizar decolagens, voos e pousos com um auxílio de um veículo não tripulado. No artigo, os autores citam que o exército dos Estados Unidos conseguiu completar essa tarefa com um custo de bilhões de dólares, porém utilizando ROS, os autores buscam realizar o mesmo trabalho utilizando apenas alguns milhares de dólares.

As aplicações que utilizam ROS não se resumem somente em áreas que não afetam a vida de pessoas no dia a dia, existem aplicações que foram criadas para facilitar os trabalhos cotidianos. Como o trabalho de Tomoya *et al.* (2017) que detecta quedas de pessoas idosas e informa para um observador para que esse possa tomar as atitudes necessárias, como enviar ou prestar socorro ao acidentado. Para monitorar e controlar o robô é utilizado o ROS e um computador além de um robô com um sensor Kinect da Microsoft. Apesar de os autores descrevem o ROS, eles não descrevem maiores detalhes sobre essa ferramenta nem como ela foi utilizada para a comunicação.

Além dos trabalhos descritos, existem outras áreas que também utilizam o ROS como a educação (ZDEŠAR *et al.*, 2017), técnicas de localização e mapeamento (SANTOS *et al.*, 2013) e (LI; BASTOS, 2017) e frameworks que auxiliam o desenvolvimento de aplicações na nuvem, como o *Rapyuta*, que é descrito na seção 2.7.2.1.

O ROSRemote não é a primeira aplicação utilizada para controle e monitoramento de robôs a distâncias e nas seções 2.7.1.1, 2.7.1.2, 2.7.2.1 e 2.7.2.2 são descritas algumas ferramentas que também podem realizar esses trabalhos.

2.7 Trabalhos relacionados ao ROSRemote

2.7.1 Ferramentas Genéricas

2.7.1.1 SSH

Assim como o TCP e o UDP, o SSH também é um protocolo de tráfego de dados pela internet. Sua principal utilização é para acesso remoto a máquinas por uma rede segura mesmo que ela esteja situada dentro de uma rede insegura. Sempre que um dado é enviado para a rede ele é criptografado para garantir que sua leitura não seja possível mesmo que alguém o intercepte. Quando o computador de destino recebe a mensagem, ele sabe como descriptografá-la (YLONEN; LONVICK, 2006).

O SSH trabalha com uma arquitetura cliente e servidor, o cliente requisita um serviço e o servidor o fornece (BARRETT *et al.*, 2005). Portanto, o usuário necessita configurar um servidor no local onde ele deseja realizar o acesso remoto e posteriormente solicitar a conexão com esse servidor, que pode ser realizada em uma mesma rede ou em redes diferentes. Quando a conexão é realizada é possível solicitar arquivos, realizar comandos e no caso desse trabalho, utilizar o ROS na máquina remota.

De acordo com as especificações dos autores Ylonen e Lonvick (2006), o SSH (*Secure Shell*) consiste em três componentes primários:

- O protocolo da camada de transporte: oferece autenticação, confidencialidade e integridade. Funciona tipicamente sobre uma conexão TCP/IP, mas pode ser utilizado em outros fluxos de dados confiáveis;
- O protocolo de autenticação de usuário: autentica o cliente que deseja utilizar o servidor;
- O protocolo de conexão: divide o túnel criptografado em vários canais lógicos.

De acordo com (BARRETT *et al.*, 2005), o protocolo SSH foi criado para cobrir três princípios básicos:

- Autenticação: determina a identidade de alguém, se um usuário tentar se autenticar em um computador remoto, é necessário uma prova digital de sua identidade. Essa prova pode ser tanto um nome de usuário e senha, quanto um par de chaves pública e privada;
- Criptografia: codifica os dados para que ninguém consiga entendê-los a não ser o destinatário;
- Integridade: garante que os dados trafegados não sejam alterados.

Entre as várias possibilidades de utilização do SSH encontram-se *logins* remotos, que permitem a conexão de forma segura a um computador que esteja localizado em outra rede, transferir arquivos de forma segura e a utilização mais comum e que também será abordada nesse trabalho, a execução de comandos em um terminal remoto. Basta o usuário se conectar com a máquina desejada através do endereço IP ou de um nome (DNS) e efetuar a autenticação. A partir desse procedimento já é possível que o usuário envie comandos como se estivesse em sua própria máquina, porém eles são enviados para a máquina remota de forma imperceptível.

Para sua utilização, o usuário deve criar um servidor no computador ao qual o usuário irá se conectar para realizar as tarefas. No Ubuntu, sistema operacional onde os testes foram executados, é necessário realizar o download da ferramenta *openssh-server* e alterar algumas configurações necessárias. Depois, é necessário que o usuário faça o *port forwarding* (ou redirecionamento de portas) para que seja possível realizar o acesso a partir de qualquer rede.

Vale ressaltar que mesmo que o SSH possa realizar o mesmo trabalho que o ROSRemote, ambos são duas ferramentas bem diferentes. O ROSRemote foi criado exclusivamente para que o desenvolvedor possa utilizar suas aplicações em um computador remoto se conectando de maneira simples através do *SpaceBrew*. Enquanto o SSH é um protocolo

que auxilia o usuário a trafegar dados de maneira segura através da internet, que é um ambiente inseguro.

2.7.1.2 VPN

VPN é uma sigla para *Virtual Private Network* e como diz o nome, é uma rede privada que só existe virtualmente e envia dados criptografados (SCOTT *et al.*, 1999). O principal motivo das VPNs terem sido criadas é para trafegar dados que não podem ser vistos por terceiros, como informações importantes de uma empresa com localizações físicas distintas (SCOTT *et al.*, 1999). Para isso, realiza-se uma conexão privada entre os dispositivos que irão compartilhar a informação e somente os dois poderão ter acesso aos dados.

De acordo com Odiyo e Dwarkanath (2011), as VPNs devem garantir algumas propriedades principais:

- Confidencialidade: a privacidade dos dados que estão sendo trocados entre os dispositivos deve ser protegida;
- Integridade: a informação enviada não pode ser modificada durante o tráfego;
- Autenticação: garante a identidade dos dispositivos que irão se comunicar, isso pode ser facilmente alcançado utilizando-se senhas ou certificados digitais.

Além da utilização para acessar informações confidenciais existem outros usos muito comuns para as redes VPNs. Scott *et al.* (1999) citam os mais importantes que são: acessar redes corporativas a distância, esconder as atividades do usuário do provedor de internet, acessar sites geo-bloqueados, burlar censuras na internet e realizar *downloads* de conteúdos ilegais.

A ideia do usuário esconder suas atividades é interessante quando ele está conectado em uma rede pública na qual vários outros dispositivos estão conectados ao mesmo tempo. Outro uso bastante comum é o acesso à sites geo-bloqueados, pois alguns sites não podem ser acessados por usuários que se encontram em países específicos ou serviços de *streaming* de vídeos (como a Netflix e o YouTube) bloqueiam alguns de seus vídeos para alguns países e o usuário não consegue acessá-los nessa região. Dessa forma, é possível se conectar com outros computadores que estejam fora desses países (através de uma VPN) e acessar os serviços bloqueados. Por último, VPNs são muito utilizadas para se realizar downloads de conteúdos ilegais. Muitos países proíbem essa prática, como utilização de gerenciadores de *torrents* para downloads de conteúdo "pirata", inclusive aplicando multas severas a quem realiza essas atividades. Nesse caso, a VPN é utilizada para a realização dos downloads como se o usuário estivesse em outro local, evitando que o fornecedor de internet que presta serviços a ele consiga interceptar sua atividade ilegal.

Existem dois tipos de VPN a chamada site-to-site (ponto-a-ponto ou roteador-a-roteador) e a VPN de acesso remoto (FORNERO, 2016). O primeiro caso é mais comumente utilizado por empresas, pois algumas delas possuem mais de um escritório em locais diferentes e precisam trocar informações confidenciais entre eles. Dessa forma optam por se conectar através de dois roteadores utilizando uma VPN. Nesse caso, os dados são trafegados de forma segura entre os dois roteadores através da internet e distribuídos internamente nas respectivas redes locais.

Existe também a VPN de acesso remoto que é a mais familiar para o usuário comum (FORNERO, 2016). Ela serve para se conectar a uma rede externa a que o usuário se encontra, seja para acessar dados localizados em outro computador, para mascarar a atividade na internet ou para oferecer mais segurança ao realizar acessos em redes públicas. Nesse tipo de VPN o usuário se conecta diretamente ao roteador da empresa ou de sua casa e trafega dados de forma segura. Para realizar essas conexões, as redes VPNs contam com três protocolos principais (SCOTT *et al.*, 1999):

- PPTP (*Point-to-point Tunneling Protocol*): significa Protocolo de Tunelamento Ponto-a-Ponto e pode ser utilizado em vários sistemas operacionais e dispositivos, pois necessita somente de três dados: nome de usuário, senha e o endereço do servidor. Esse protocolo é o mais rápido de todos, pois seus dados não são criptografados causando pouco *overhead* (ODIYO; DWARKANATH, 2011). A única desvantagem desse protocolo é que, por ser antigo, é menos seguro do que os mais atuais;

- L2TP (*Layer 2 Tunnelling Protocol*): criada a partir da união da rápida conectividade do PPTP (da Microsoft) e da flexibilidade, acessibilidade e baixo custo do L2F (Layer 2 Forwarding da Cisco) (MICROSOFT, 2017). A maior vantagem desse protocolo em relação ao PPTP é que o L2TP pode ser utilizado em redes que não são baseadas em IP;

- IPSec (*Internet Protocol Security*): fornece segurança para a camada de rede do modelo OSI (CARMOUCHE, 2007), portanto funciona em conjunto não só com as redes IP atuais mas também com as futuras, como o IPv6. O IPSec realiza duas operações, chamadas de modos de operação: a primeira é o modo de transporte, que é responsável por criptografar as mensagens em um pacote e a segunda é o modo de "tunelamento", chamado assim devido a uma analogia a um túnel de carros, por onde todos os dados trafegam e não podem ser vistos por quaisquer dispositivos ou serviços que estejam foram dele.

As seções 2.7.1 e 2.7.2 descrevem alguns trabalhos similares ao ROSRemote. Dois deles, os da seção 2.7.1.1 e 2.7.1.2 não foram criados com objetivos específicos de realizar o controle e monitoramento dos robôs, porém podem ser utilizados para esse fim. Já

as ferramentas descritas nas seções 2.7.2.1 e 2.7.2.2 foram criadas exclusivamente para trabalhar com robôs, porém de uma forma diferente do ROSRemote.

2.7.2 Ferramentas Específicas Desenvolvidas em ROS

2.7.2.1 Rapyuta

O *Rapyuta* é um *framework* open-source que auxilia desenvolvedores a criarem aplicações robóticas que funcionam na nuvem (RAPYUTA, 2015). Ele é baseado em um modelo de computação de alta escalabilidade que aloca ambientes computacionais seguros, chamados de clones, para cada robô que se conecta à ele. Esses clones são fortemente conectados entre si, permitindo que robôs compartilhem suas informações com outros (MOHANARAJAH *et al.*, 2015a). Além disso, o *framework Rapyuta* oferece amplo acesso ao *RoboEarth*, descrito na seção 2.3, e permite que os robôs utilizem dados coletados através da experiência de outros robôs para a realização de suas próprias tarefas.

O que faz o *Rapyuta* ser tão potente é que, até a sua criação, o *RoboEarth* era apenas um repositório de dados e informações enviadas por vários robôs ao redor do mundo. Porém, a partir de seu desenvolvimento, foi possível retirar dos robôs todo o processamento das informações e colocá-lo na nuvem, permitindo que os robôs possuam cada vez menos recursos locais e diminuindo o preço desses robôs. Devido a esse fato, é comum ver esse *framework* ser chamado de *RoboEarth Cloud Engine* (MOHANARAJAH *et al.*, 2015a).

Outra vantagem desse framework é sua compatibilidade com o ROS, sendo possível utilizar praticamente todos os pacotes desenvolvidos até o momento sem que seja necessário possuir robôs caros e com recursos variados, pois todo o processamento do ROS pode ser feito fora do robô. Além disso, também é possível configurar o *Rapyuta* para funcionar em compatibilidade com outros *middlewares* para controle de robôs.

No entanto, o *Rapyuta* funciona de forma diferente do ROSRemote, pois serve principalmente para auxiliar em processamentos e tomadas de decisões que sejam necessárias. Assim é possível que o usuário controle um robô utilizando o *Rapyuta*, porém ele não foi criado com esse intuito, mas sim com a função de diminuir os custos de robôs, evitando que o usuário necessite realizar um investimento em robôs caros e com muitos recursos.

Da mesma forma que o ROSRemote, o *Rapyuta* também utiliza *WebSockets*, explicado na seção 2.4.1, para realizar a conexão entre o servidor e os robôs, pois esse protocolo proporciona uma conexão rápida e em duas vias. Isso é necessário, pois da mesma forma que o robô pode enviar informações para o servidor, ele também pode receber informações e comandos. A Figura 2.7.1 demonstra como ocorre o funcionamento do *Rapyuta*. Os retângulos que se encontram na nuvem são considerados ambientes seguros nos quais o robô realiza seus processamentos. Esses ambientes são intimamente conectados entre si e possuem uma conexão em banda larga com o *RoboEarth*, representado pelos cilindros na nuvem ao fundo da imagem.

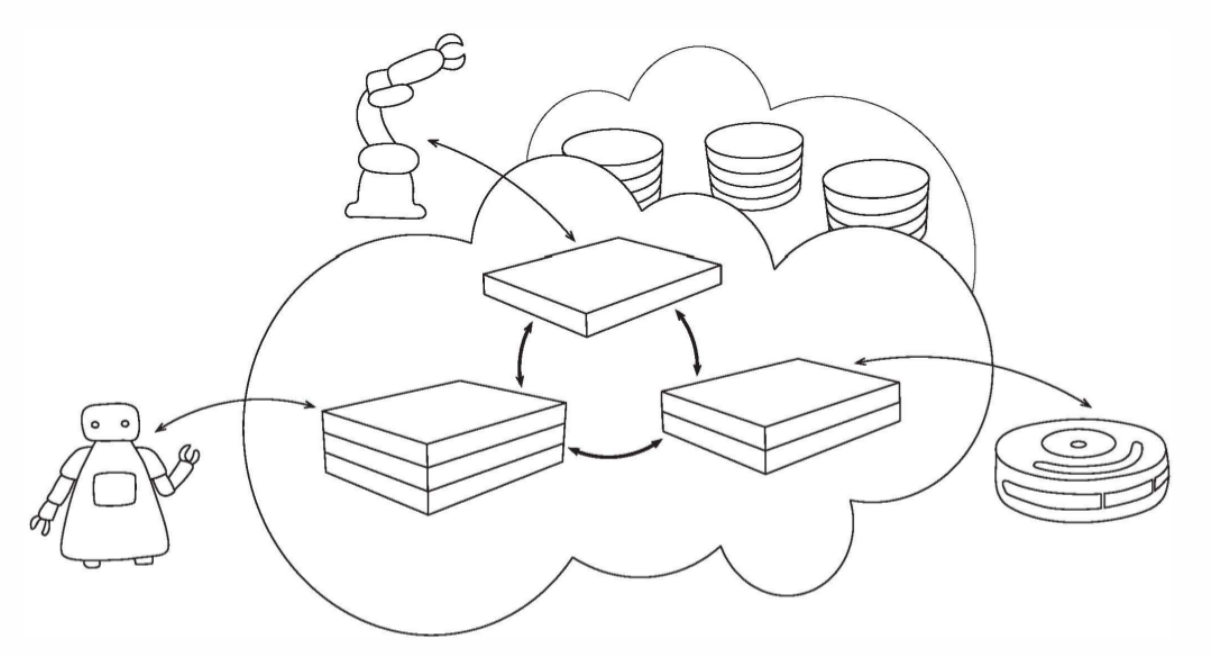

Figura 2.7.1 – Visão geral do Rapyuta. Fonte: (MOHANARAJAH *et al.*, 2015a).

De acordo com (MOHANARAJAH *et al.*, 2015a), o *Rapyuta* é dividido em quatro componentes principais;

- Ambientes computacionais: implementados utilizando Linux, pois permite maior segurança e escalabilidade. Os ambientes computacionais são configurados para executar qualquer processo que seja um nó do ROS e todos eles se comunicam utilizando a comunicação entre processos do próprio ROS;
- Comunicação e protocolos: utiliza *Endpoints* para definir uma interface e estes *Endpoints* comunicam entre si através de portas. As interfaces são utilizadas para comunicar processos do *Rapyuta* com processos que ocorrem fora dele;
- Conjunto de tarefas centrais: existem quatro tipos de tarefas principais que administram o sistema:
  - Conjuntos de tarefas mestre: é o controlador principal. Ele mantém e comanda a estrutura de dados, como a conexão entre robôs e o *Rapyuta*;
  - Conjunto de tarefas do robô: recursos necessários para se comunicar com um robô;
  - Conjunto de tarefas de ambiente: recursos necessários para se comunicar com um ambiente computacional;
  - Conjunto de tarefas de contêiner: recursos necessários para iniciar e parar ambientes computacionais. Dentro de cada máquina existe um conjunto de tarefas de contêiner sendo executado.

- Estrutura de dados de comando: o *Rapyuta* funciona através de uma estrutura de dados centralizada. Ela é gerenciada pelo conjunto de tarefas mestre e consiste de quatro componentes: a Rede, o Usuário, o Balanceador de Carga e o Distribuidor.

O framework *Rapyuta* é uma ferramenta nova que existe a menos de três anos e portanto, não existem muitos trabalhos que o utilizem, apesar de existir uma vasta quantidade de artigos que referenciam esse framework. Em uma busca pelas principais bases de dados utilizando *Rapyuta* como palavra chave, foram encontrados cerca de 50 artigos. Apesar dessa quantidade menos de 10 descrevem aplicações reais utilizando essa ferramenta e além disso, alguns deles descrevem a mesma aplicação. Todo o restante apenas cita a ferramenta ou a compara com outras aplicações que auxiliam no processamento de dados fora do robô. Portanto é possível assumir que assim como o *SpaceBrew*, o *Rapyuta* também é pouco utilizado.

Entre as aplicações encontradas, é possível destacar aplicações de planejamento de caminhos, segundo Lam e Lam (2014), visto que realizar essa tarefa é algo computacionalmente custoso. Essas aplicações são capazes de construir um caminho para o robô alcançar um destino, resolver o problema de caminho mínimo e ainda seguir pessoas. Além disso, também foram encontradas algumas aplicações relacionadas a sistemas multi-robôs (ROSA; ROCHA, 2017), mapeamento em três dimensões de uma localização utilizando robôs de baixo custo (MOHANARAJAH *et al.*, 2015b) e uma aplicação relacionada a prevenção de colisões aéreas entre Quadcópteros (SHINDE *et al.*, 2016), como drones.

2.7.2.2 ROSLink

Dentre todas as soluções pesquisadas, o ROSLink é a que mais se assemelha ao ROS-Remote. Assim como esse trabalho, a motivação do ROSLink é que o ROS não oferece suporte para monitoramento e controle de robôs pela internet (KOUBAA *et al.*, 2017). A abordagem padrão do ROS é mostrada na Figura 2.7.2 e funciona de forma que um ou mais usuários só conseguem se conectar a um robô. Não é possível que um usuário se conecte a mais de um robô e envie comandos simultâneos.

Ainda no mesmo trabalho, Koubaa *et al.* (2017) demonstra outra abordagem do ROS, na tentativa de conectar vários robôs a vários usuários. Essa abordagem é conhecida por centralizada e funciona com um único *master* conectando todos os os usuários a todos os robôs. Essa abordagem possui algumas desvantagens, entre elas a falta de escalabilidade e conflito que vários Tópicos, Nós e Serviços com o mesmo nome podem causar. Essa abordagem é mostrada na Figura 2.7.3.

A maioria das aplicações na internet são baseadas em uma configuração cliente/servidor e na maioria das vezes, o servidor é implementado diretamente no robô. Isso impede que o ROS seja utilizado pela internet, pois o robô nem sempre possui um IP público. Devido a esse fato, o ROSLink foi criado como uma forma de retirar o servidor ROS dos

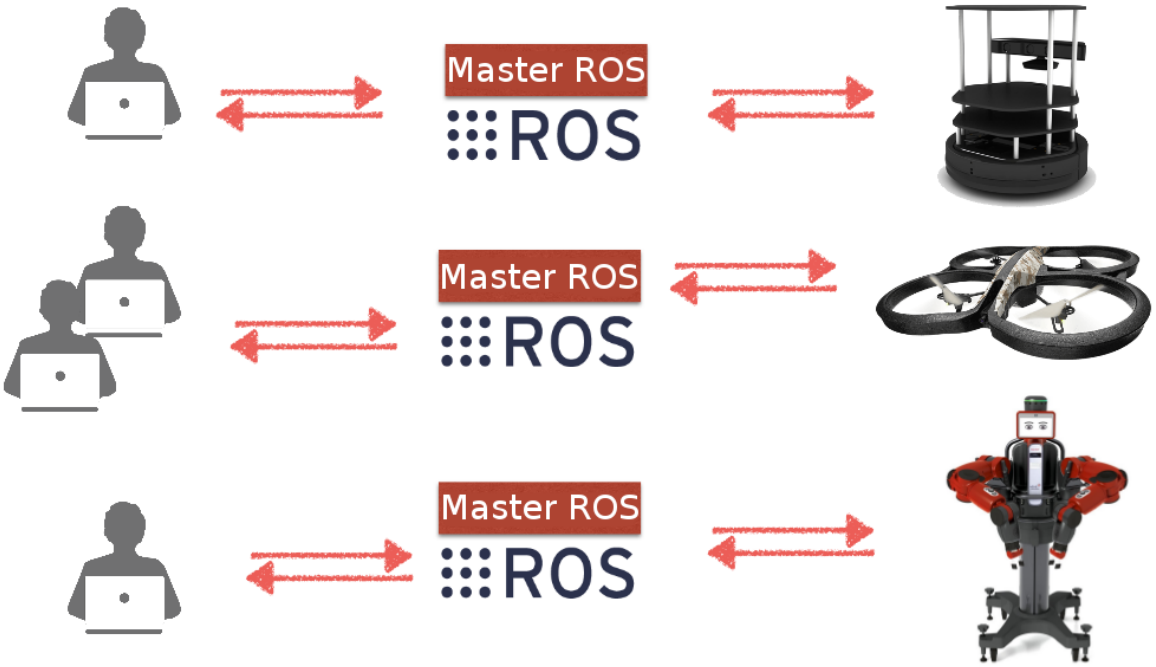

Figura 2.7.2 – Abordagem padrão do ROS. Um ou mais usuários se conectam a um único robô. Fonte: Adaptado de (KOUBAA *et al.*, 2017).

robôs. Nele, o robô e o usuário implementam um cliente, enquanto o servidor é transferido para um domínio público, como mostrado na Figura 2.7.4. Isso permite que vários usuários se conectem a vários robôs ao mesmo tempo e que cada um deles possua seu próprio *master*. Por meio de um *bridge* o ROS é conectado a um servidor na internet que atua como um *proxy*. Essa solução é a mesma utilizada pelo ROSRemote, com algumas diferenças na configuração e utilização que serão explicadas no capítulo 4.

O ROSLink funciona baseado em uma arquitetura composta por três partes:

- O bridge do ROSLink: conecta o ROS remoto ao proxy na nuvem e possui duas funcionalidades principais: (1) lê os dados dos tópicos, serializa em um JSON e o envia; (2) recebe os dados do JSON, deserializa, analisa e executa o comando no ROS;
- A nuvem e o proxy do ROSLink: conecta a aplicação do usuário ao robô por meio do bridge do ROSLink;
- A aplicação cliente do ROSLink: monitora e controla as aplicações do robô.

O protocolo de transporte do ROSLink funciona tanto com UDP quanto com WebSockets e TCP, isso permite uma flexibilidade no envio dos dados. Dessa forma, dados menos importantes podem ser enviados de forma mais rápida pelo UDP e dados mais importantes podem ser enviados de forma mais segura e confiável pelo TCP. Esse protocolo é baseado na troca de mensagens do tipo JSON e essas mensagens possuem tipos pré especificados pelos desenvolvedores. Elas podem ser dos tipos:

Figura 2.7.3 – Abordagem centralizada do ROS. Um ou mais usuários se conectam a um ou mais robôs por meio de um único *Master*. Fonte: Adaptado de (KOUBAA *et al.*, 2017).

- Mensagem de presença: o robô deve declarar sua presença sempre para ser considerado ativo;
- Mensagem de movimento: contém informação sobre o posicionamento do robô;
- Mensagem de sensor: responsável por recuperar os dados dos sensores internos do robô;
- Comandos de movimento: responsável por realizar a movimentação do robô por meio do envio de mensagens.

Essas mensagens possuem um cabeçalho e uma carga útil. A estrutura do cabeçalho pode ser vista na Figura 2.7.5 e a carga útil é simplesmente a mensagem ou o comando que o usuário deseja enviar para o robô.

Aparentemente o ROSLink é útil e funciona rápido, porém o artigo aparenta estar incompleto e não descreve como é realizada a conexão entre o robô e o usuário. Os testes mostrados no artigo são promissores, porém a dificuldade em entender como ocorrem e replicá-los, além do fato de não ser possível encontrar outro artigo que utiliza o ROSLink, acaba diminuindo a atratividade da ferramenta.

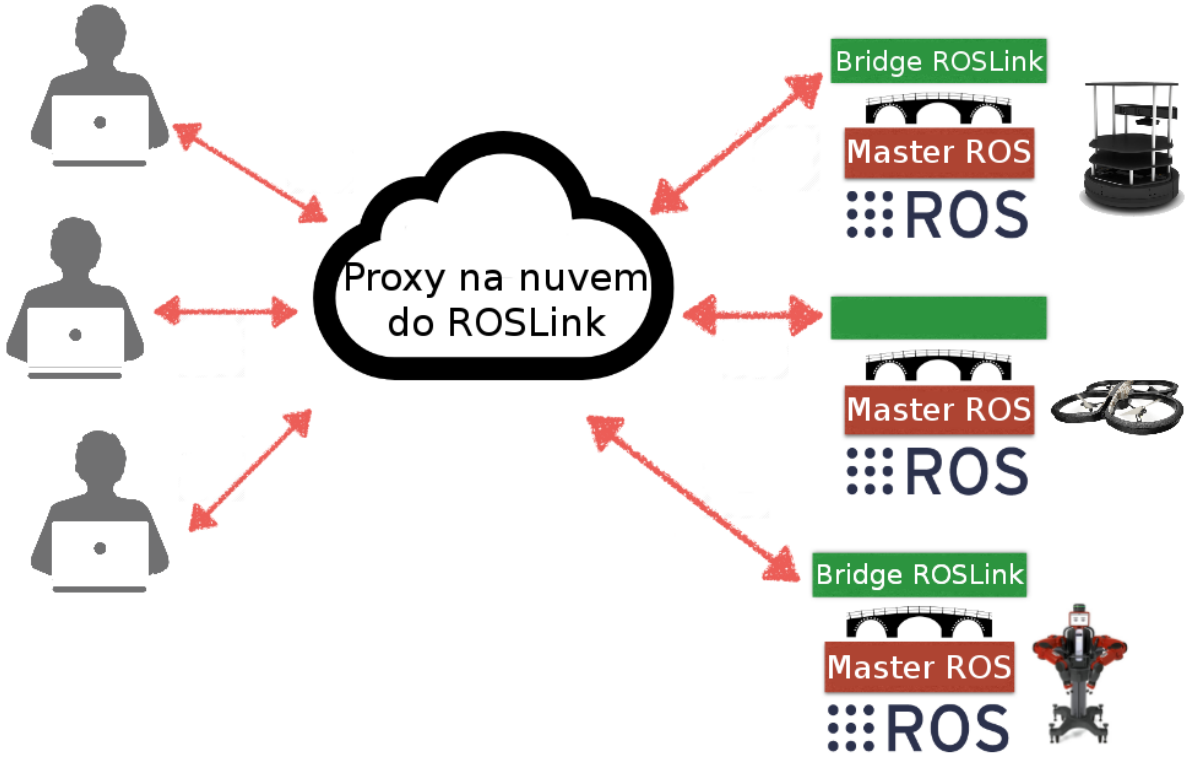

Figura 2.7.4 – Abordagem centralizada do ROS. Um ou mais usuários se conectam a um ou mais robôs por meio de um único *Master*. Fonte: Adaptado de (KOUBAA *et al.*, 2017).

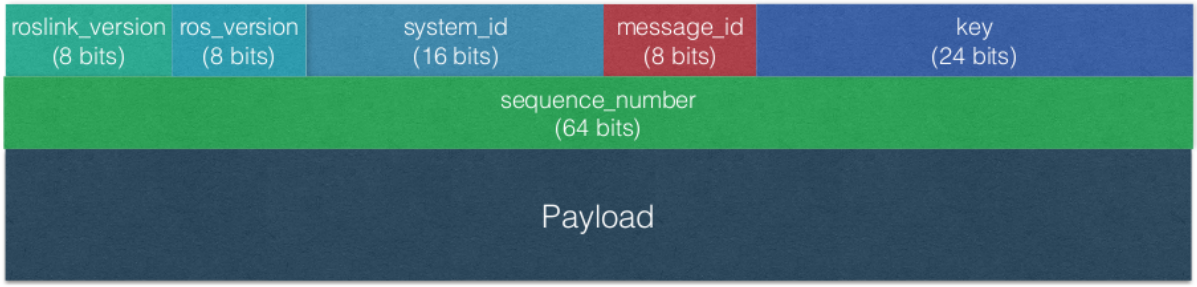

Figura 2.7.5 – Cabeçalho de mensagens do ROSLink. Fonte: (KOUBAA *et al.*, 2017).

2.8 Considerações Finais

Esse capítulo cita qual o principal problema este trabalho busca resolver, descrevendo a área onde se encontra, quais as ferramentas disponíveis para o usuário até o momento e quais ferramentas foram utilizadas para a realização deste projeto.

Para tentar resolver os problemas descritos com conexão e dificuldade de utilização foi criado o ROSRemote e o capítulo 3 mostra como esse *framework* foi criado e como é realizada a conexão entre ele e o *SpaceBrew*, bem como uma explicação sobre as funcionalidades da ferramenta.

3 ROSRemote

Este capítulo descreve a ideia principal do trabalho, quais foram as dificuldades para completá-lo e como ocorre sua utilização e desenvolvimento de futuras funcionalidades. Para isso é mostrado como ocorre a comunicação do *framework* ROSRemote com o *SpaceBrew*, todo o fluxo de dados e como foi feita a programação do software, utilizando-se a linguagem Python.

Vários fatores levaram a escolha do Python como a linguagem utilizada nesse trabalho, entre eles é possível citar que ela possui uma ênfase em facilitar a leitura do código e isso é um atrativo em uma licença *open-source*. Outra vantagem em relação à outras linguagens suportadas pelo Ros é que, utilizando Python é possível escrever o mesmo código utilizando menos linhas, além de permitir programas limpos tanto em pequena quanto em grandes escalas (ROSSUM *et al.*, 2007).

Trechos de código são mostrados e explicados para que, caso seja necessário adicionar novas funções à ele, o desenvolvedor possa entender toda a estrutura e onde cada novo código deve ser inserido.

3.1 ROSRemote e SpaceBrew

Como mencionado na introdução, o objetivo do trabalho é criar uma aplicação que simule o ROS com algumas de suas funções e permita que elas sejam enviadas e executadas em um *Master* remoto. Após esta etapa, é preciso verificar a viabilidade da aplicação e constatar se o *SpaceBrew* é responsável ou não pelo comportamento da aplicação. Caso a aplicação seja lenta, o *SpaceBrew* não é adequado caso contrário, seu uso é apropriado para esta e outras aplicações. A maior dificuldade do trabalho foi realizar a comunicação entre dois *masters* e validar os comandos do ROS, pois se o comando enviado não existir ou estiver com a sintaxe incorreta, a aplicação não realiza o trabalho para o qual ela foi proposta. Portanto, para facilitar a manutenção, o software foi dividido em partes e cada parte é responsável por uma função: *rostopic*, *rosservice*, *rosrun* e *roscommands* e cada uma delas tem uma seção dedicada. Toda a comunicação remota foi realizada utilizando-se o *SpaceBrew*, descrito na seção 2.5.1, pois com essa ferramenta não é necessário encontrar manualmente o endereço IP do destino. Todo o endereçamento e comunicação é feito automaticamente o que auxilia o desenvolvimento do projeto.

O ROSRemote pode ser utilizado tanto para controlar quanto para monitorar robôs e esse segundo uso talvez seja o mais comum e indicado, pois os dados que trafegam pela internet estão sujeitos a atrasos na entrega. Esse atraso pode acarretar em danos ao robô, caso ele esteja sendo controlado, mas não no caso em que ele simplesmente esteja sendo

monitorado. Além disso, no segundo caso talvez não importe para o usuário o tempo que o dado demora para trafegar, mas sim que ele consiga se atualizar com a situação do robô.

Para trafegar dados pelo servidor do *SpaceBrew* é necessário, primeiramente, a criação de um cliente local e um remoto que serão interligados. O cliente local é responsável por enviar os comandos e o remoto executa os comandos e envia a resposta após a execução. O *SpaceBrew* foi criado utilizando a linguagem JavaScript, portanto para a criação desse cliente foi necessário realizar uma ligação entre esta linguagem e o Python. A princípio houve a tentativa de se utilizar *WebSockets* para comunicar com o servidor, porém não houve sucesso. Então foi utilizado o *PySpaceBrew* (MAYER, 2013), uma biblioteca que auxilia os clientes do *SpaceBrew* a serem implementados utilizando a linguagem Python. Essa biblioteca foi necessária, pois o ROS não possui suporte a JavaScript até o momento.

Assim como o ROS, o *SpaceBrew* também trabalha com "publicadores" e "assinantes" e eles são necessários para a criação de um cliente. É possível que um mesmo cliente envie e receba dados, porém para a comunicação entre dois ou mais objetos, é necessário dois ou mais clientes. No Algoritmo 2 é possível entender a criação desses clientes.

Algoritmo 2: Código para criação dos clientes

```
1 brew = Spacebrew(name=name, server=server)
2 brew.addPublisher("Publisher")
3 brew.addSubscriber("Subscriber")
4 try:
5   brew.start()
6   brew.subscribe("Subscriber", received)
```

A linha 1 do Algoritmo 2 é responsável por criar um objeto *SpaceBrew* que contém o nome da aplicação e o servidor ao qual ela deverá se conectar. Como o grupo Rockwell disponibiliza um servidor, esse foi utilizado para testar a aplicação, o endereço é "sandbox.spacebrew.cc". Nas linhas 2 e 3 podem ser observadas as criações do *publisher* e do *subscriber* para o cliente e dentro do comando *try*, o *SpaceBrew* é inicializado e a função de *callback* é definida. Assim como no ROS, o *SpaceBrew* também trabalha com esse tipo de função, que nada mais é do que a indicação do que deverá ser executado após um comando. Em ROS é comum utilizar essa função após chamar um serviço, ela indica o que deverá ser realizado após o serviço ser invocado. No *SpaceBrew*, a função indica o que deverá ser feito após receber uma mensagem que veio de um cliente remoto. Após a criação desse cliente, mensagens já podem ser enviadas e recebidas através da internet e é por meio delas que ocorre a comunicação nesse trabalho.

Como mostra a Figura 3.1.1, a comunicação ocorre de forma simples entre os dois *Masters*, um deles recebe os comandos do usuário e envia para o controle do robô remoto.

A aplicação foi dividida para facilitar a manutenção do código. Porém, algumas partes comuns são utilizadas em vários momentos, foram inseridas no arquivo principal e são chamadas por todas as outras classes. Como é possível observar no diagrama da Figura 3.1.2,

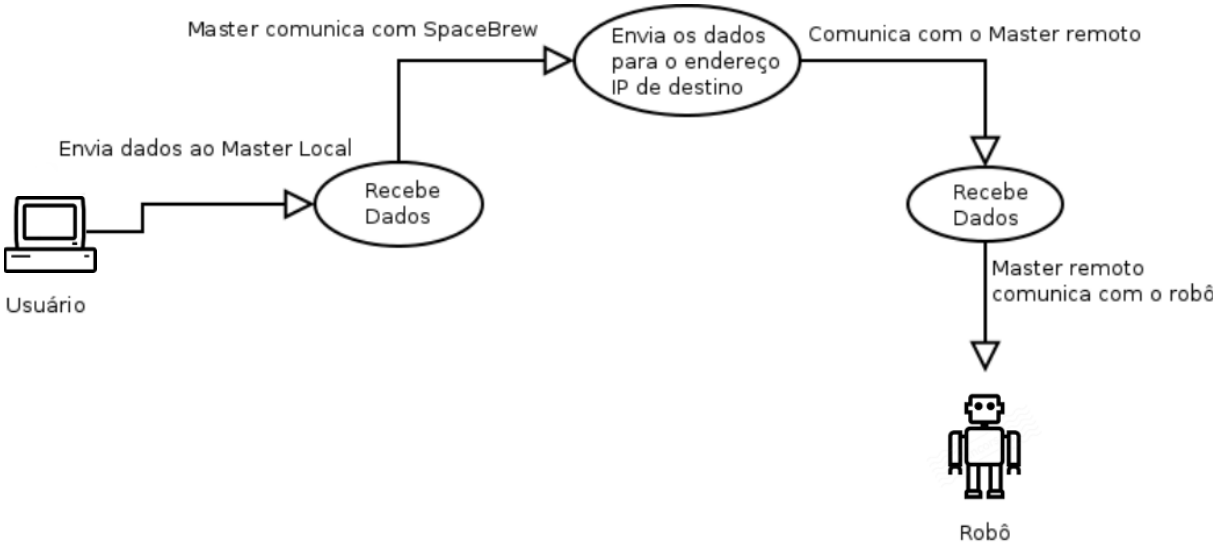

Figura 3.1.1 – Visão geral da comunicação do ROSRemote

a classe principal da aplicação se comunica com todas as outras bidirecionalmente, no entanto classes que não são a principal não podem se comunicar entre si, isso se deve ao fato de que não é possível chamar dois comandos do ROS simultaneamente.

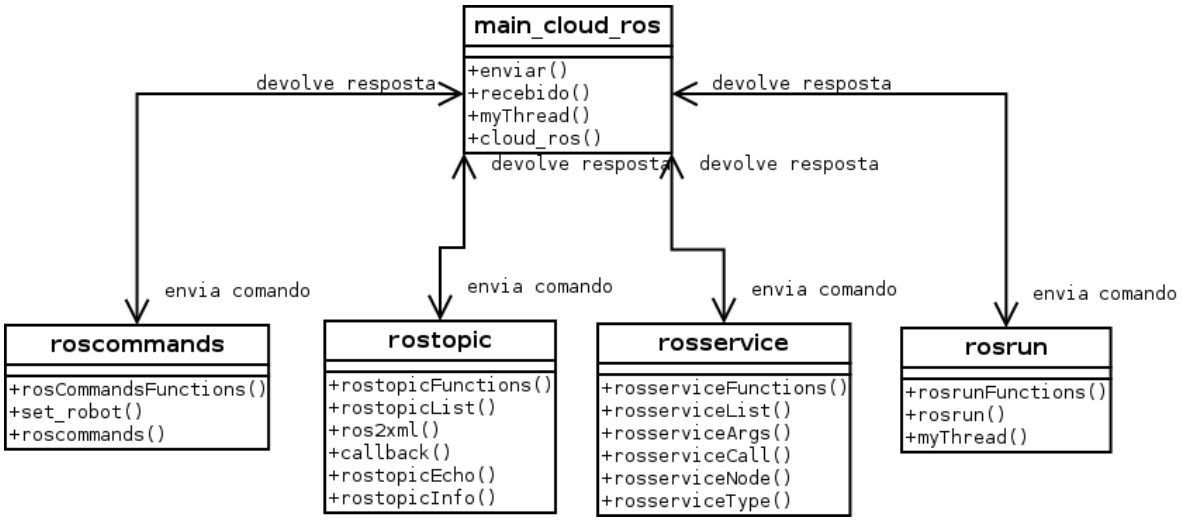

Figura 3.1.2 – Diagrama UML da aplicação ROSRemote.

A função da classe principal da aplicação é apenas redirecionar o comando para a classe correta, tanto no momento do envio quanto no recebimento. Além disso possui uma *thread* (que são linhas de execução de um programa) que lê todas as teclas pressionadas no terminal para verificar se algumas delas é um comando a ser enviado para o controle do robô. O trecho do código apresentado no Algoritmo 3 demonstra como é feita a separação do comando que deverá ser enviado

Algoritmo 3: Código para direcionar o envio de comandos.

```
1  def send(req):
2    global brew
3    command = req.command.split(“ ”)
4    if(command[0] == “rostopic”):
5      rostopicFunctions(req.command, brew)
6    elif(command[0] == “rosservice”):
7      rosserviceFunctions(req.command, brew)
8    elif(command[0] == “rosrun”):
9      rosrunFunctions(req.command, brew)
10   elif(command[0] == “roscommands”):
11     rosCommandsFunctions(req.command, brew)
12   else:
13     rospy.logwarn(“Incorrect command syntax”)
```

A variável “brew” é um objeto para a classe *SpaceBrew*, ela mantém os dados da criação do cliente e é utilizada para enviar os comandos, esse envio será detalhado nos capítulos 3.2, 3.3, 3.4 e 3.5. O comando é recebido através da variável *req* e separado pelos espaços, logo verifica-se a qual classe ele deverá ser enviado. Por exemplo, se o usuário insere o comando simples *rostopic list*, que lista todos os tópicos atuais, a função o separa em duas palavras e envia o comando para a classe *rostopic* 3.2.

A função “send” é requisitada pelo serviço do ROS, ou seja, ela é a função de *callback* do serviço, diferente da função “received”, que é a função de *callback* do *SpaceBrew*. Ao se chamar o serviço /send_data criado pelo ROSRemote, os comandos são enviados para o *Master* remoto. O código referente a esse trecho é mostrado no Algoritmo 3. Esse serviço, juntamente com o Nó que irá publicá-lo, são criados na função principal da aplicação e são mostrados no Algoritmo 4. A primeira linha inicia o nó e a terceira cria o serviço.

Algoritmo 4: Código de criação do nó e serviço do ROS.

```
1  rospy.init_node(‘cloud_ros_node’)
2  rospy.loginfo(“cloud_ros node is up and running!!!”)
3  s = rospy.Service(‘send_data’, command, send)
```

Para chamar o serviço no terminal do Linux basta digitar *“rosservice call”*, que é o comando para os serviços do ROS e, em seguida, digitar o nome do serviço antecedido por barra, nesse caso /send_data. Ao final basta escrever, entre aspas, o comando a ser enviado remotamente, por exemplo, *“rostopic list”*, como no Algoritmo 5.

Algoritmo 5: Exemplo de utilização do serviço.

```
1  rosservice call \send_data “rostopic list”
```

Da mesma forma que há a função que envia os dados, também há uma para recebê-los após serem transferidos. No Algoritmo 6 é possível salientar alguns trechos importantes dessa função. Na linha 3 é possível notar que há uma comparação do tipo de comando, nesse caso se o comando enviado é um *rostopic*, cria-se uma chamada dinâmica (que é o ato de criar, dinamicamente, uma chamada para a função desejada) à função desejada utilizando a variável *method*. Na linha 5, são enviados os parâmetros adicionais necessários para a função como o cliente do *SpaceBrew* e o tópico desejado. Caso o comando não seja um *rostopic*, o código é capaz de enviar para o local correto utilizando algumas comparações. Ao final, caso o tipo de ação não seja o envio de um comando, significa que o dado recebido é uma resposta da requisição, então na linha 16 do Algoritmo 6, ocorre a impressão dos dados.

Algoritmo 6: Código para direcionar o recebimento dos comandos no *master* remoto

```
1  def recebido(data):
2    global brew
3    if data[‘action’]==“send” and data[‘commandRos’]=‘rostopic’:
4      method = getattr(rostopic, data[‘function’])
5      result = method(brew, data[‘topic’])
6    elif data[‘action’]==“send” and data[‘commandRos’]==‘rosservice’:
7      method = getattr(rosservice, data[‘function’])
8      result = method(brew, data[‘service’], data[‘args’])
9    elif data[‘action’]==“send” and data[‘commandRos’]==‘rosrun’:
10     method = getattr(rosrun, data[‘function’])
11     result = method(brew, data[‘package’], data[‘executable’], data[‘parameters’])
12   elif data[‘action’]==“send” and data[‘commandRos’]==‘roscommands’:
13     method = getattr(roscommands, data[‘function’])
14     result = method(brew, data[‘commands’])
15   else:
16     rospy.logwarn(data[’title’]+”\n”+data[’datum’])
```

O modelo geral utilizado para o desenvolvimento da aplicação pode ser visualizado na Figura 3.1.3 e funciona da seguinte forma:

- O usuário insere o comando no terminal local do ROS, chamando o serviço que dá início ao programa (/send_data);
- A classe principal envia para a classe correspondente, onde o comando é processado;
- Cada classe envia a requisição por meio do *SpaceBrew* para o *Master* remoto;

- O *Master* remoto recebe os dados através da função *main* e reenvia para a classe correspondente ao pedido no *Master* remoto;
- O comando é executado no robô ou no *Master* remotos e o resultado é devolvido através do *SpaceBrew* para o *Master* local, onde se encontra o usuário.

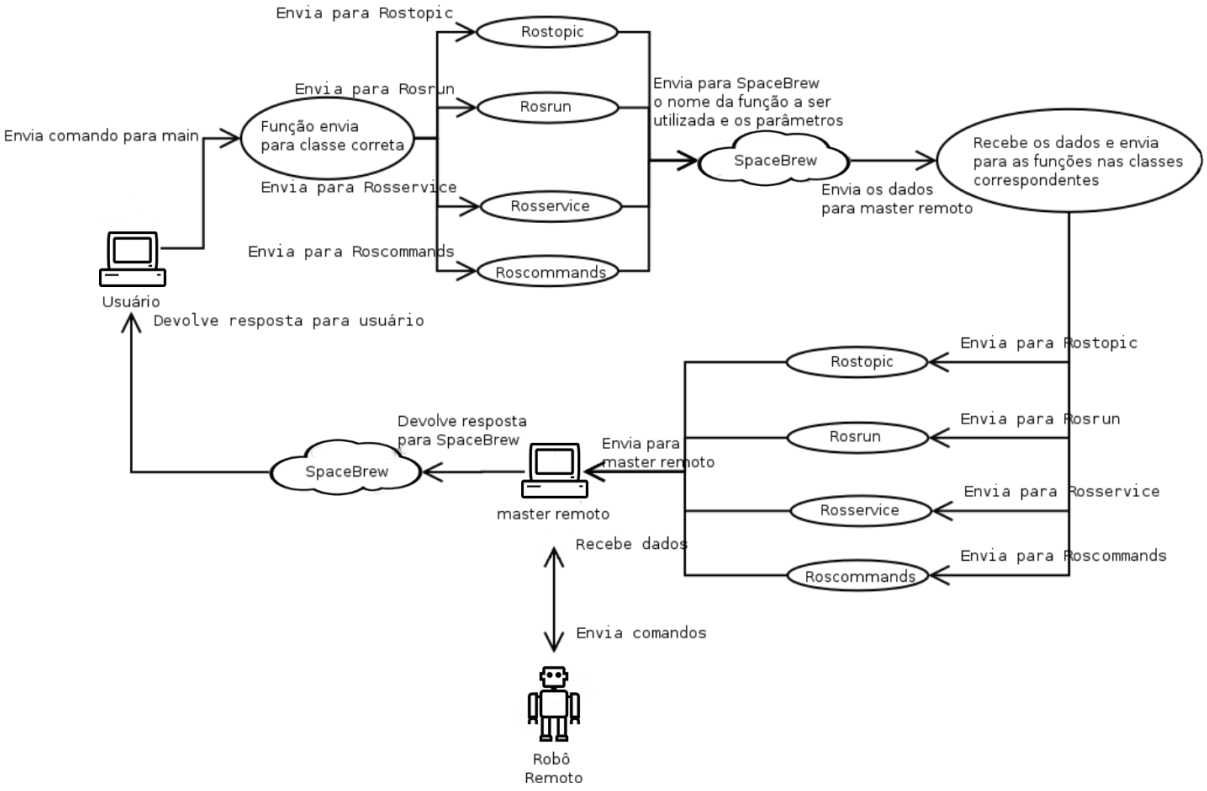

Figura 3.1.3 – Fluxo de dados da aplicação ROSRemote.

Em resumo, para atingir o objetivo desse trabalho foi necessário recriar trechos do ROS para que, ao invés das funções serem executadas localmente, elas fossem enviadas e executadas remotamente. A maneira como ele foi recriado e as partes do programa que realizam essas funções são demonstradas nas seções 3.2, 3.3, 3.4 e 3.5.

3.2 Rostopic

O *Rostopic* é um conjunto de ferramentas de linha de comando que possui todos os comandos necessários para que o usuário possa recuperar informações sobre Tópicos existentes e que estejam publicando quaisquer dados. É possível recuperar os Tópicos atuais, publicadores, assinantes e mensagens. Atualmente a aplicação descrita nesse trabalho abrange alguns comandos principais dessa ferramenta, como:

- List: lista todos os Tópicos ativos;
- Echo: imprime as mensagens que são publicadas em um Tópico;

- Info: exibe as informações do Tópico, incluindo assinantes e publicadores;

Dentro da classe *rostopic*, primeiramente o que ocorre é a decisão de qual o tipo de comando foi enviado. De acordo com essa decisão, o nome do método correspondente é passado por meio de uma variável para o *SpaceBrew* e ao receber os dados no lado remoto, a classe sabe exatamente qual função deverá ser chamada. O fluxo desses dados é mostrado na Figura 3.2.1. A responsável por essa decisão é a função *rostopicFunctions*, que realiza algumas comparações a fim de descobrir qual comando foi enviado para o *Master* remoto e caso não seja encontrado nenhum comando correspondente, uma mensagem de sintaxe incorreta é impressa.

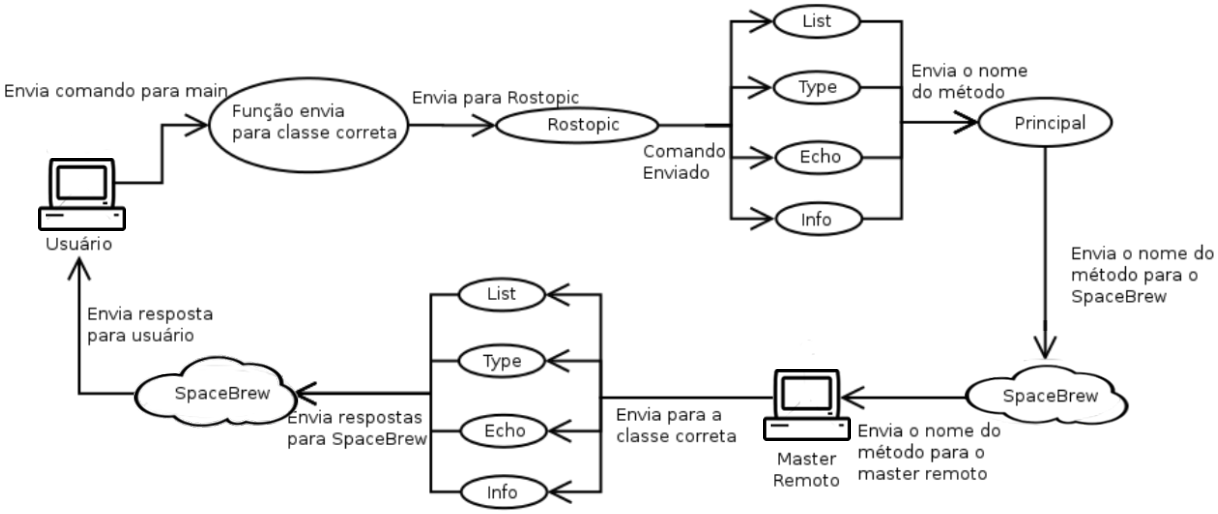

Figura 3.2.1 – Fluxo de dados do *Rostopic*.

Algoritmo 7: Código que indica qual o comando dado e devolve o nome da função correspondente.

```
1  commandSplit = command.split(“ ”)
2  if commandSplit[1] == “list”:
3    data = {‘commandRos’:‘rostopic’, ‘function’:‘rostopicList’, ‘action’:‘send’,
     ‘topic’:‘’}
4    brew.publish(“Publisher”, data) rospy.logwarn(“Command sent = ”+command)
5  elif comandoSplit[1] == “echo”:
6    if len(commandSplit) != 4:
7      rospy.logwarn(“syntax = rostopic echo /topic frequency”)
8    else:
9      data = {‘commandRos’:‘rostopic’, ‘function’:‘rostopicEcho’, ‘action’:‘send’,
     ‘topic’:comandoSplit[2][1:]}
10     brew.publish(“Publisher”, data)
11     rospy.logwarn(“Command sent = ”+command)
12 elif comandoSplit[1] == “info”:
13   data = {‘commandRos’:‘rostopic’, ‘function’:‘rostopicInfo’, ‘action’:‘send’,
     ‘topic’:commandSplit[2]}
14   brew.publish(“Publisher”, data)
15   rospy.logwarn(“Command sent = ”+command)
16 elif comandoSplit[1] == “type”:
17   data = {‘commandRos’:‘rostopic’, ‘function’:‘rostopicType’, ‘action’:‘send’,
     ‘topic’:comandoSplit[2]}
18   brew.publish(“Publisher”, data)
19   rospy.logwarn(“Command sent = ”+command)
```

O Algoritmo 7 é basicamente igual em sua totalidade, diferenciando-se apenas nas variáveis que são enviadas. Na linha 3, a variável *‘topic’* é enviada vazia, enquanto na linha 9 ela é preenchida com os dados inseridos pelo usuário. Isso se dá pelo fato de que o comando *rostopic list* não necessita do envio de um Tópico específico para funcionar, mas o *rostopic echo* sim.

No Algoritmo 7, a linha 1 serve apenas para separar o comando pelos espaços a fim de facilitar as comparações. Dentro dos *ifs* de comparação ocorre uma atribuição dos dados necessários à variável *data*: o *‘commandRos’* recebe o tipo do comando; a *‘function’* recebe o nome da função correspondente; o *‘action’* indica apenas que os dados devolvidos deverão ser enviados para o *Master* remoto e; o *‘topic’* indica o nome do tópico desejado ou é devolvido vazio para a classe principal, caso não seja necessário enviar o nome do Tópico.

O comando *rospy.logwarn* imprime o comando enviado no terminal. Esse comando

serve para *debug* do código e também para que o usuário saiba se o comando que enviou está correto ou possui algum erro.

Os valores que são atribuídos à variável *'function'* são os nomes das funções utilizadas de acordo com o comando enviado pelo usuário. Dessa maneira é possível realizar chamadas dinâmicas a elas, como apresentado no algoritmo 6. As funções secundárias do código serão descritas nas seções 3.2.1, 3.2.2 e 3.2.3.

3.2.1 Rostopic List

Esse comando é um dos mais utilizados no ROS e serve para que o usuário possa visualizar uma lista de quais tópicos estão sendo publicados no momento (ROS, 2016f). Na aplicação descrita neste trabalho, foi necessário criar uma chamada ao comando *rostopic list* através do Python. Essa chamada é enviada para o *Master* remoto e esse devolve os tópicos que estão sendo publicados no momento. O código que realiza esse procedimento é apresentado no Algoritmo 8, ele recupera os tópicos publicados e os envia para o usuário por meio do *SpaceBrew*.

Algoritmo 8: Código que recupera os tópicos publicados remotamente e envia para o usuário.

```
1 master = rosgraph.masterapi.Master('/rostopic')
2 resp = master.getPublishedTopics('/')
3 datum = ""
4 for i in range(0, len(resp)):
5   datum += "\n"+resp[i][0]
6 data = {'datum':datum, 'title':"Rostopic list results", 'action':'receive'}
7 brew.publish("Publisher", data)
```

A linha 1 cria um objeto do tipo *Master*, utilizado para chamar a função necessária para recuperar os tópicos, nesse algoritmo essa função é o *getPublishedTopics*. Após recuperar os Tópicos, todos os dados são adicionados na variável datum, empacotados em um JSON e devolvidos para o usuário através do *SpaceBrew*. Esses dados são impressos ao usuário utilizando a última linha da função principal explicada no Algoritmo 6.

3.2.2 Rostopic Echo

Além do *rostopic list* também foi implementado o *rostopic echo* e esse foi o código mais complicado de ser criado devido a alguns fatores. Primeiramente, para chamar a função de *callback* é necessário saber a qual tipo ela pertence, caso contrário, não é possível realizar a chamada. Outra dificuldade na criação desse trecho de código é que para realizar o envio da resposta com todos os dados recebidos, é necessário converter os dados em um xml, além de devolver a resposta em uma taxa pré-definida pelo usuário. Para sobrepor

esses obstáculos, foram criados códigos específicos para cada tarefa. O trecho principal do *rostopic echo* é apresentado no Algoritmo 9

Algoritmo 9: Código que recupera os dados do *rostopic* echo e envia para a criação do xml que é devolvido ao usuário.

```
1 proc = subprocess.Popen([“rostopic type ”+topic], stdout=subprocess.PIPE,
    shell=True)
2 (dados, err) = proc.communicate()
3 dado = dados.split(“/”)
4 dado[0] += “.msg”
5 mod = __import__(dado[0], fromlist=[dado[1]])
6 klass = getattr(mod, dado[1].strip())
7 rospy.Subscriber(topic, klass, callback)
```

A linha 1 recupera o tipo do tópico desejado e a chamada *subprocess.Popen* envia o comando diretamente ao terminal e recebe a resposta do mesmo. O dado é separado pelas barras para ser possível utilizar o nome do tópico e seu tipo separadamente. A linha 7 serve para realizar a chamada do *rostopic echo* e enviar o nome do tópico e classe para a função *callback*, visto que ela é a responsável por converter os dados para xml e devolver para o usuário local. A função de *callback* simplesmente recebe os dados, chama a função de conversão para xml e envia os dados para o *SpaceBrew*.

Algoritmo 10: Código que envia a resposta do rostopic echo para ser convertido em xml e devolve a resposta para o *SpaceBrew*.

```
1 while(stop_ == False):
2   xml = ros2xml(resp, “”)
3   data = {‘datum’:xml, ‘title’:“Rostopic echo results ”, ‘action’:‘receive’}
4   brew_.publish(“Publisher”, data)
5   time.sleep(freq)
```

No algoritmo 10 é possível notar que a linha 2 envia a resposta para a função de conversão para xml e o recebe pronto. A linha 3 cria o JSON enquanto a linha 4 o envia para o *SpaceBrew*. Esse trecho auxilia na resolução do problema de recuperação e envio dos dados necessários. Já o método que converte a resposta do ROS para xml foi encontrado em um fórum e pode ser visto em (ANSWERS, 2011).

3.2.3 Rostopic info

O *rostopic info* foi o comando mais facilmente implementado dentro dessa classe, possui poucas linhas, uma para enviar ao terminal o comando desejado e uma linha para receber a resposta do terminal. Além disso também possui uma linha que monta o JSON e outra

que o envia de volta para o usuário. O algoritmo completo para o *rostopic info* é mostrado no Algoritmo 11

Algoritmo 11: Código que envia a resposta do *rostopic info* para o *SpaceBrew*.

```
1 proc = subprocess.Popen(["rostopic info "+topic], stdout=subprocess.PIPE,
    shell=True)
2 (dados, err) = proc.communicate()
3 data = 'datum':datum, 'title':'Rostopic info results '+topic, 'action':'receive'
4 brew.publish("Publisher", data)
```

3.3 Rosservice

O *Rosservice* (ROS, 2016d) é um conjunto de ferramentas de linha de comando que contém os comandos necessários para que seja possível a manipulação de serviços existentes no ROS, como chamar e listar os mesmos. Entre os vários comandos dessa ferramenta, os principais foram implementados nesse trabalho:

- List: lista todos os serviços atualmente disponíveis;
- Args: imprime os argumentos necessários para realizar a chamada ao serviço;
- Call: chama o serviço enviando os argumentos necessários;
- Node: imprime o nome do nó que fornece um serviço;
- Type: imprime o tipo do serviço.

Assim como ocorre com o *Rostopic*, descrito na seção 3.2, a classe *Rosservice* recebe o comando e decide para onde a requisição deve ser enviada utilizando o nome da função que deve ser executada. Essa função é armazenada em uma variável e enviada através do *SpaceBrew* que recebe os dados no *Master* remoto e realiza a operação correspondente. O fluxo de dados do *Rosservice* é apresentado na Figura 3.3.1.

Para padronizar os nomes de variáveis e funções, eles são muitos similares. Como exemplo pode ser citado o *rostopicFunctions* do algoritmo 7, que lida com as funções da classe *rostopic*, e a função *rosserviceFunctions*, que lida com as funções do *Rosservice*.

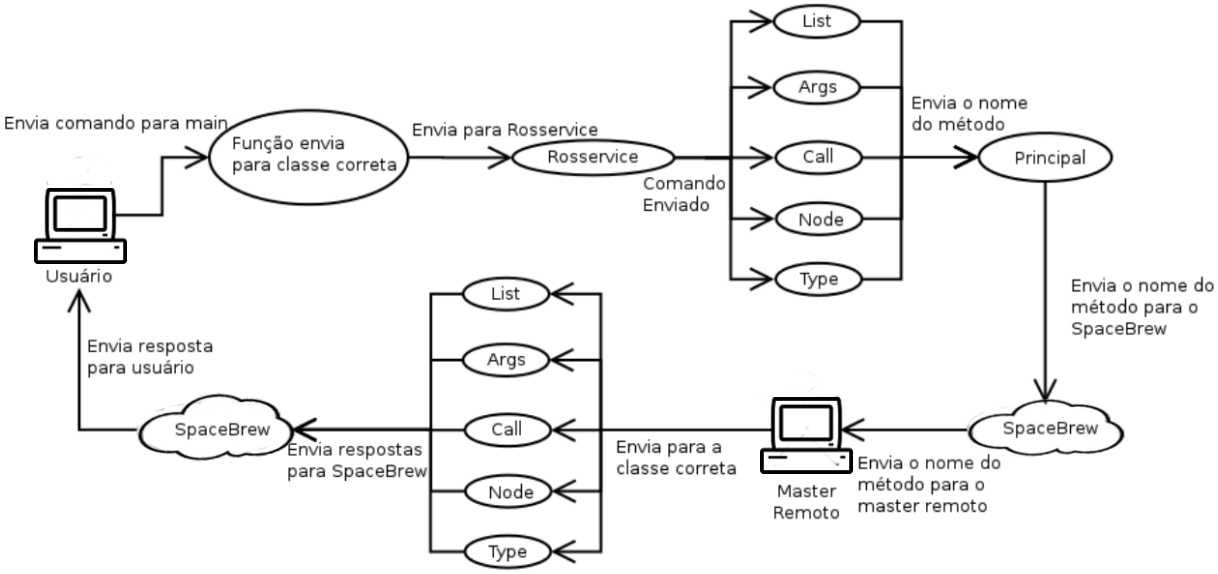

Figura 3.3.1 – Fluxo de dados do *Rosservice*.

Algoritmo 12: Trecho do código que indica qual o comando dado e devolve o nome da função correspondente.

```
1  elif commandSplit[1] == “call”:
2    if len(commandSplit) < 3:
3      rospy.logwarn(“syntax = rosservice call /service”)
4    elif len(commandSplit) == 3:
5      data = {‘commandRos’:‘rosservice’, ‘function’:‘rosserviceCall’, ‘action’:‘send’,
     ‘service’:commandSplit[2], ‘args’:‘’}
6      brew.publish(“Publisher”, data)
7      rospy.logwarn(“Command sent = ”+command)
8    else:
9      argsSplit = comando.split(“‘”)
10     data = {‘commandRos’:‘rosservice’, ‘function’:‘rosserviceCall’, ‘action’:‘send’,
     ‘service’:commandSplit[2], ‘args’:argsSplit[1]}
11     brew.publish(“Publisher”, data)
12     rospy.logwarn(“Command sent = ”+command)
```

O Algoritmo 12 possui um trecho do código responsável por decidir qual a função e como ela deve ser executada. A comparação da linha 2 serve apenas para verificar se a sintaxe está correta e caso não esteja, uma mensagem é enviada ao usuário. Além disso, se a sintaxe estiver correta ainda é necessário identificar os serviços entre os dois tipos existentes: os que possuem argumentos e os que não possuem. No primeiro caso, o usuário deverá enviar apenas os comandos *rosservice call /nome_do_serviço*, já no segundo caso é necessário informar também os argumentos necessários *rosservice call /nome_do_serviço*

"argumento1, argumento2, ..." e assim por diante.

O nome da função é enviado para o *SpaceBrew* para que ela possa ser executada no *Master* remoto. Se houver argumentos, eles são enviados juntamente dentro do mesmo JSON por meio da variável *'args'* e caso contrário, ela é enviada vazia. Isso ocorre para todos os comandos já citados, porém como o código é extenso e basicamente o mesmo, não é necessário que seja apresentado por completo. As seções 3.3.1, 3.3.2, 3.3.3, 3.3.4 e 3.3.5 explicam cada uma das funções inseridas nessa classe.

3.3.1 Rosservice List

O *Rosservice List* lista todos os serviços disponíveis no momento. No caso do *Rosservice List* o código apenas envia o comando, recupera a resposta, a encapsula em um JSON e devolve ao *SpaceBrew*, como demonstrado no Algoritmo 13.

Algoritmo 13: Código que recupera os serviços rodando no ROS e devolve para o *SpaceBrew*.

```
1 proc = subprocess.Popen(["rosservice list"], stdout=subprocess.PIPE, shell=True)
2 (dados, err) = proc.communicate()
3 data = {'datum':datum, 'title':'Rosservice list results', 'action':'receive'}
4 brew.publish("Publisher", data)
```

Toda a validação da sintaxe é feita na função principal da classe (*rosserviceFunctions*) indicada no Algoritmo 12. Isso é importante porque nessa parte do código o processamento ainda é feito de forma local e caso ocorra algum problema, os dados não são enviados para o *Master* remoto. Essa ação economiza tempo, pois processamentos locais ocorrem muito mais rápido do que os remotos.

3.3.2 Rosservice Args

O *Rosservice Args* é utilizado para mostrar os argumentos necessários para se chamar um serviço e pode ser chamado simplesmente enviando como parâmetro o serviço o qual se deseja obter a informação. Desse modo, basta inserir o comando *rostopic args /nome_do_serviço* e o terminal imprime todos os argumentos necessários, bem como os tipos dos mesmos.

O algoritmo que realiza essa operação envia o comando para o terminal remoto, recupera a resposta e a devolve através do *SpaceBrew*, como é possível verificar no Algoritmo 14.

Algoritmo 14: Código que mostra os argumentos necessários para se chamar um serviço.

```
1 proc = subprocess.Popen([“rosservice args ”+service], stdout=subprocess.PIPE,
    shell=True)
2 (dados, err) = proc.communicate()
3 data = {‘dados’:dados, ‘title’:‘rosservice type results ’+service, ‘action’:‘receive’}
4 brew.publish(“Publisher”, data)
```

3.3.3 Rosservice Call

O *Rosservice Call* serve para iniciar um serviço enviando ou não argumentos, já que nem sempre um serviço necessita de dados de entrada. Portanto foi necessário realizar uma validação para verificar se o usuário enviou ou não argumentos para a chamada ao serviço e após esse processo, a resposta é recuperada do terminal e enviada novamente ao usuário quando necessário.

Algoritmo 15: Código que inicia um serviço com ou sem argumentos.

```
1 if args==“”:
2   proc = subprocess.Popen([“rosservice call ”+service], stdout=subprocess.PIPE,
    shell=True)
3 else:
4   proc = subprocess.Popen([“rosservice call ”+service+“ ‘”+args+“’”],
    stdout=subprocess.PIPE, shell=True)
5 (dados, err) = proc.communicate()
6 data = {‘datum’:datum, ‘title’:‘service started: ‘+service+’ ‘+args,
    ‘action’:‘receive’}
7 brew.publish(“Publisher”, data)
```

Nem sempre um serviço necessita devolver uma resposta, em alguns casos ele simplesmente realiza determinadas tarefas e pausa o processamento. O código apresentado no Algoritmo 15 funciona tanto para quando o usuário necessita de uma resposta quanto para quando ela não é necessária.

3.3.4 Rosservice Node

O *Rosservice Node* é responsável por imprimir o nó do ROS que está fornecendo um serviço. Essa chamada não necessita de argumentos, portanto sua utilização e implementação são simples. O Algoritmo 16 mostra como ele foi criado e funciona exatamente da mesma maneira que o *Rosservice List*, descrito na seção 3.3.1 e o *Rosservice Args*, descrito

na seção 3.3.2.

Algoritmo 16: Código que devolve ao usuário o nó responsável por um serviço oferecido pelo ROS.

```
1 proc = subprocess.Popen([“rosservice node ”+service], stdout=subprocess.PIPE,
    shell=True)
2 (dados, err) = proc.communicate()
3 data = {‘dados’:dados, ‘title’:‘Rosservice node results ’+service, ‘action’:‘receive’}
4 brew.publish(“Publisher”, data)
```

3.3.5 Rosservice Type

O *Rosservice Type* é responsável por mostrar o tipo do serviço desejado. Ele difere das outras funções por poder ser utilizado com um recurso extra que permite a impressão do arquivo srv (ROS, 2016e) utilizado na criação do serviço.

O comando pode ser ativado utilizando-se *rostopic type /nome_do_serviço*, porém para realizar a impressão do arquivo srv no terminal, é necessário enviar parâmetros extras após o nome_do_serviço, como o *rossrv show*. Vale ressaltar que é necessário separar as duas chamadas utilizando-se uma barra vertical (|)

O Algoritmo 17 refere-se ao *rosservice type* e já possui a possibilidade de se utilizar o campo extra se o usuário desejar verificar o arquivo srv juntamente com o tipo do serviço.

Algoritmo 17: Código que devolve ao usuário o tipo do serviço requisitado.

```
1 if args==“”:
2   proc = subprocess.Popen([“rosservice type ”+service], stdout=subprocess.PIPE,
    shell=True)
3 else:
4   proc = subprocess.Popen([“rosservice type ”+service +“ | ”+ args],
    stdout=subprocess.PIPE, shell=True)
5 (datum, err) = proc.communicate()
6 data = {‘datum’:datum, ‘title’:‘Rosservice type results ’+service, ‘action’:‘receive’}
7 brew.publish(“Publisher”, data)
```

3.4 Rosrun

O *Rosrun* é uma ferramenta que permite que o usuário possa utilizar um executável em um pacote do ROS que se encontra em qualquer local do computador (ROS, 2016). Para tanto, basta utilizar o comando *rosrun /nome_do_pacote /nome_do_executável* e o executável é iniciado.

O *rosrun* faz com que o terminal entre em modo de espera e não prossiga o processamento, para utilizar o pacote iniciado é necessário que o usuário abra um novo terminal.

Nesse trabalho, essa possibilidade não existia, pois uma vez que o serviço está em funcionamento, não há maneiras de se utilizar outro terminal. Essa utilização acarretaria na criação de um novo cliente *SpaceBrew*, fazendo com que a aplicação não funcionasse corretamente e para resolver esse problema foram utilizadas *Threads*. Um programa comum possui somente uma *Thread*, isso quer dizer que só pode realizar um processamento por vez. Para que seja possível a execução de dois processos ao mesmo tempo, é necessária a criação de *Threads*, uma que executará o processo secundário e outra que continuará com o principal. No caso do ROSRemote, uma *Thread* sempre lida com o processo principal e outra é criada quando o *rosrun* é utilizado, isso garante que o pacote funcione corretamente e permite que o terminal continue livre para receber outros comandos.

Assim como outros comandos o *rosrun* possui uma função principal, porém essa não é responsável pela distribuição dos comandos para os locais corretos, visto que o *rosrun* não possui variações como *list* ou *echo*. A função principal somente verifica se o comando foi corretamente inserido, sendo enviado parâmetros extras ou não e no segundo caso são enviados vazios.

Algoritmo 18: Código que verifica se o comando *rosrun* foi inicializado utilizando-se parâmetros extras ou não.

```
1 if len(commandSplit) == 3:
2   data = {'commandRos':'rosrun', 'function':'rosrun', 'action':'send',
    'package':commandSplit[1], 'executable':commandSplit[2], 'parameters':''}
3   brew.publish("Publisher", data)
4   rospy.logwarn("Command sent = "+comando)
5 elif len(CommandSplit) == 4:
6   data = {'commandRos':'rosrun', 'function':'rosrun', 'action':'send',
    'package':commandSplit[1], 'executable':commandSplit[2],
    'parameters':commandSplit[3]}
7   brew.publish("Publisher", data)
8   rospy.logwarn("Command sent = "+command)
```

As linhas 1 e 5 do Algoritmo 18 servem para decidir se o usuário enviou um parâmetro extra ou não. Caso não tenha enviado, a variável *parameters* é iniciada como vazia e enviada para o *SpaceBrew*, como apresentado ao final da linha 2.

Ao se enviar algum parâmetro no ROS é necessário acrescentar ao final do nome do executável do pacote, o nome do parâmetro e seu valor da seguinte forma: *nome_do_parâmetro := valor_do_parâmetro*. Ao implementar essa função na aplicação, foi constatado que os caracteres ":=" constituem um comando reservado do Python e portanto, não poderiam ser utilizados. Para sobrepor esse obstáculo, ao invés de ":=" foi utilizado "@", isso significa que o usuário, ao chamar o executável do pacote com *rosrun* e enviar parâmetros, deve fazê-lo inserindo o comando *nome_do_parâmetro@valor_do_parâmetro*. O ROSRemote fica

responsável por realizar as substituições internamente a fim de acionar o executável. O Algoritmo 19 mostra não somente como ocorre essa substituição, mas também como é feita a chamada ao *rosrun* na aplicação.

Algoritmo 19: Código que substitui "@" por ":=" e envia o comando *rosrun* para o *master* remoto.

```
1  parameters = parameters.replace("@", ":=")
2  global proc
3  if(parameters != ''):
4   proc = subprocess.Popen(["rosrun " +package +" "+ executable+" "+parameters],
   stdout=subprocess.PIPE, shell=True)
5  else:
6   proc = subprocess.Popen(["rosrun " +package +" "+ executable],
   stdout=subprocess.PIPE, shell=True)
7  data = {'datum':datum, 'title':'Package is running ', 'action':'receive'}
8  thread1 = myThread(1, "Thread-1", 1)
9  thread1.start()
10 brew.publish("Publisher", data)
```

A linha 1 simplesmente realiza a substituição do "@" pelo ":=", com isso é possível prosseguir com o restante da operação sem maiores problemas. As linhas 4 e 6 iniciam o executável do pacote com o *rosrun*, enviando para o terminal remoto o comando com ou sem parâmetros, respectivamente. Após essa etapa uma nova *Thread* é criada para que o usuário possa continuar utilizando a aplicação e uma mensagem é enviada indicando que o pacote está pronto para ser utilizado.

As linhas 8 e 9 do Algoritmo 19 criam e chamam a função que inicia a *Thread*. Para utilizar esse recurso foi necessário construir uma classe chamada myThread, mostrada no Algoritmo 20. O comando *run* é responsável por iniciar a *Thread* após ela ser criada com id, nome e um contador. Para chamá-la é utilizada a função *start* que apenas realiza a comunicação com o *Master* remoto para ativar o executável que o usuário deseja.

Algoritmo 20: Código que cria a *thread*.

```
1 class myThread (threading.Thread):
2   def __init__(self, threadID, name, counter):
3     threading.Thread.__init__(self)
4     self.threadID = threadID
5     self.name = name
6     self.counter = counter
7   def run(self):
8     global proc
9     (datum, err) = proc.communicate()
```

Feito isso, a *Thread* criada fica responsável pela execução do pacote e a principal continua esperando mais comandos do usuário. Da mesma forma isso ocorre com o *roscommands*, que é explicado na seção 3.5.

3.5 Roscommands

O *roscommands* é uma ferramenta inexistente no ROS e foi criada para que comandos externos ao ROS possam ser enviados, como exemplo pode ser citado o já implementado *teleop*, que serve para movimentar o robô. Outros comandos podem ser implementados futuramente, conforme necessário.

Da mesma forma que no *Rosrun*, no *Roscommands* também foi necessário a utilização de *Threads*, pois é preciso manter o terminal sempre esperando por um comando do usuário. Essa *Thread* foi inserida na classe principal do programa e inicializada para ler qualquer tecla digitada pelo usuário. Quando uma tecla é pressionada no teclado, o comando é executado, enviado pelo *SpaceBrew* e o *Master* remoto o processa quando for necessário, caso nada seja pressionado, o terminal fica em estado de espera.

Para implementar essa classe foram necessários alguns cuidados. Ao se enviar um comando para o robô se movimentar, é necessário saber qual robô deve responder a requisição, para isso foi criada uma função auxiliar denominada *set_robot*, que recebe como parâmetro os nomes dos robôs que devem responder ao comando. Na aplicação foi implementada a possibilidade de movimentar vários robôs ao mesmo tempo, para isso basta enviar os nomes de todos os robôs como parâmetro.

Assim como outras classes, a *roscommands* possui uma função principal que é responsável por dividir o comando e enviar o nome da função que deve ser chamada.

Algoritmo 21: Trecho do código que indica qual o comando dado e devolve o nome da função correspondente.

```
1 commandSplit = comando.split(“ ”)
2 if len(commandSplit) == 1:
3   data = {‘commandRos’:‘roscommands’, ‘function’:‘roscommands’, ‘action’:‘send’,
    ‘commands’:commandSplit[0]}
4   brew.publish(“Publisher”, data)
5   rospy.logwarn(“Command sent = ”+command)
6 elif (comandoSplit[1] == “set_robot”):
7   data = {‘commandRos’:‘roscommands’, ‘function’:‘set_robot’, ‘action’:‘send’,
    ‘commands’:comandoSplit[2]}
8   brew.publish(“Publisher”, data)
9   rospy.logwarn(“command sent = ”+command)
```

O algoritmo 21 é diferente dos demais. Após dividir o comando pelos espaços vazios, ele verifica se o resultado é somente um comando ou vários. No primeiro caso, o código sabe que foi enviado um comando simples de movimentação do robô. Já para o segundo caso, o código sabe que o que foi enviado foi o nome do robô que será escolhido para realizar os movimentos. Após essa decisão, o comando ou o nome do robô é armazenado na variável *commands*, encapsulado em um JSON e enviado para o *SpaceBrew* para que o *Master* remoto os receba e realize as operações necessárias.

A função *set_robot* apenas armazena o nome robô que deverá ser utilizado, ele é mantido como uma variável global para poder ser utilizado em qualquer local do código. A função *roscommands*, até o momento recebe a direção de movimentação do robô e altera sua velocidade na direção desejada, logo após retorna a velocidade para zero a fim de evitar colisões.

Outras funções podem ser implementadas futuramente porém as funções mais utilizadas, bem como os comandos para movimentar o robô, foram inseridos na aplicação. O capítulo 4 mostra alguns testes realizados com a aplicação e os resultados dos mesmos.

3.6 Pacote ROS

Como explicado anteriormente, todo o pacote foi baseado em um único serviço do ROS: o */send_data*. O pacote funciona como um *proxy* que “converte” solicitações locais em remotas, isso significa que permite que o usuário utilize o ROS de forma semelhante a qual ele utilizaria normalmente, mas o comando é enviado para o *Master* remoto.

A utilização do pacote é simples, basta iniciar seu executável em Python através do comando *rosrun* (rosrun cloud_ros main_cloud_ros) isso é necessário para criar o cliente *SpaceBrew* e o nó responsável pela criação do serviço */send_data*. Quando o usuário

insere um comando, é necessário enviá-lo através do serviço */send_data*, sendo assim, ao se utilizar o comando *rostopic list* para recuperar todos os tópicos, o usuário deve digitar */send_data "rostopic list"*. Desse modo o serviço é chamado, o comando que se encontra entre aspas é enviado através do *SpaceBrew* e a resposta é devolvida ao usuário.

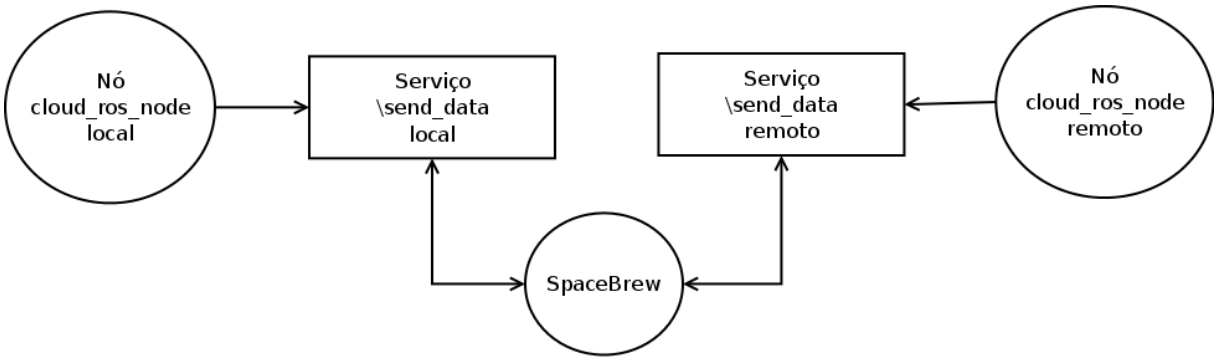

Figura 3.6.1 – Fluxo de comunicação entre o *SpaceBrew* e o ROS.

A Figura 3.6.1 mostra como ocorre a comunicação do ROS com o *SpaceBrew*. É necessário que ambos os *Masters* iniciem o serviço, pois dessa forma são criados clientes que realizam a comunicação. Vale ressaltar que é possível enviar comandos e receber respostas bidirecionalmente.

A maioria dos comandos deve ser inserida como se o usuário estivesse os usando localmente, como acontece com o *rostopic list* e *rosrun*. Porém outros comandos, como o *rostopic echo*, além de ser enviado o nome do tópico que se deseja imprimir os dados, também deve ser enviada a frequência de publicação (em segundos) visto que, por uma limitação do *SpaceBrew*, não é possível que ocorra o tráfego de uma grande quantidade de dados repetidamente. Todo o código para o pacote pode ser encontrado no GitHub (PEREIRA, 2017a), onde o usuário pode clonar o repositório e realizar suas alterações. Além disso todo o pacote é descrito no site do Wiki ROS e pode ser visto em (PEREIRA, 2017b).

3.7 Considerações Finais

Esse capítulo descreveu como foi criado o ROSRemote, suas principais funcionalidades e como utilizá-lo. O próximo capítulo descreve os testes realizados com o ROSRemote e outras ferramentas similares e expõe os resultados obtidos a partir dos testes.

4 Resultados

Nesse capítulo serão descritos todos os testes realizados, suas configurações e máquinas utilizadas, bem como os resultados dos mesmos. Para tanto, foram medidos tempos e velocidades da internet utilizando-se o servidor disponibilizado pelo *SpaceBrew* e o servidor criado com o auxílio de um computador dentro da rede da Universidade Federal de Itajubá. A fim de facilitar a comparação foram criados gráficos com todos os tempos medidos e o tempo médio para indicar o quão viável o ROSRemote é.

Ao fim serão realizados testes utilizando SSH, VPN, Rapyuta e ROSLink a fim realizar comparações dessas ferramentas com o ROSRemote, indicando as vantagens e desvantagens de cada uma delas em relação ao *framework* descrito neste trabalho.

4.1 Planejamento dos Experimentos

Para garantir a viabilidade do trabalho, alguns testes foram criados e o tempo até o usuário receber a resposta foi medido. Todos os testes foram feitos em ambientes controlados e em nenhum deles foi possível inserir interferências externas. No entanto, todos os testes foram realizados sob as mesmas condições, garantindo a credibilidade necessária.

Os teste utilizando SSH e VPN possuem as mesmas disposições de componentes do que os testes com o ROSRemote, a única diferença é que no caso do *framework* é utilizado o servidor do *SpaceBrew*, que adiciona mais um elemento na troca de informações. A disposição dos elementos no teste com o SSH é mostrada na Figura 4.1.1. Vale ressaltar que a disposição dos testes com VPN é exatamente a mesma da Figura 4.1.1, porém ao invés de ser criado um túnel SSH, é criado um túnel VPN.

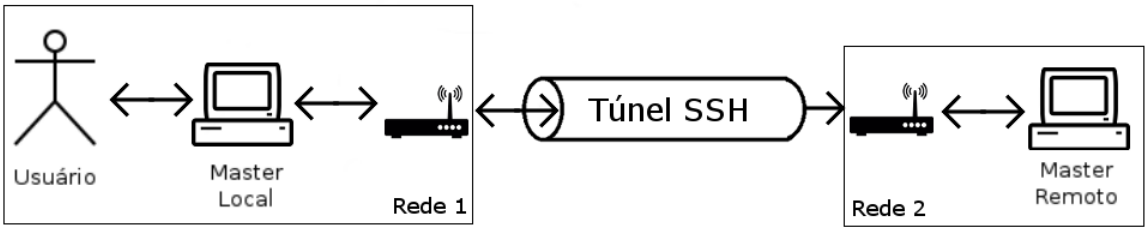

Figura 4.1.1 – Comunicação entre dois computadores através do SSH.

Para a medição dos tempos do SSH e da VPN foi utilizada a ferramenta *WireShark* (COMBS *et al.*, 2008) para Ubuntu, com ela foi possível descobrir o tempo inicial e final do envio e recebimento dos dados e a partir desses tempos, realizar uma subtração simples

para calcular o tempo total. Todos os tempos foram captados a partir do momento que o usuário pressiona a tecla responsável por enviar o comando até o momento em que os dados são impressos. Como o SSH e a VPN enviam um pacote (Acknowledgement ou ACK) para verificar se a conexão ainda está funcionando corretamente, a partir desse momento os dados no *WireShark* foram capturados, após isso o SSH e a VPN enviam os dados e, ao receberem a resposta, enviam um segundo ACK apenas para confirmar o recebimento. O tempo foi medido a partir do primeiro ACK até os dados serem recebidos antes do segundo ACK.

Enquanto para o SSH e a VPN foi necessário utilizar um software para medir o tempo de tráfego de dados, para o ROSRemote a solução foi mais simples, dentro do próprio código fonte da aplicação foi inserido um trecho que captura o tempo inicial e final da comunicação e realiza a subtração automaticamente, mostrando o tempo total de trânsito dos dados. O tempo inicial foi medido a partir do momento em que o usuário aperta a tecla ”Enter”para enviar o comando e o tempo final foi medido no momento em que a resposta é impressa no terminal para o usuário.

Para coletar os dados foram realizados uma sequência de 100 testes para cada uma das ferramentas utilizadas na comparação e o tempo de resposta foi medido. A velocidade da internet e o “ping” também foram medidos. Todos os testes foram realizados no laboratório de robótica da Universidade Federal de Itajubá com três computadores e dois roteadores que possuem a função de NAT, para traduzir IPs privados em públicos e vice-versa.

Além de medir os tempos e velocidades, também foram avaliados alguns problemas que ocorreram durante os testes, algumas ferramentas não funcionaram como esperado e tudo é descrito na seção de resultados, juntamente com a avaliação de qual ferramenta foi mais eficiente na comunicação com o *Master* e robô remotos.

Os testes realizados para o ROSRemote foram diversos. Primeiramente foi utilizado apenas dois computadores e o servidor gratuito do *SpaceBrew*. Posteriormente foi criado um servidor local e foram realizados testes em mais de dois computadores simultaneamente, para verificar se o atraso no envio dos dados seria grande.

Vale ressaltar que os testes foram feitos todos com o envio e recebimento de comandos e dados simples, portanto as configurações das máquinas utilizadas nos testes não influenciam o resultado final de uma forma perceptível. Também é importante dizer que o ROS funciona em Linux e macOS, porém o laboratório de robótica da Unifei possui computadores somente com com Linux instalado. Portanto, todos os testes foram realizados com a utilização desse sistema operacional.

No primeiro teste foram utilizados dois computadores se comunicando através do *SpaceBrew*. Para o funcionamento da aplicação, essa comunicação deve ser bidirecional, ou seja, os dois computadores precisam enviar e receber informações. Desse modo é possível enviar comandos e receber respostas de qualquer um dos dois computadores.

A arquitetura do primeiro teste é mostrada na figura 4.1.2. O envio de comandos

ocorre apenas pelo usuário através do *Master* local e a recepção e envio da resposta, somente pelo *Master* remoto. O usuário é capaz de visualizar os dados da mesma forma que ocorreria se tivesse sido utilizado o ROS somente de forma local.

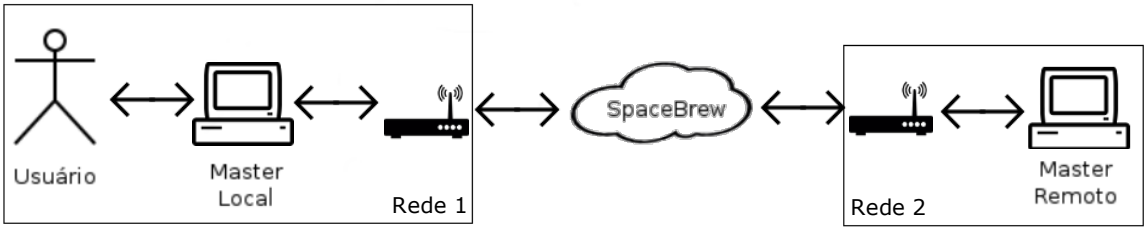

Figura 4.1.2 – Comunicação entre dois computadores através do *SpaceBrew*.

Nesse modelo, o usuário insere os comandos necessários no *Master* local, utilizando o serviço */send_data* e automaticamente, o ROSRemote envia os dados para o *SpaceBrew* que possui o endereço do *Master* remoto e os reenvia esse comando. Ao receber os dados, o *Master* remoto realiza as operações necessárias e retorna a resposta pelo mesmo caminho pelo qual recebeu o comando. Desse modo o usuário visualiza em sua máquina a resposta recebida de outra máquina como se ela houvesse sido requerida localmente.

A arquitetura do segundo teste foi muito semelhante a do primeiro, diferenciando apenas pelo fato de que um robô foi utilizado pelo *Master* remoto. Nesse caso, esse *Master* se comunica com o robô, repassando os comandos que foram enviados pelo usuário, como mostrado na figura 4.1.3.

Figura 4.1.3 – Comunicação entre dois computadores através do *SpaceBrew* e um robô se comunicando com o *master* remoto.

Esse modelo permite que o usuário comande o robô que está em outra rede diferente da sua. Para os testes foram utilizados comandos tele-operacionais de movimentação do robô, além de realizar a leitura de alguns dados publicados por ele.

Outro teste realizado envolveu mais de dois *Masters* simultâneos, todos conectados em sub-redes iguais, porém com diferentes endereços de IP públicos. Vale lembrar que devido a uma limitação do próprio *SpaceBrew*, não é possível realizar a comunicação entre dois computadores que compartilham o mesmo IP público. Por exemplo, se dois

computadores estão conectados em uma mesma sub-rede utilizando o mesmo roteador, as saídas de ambos geralmente será através do mesmo IP público, portanto, não é possível realizar a conexão entre esses dois *Masters* utilizando o ROSRemote.

Como mostrado na figura 4.1.4, o *Master* local se conecta ao mesmo tempo aos dois *Masters* e envia dados aos dois também de forma simultânea. Desse modo é possível verificar se enviar um comando para mais de um local pode influenciar na velocidade do recebimento das informações.

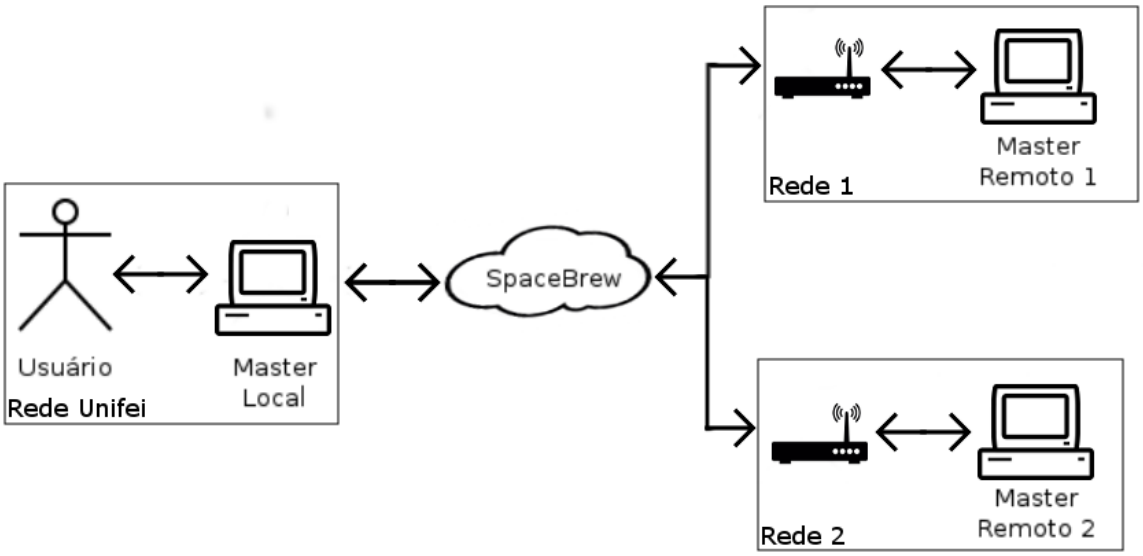

Figura 4.1.4 – Comunicação entre três computadores através do *SpaceBrew*.

Esse teste é importante para o caso de se desejar recuperar informações de mais de um local ao mesmo tempo, pois o *SpaceBrew* permite conectar vários clientes entre si e isso foi um grande auxílio para que esse teste pudesse ser realizado.

Além dos já citados, também foi realizado mais um teste que se assemelha muito ao da Figura 4.1.4, diferenciando-se pelo fato de um robô se comunicar com um dos dois *Masters* remotos. A ideia inicial do teste era conectar um robô a cada *Master*, mas devido a limitação do próprio *SpaceBrew* de não permitir que dois clientes em uma mesma rede se comuniquem através dele, um dos dois computadores não pode ser conectado à rede local, onde se localizam os robôs. Devido a esse fato não foi possível realizar essa comunicação *Master*/robô e a Figura 4.1.5 mostra como foi a configuração final do teste com três computadores e um robô.

Para fins de comparação com o servidor do *SpaceBrew*, foi criado um servidor local que recebeu as conexões dos clientes do *SpaceBrew* e as tratou da mesma forma que o servidor remoto. Dessa forma foi possível comparar se um servidor local poderia tratar os dados recebidos e enviar as respostas de forma mais eficiente que o servidor localizado na nuvem. Esse teste serviu para comprovar se a velocidade do tráfego dos dados aumentaria ou não com a utilização de um servidor dedicado.

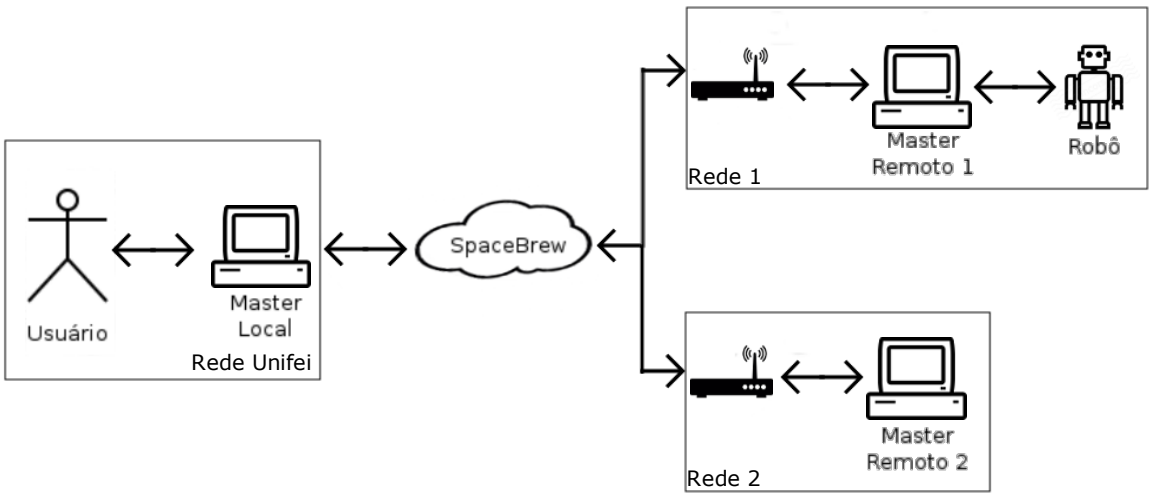

Figura 4.1.5 – Comunicação entre três computadores através do *SpaceBrew* e um robô se comunicando com um *Master* remoto.

Todos os testes foram realizados utilizando-se a rede da Universidade Federal de Itajubá, um computador foi conectado ao roteador do Laboratório de Robótica, outro foi conectado à um segundo roteador que possui outro endereço IP público e o terceiro conectado à rede sem fio da Unifei, utilizando outro IP público. Além disso, o ponto de acesso da rede da Unifei se encontrava longe dos computadores, resultando em um sinal fraco e, consequentemente, em uma velocidade mais baixa no tráfego dos dados.

4.2 Resultados do ROSRemote

O ROSRemote surgiu a partir de uma lacuna que vem tentando ser preenchida por vários pesquisadores, a utilização do ROS em uma rede que não seja local. Como toda aplicação, o ROSRemote possui vantagens e desvantagens. Entre essas desvantagens encontra-se uma das mais preocupantes que é a impossibilidade de se iniciar a aplicação no computador remoto por meio da internet. Isso significa que para o usuário poder utilizar o ROSRemote, é necessário que ele ou outra pessoa execute o arquivo principal do pacote no *Master* remoto a fim de que um cliente *SpaceBrew* seja criado e a conexão possa ser estabelecida.

Outra desvantagem é que, caso o usuário necessite conectar uma grande quantidade de *Masters* ao mesmo tempo, é necessário realizar essas ligações manualmente, selecionando o publicador e o assinante. Dessa forma, supondo que o usuário deseja conectar um número N de robôs em *multicast* (termo utilizado para indicar que todos recebem e enviam comandos de e para todos), será necessário conectar (N-1) publicadores a (N-1) assinantes, totalizando uma quantidade de $(N\text{-}1)^2$ cliques que o usuário deverá realizar na interface

Web Admin Tool para conectar todos os dispositivos. O *SpaceBrew* mantém salvas todas as conexões realizadas, portanto caso seja necessário desconectar e reconectar todos os clientes, as conexões criadas anteriormente são restauradas sem a intervenção do usuário. Esse recurso oferece auxílio em um segundo momento, porém não evita que seja necessário realizar todas as conexões ao menos uma vez.

O *SpaceBrew* é uma ferramenta para se trabalhar com dispositivos que se encontram em redes diferentes, portanto não pode ser utilizado por dois dispositivos que utilizam a mesma rede. Todo o processamento do *SpaceBrew* é baseado no IP público de um usuário e caso dois clientes com mesmo endereço IP público tentem se conectar, o segundo sobrescreve o primeiro. Portanto dois dispositivos conectados através de um mesmo roteador não podem se comunicar pelo *SpaceBrew* e consequentemente, nem pelo ROSRemote, o que acarreta em uma desvantagem. Caso o usuário necessite conectar vários *Masters* utilizando uma mesma rede, ele deverá buscar outras ferramentas que possam realizar esse trabalho.

Além das desvantagens já citadas, também existe o problema com o servidor *SpaceBrew* ser um ponto de falha. Como todas as conexões obrigatoriamente precisam passar por ele, se ocorrer alguma falha com esse servidor todas as aplicações são desconectadas.

Por outro lado, trabalhar com o ROSRemote também traz vantagens ao usuário. Para a aplicação funcionar, é necessário que sejam conectados dois ou mais *Masters*, com isso o usuário pode interligar várias plataformas distintas. É possível fazer com que uma aplicação que esteja sendo executada no sistema Ubuntu se comunique com outra que esteja sendo utilizada no macOS, por exemplo. Esse mesmo fato faz com que a aplicação também consiga se comunicar utilizando distribuições diferentes de Ubuntu ou do próprio ROS. Um *Master* que esteja funcionando em um Ubuntu 14.04 com ROS Groovy pode se comunicar com outro sendo utilizado em um Ubuntu 12.04 que utilize o ROS Hydro. Além disso, também é possível realizar a comunicação de dispositivos diferentes, como robôs, smartfones e notebooks, contando que todos possuam capacidade de trabalhar com ROS. Por último, também é possível conectar aplicações diferentes, cada usuário pode criar sua própria aplicação e recuperar dados de outras, mesmo que ambas sejam diferentes.

Ainda existe outra vantagem de se trabalhar com mais de um *Master* ao mesmo tempo. Caso ocorra a falha de um deles, o sistema como um todo ainda continua em funcionamento. Esse é um ponto bastante importante, pois evita que o usuário perca todos os seus dados. Se uma falha ocorrer, o usuário perderá os dados apenas daquele *Master* onde a falha ocorreu.

Ainda há a vantagem com relação a segurança de alguns sistemas, pois algumas ferramentas podem ser bloqueadas por *Firewalls*. Isso em alguns casos, impossibilita o uso de outras ferramentas, mas não atrapalha o funcionamento do *SpaceBrew*, pois com ele o usuário pode utilizar um servidor localizado na nuvem evitando que seja necessário realizar o redirecionamento de portas e criar regras de acesso. Essas ações podem ser

bloqueadas ao usuário comum em alguns locais, o que acaba inviabilizando a utilização de algumas ferramentas para envio de dados remotos.

Com a utilização do *SpaceBrew*, é possível que o usuário crie e remova conexões em tempo de execução. Isso é vantajoso pois permite que o usuário possa cessar o recebimento de informações de um determinado *Master* e retomar quando necessário. É possível realizar todas essas ações sem a necessidade de interromper toda a aplicação para refazer todas as conexões e perder os dados existentes.

Por último, Koubaa *et al.* (2017) descrevem duas vantagens para sua aplicação ROSlink que também justificam a utilização do ROSRemote. A primeira delas é que outras ferramentas, como a VPN, descrita na seção 2.7.1.2, ou o SSH, descrito na seção 2.7.1.1, necessitam realizar o redirecionamento de portas para serem configurados e utilizados. Os autores apontam que no caso do usuário estar utilizando redes 3G/4G essa prática se torna impossível. A segunda vantagem é em relação as aplicações que conectam vários usuários e robôs a um mesmo *Master*, nesse caso, existem dois problemas que não ocorrem no ROSRemote:

- Conflito: vários Tópicos, Nós e Serviços podem ter o mesmo nome, necessitando um estudo profundo sobre o domínio de nomes para evitar que dados errados sejam enviados ao usuário;
- Escalabilidade: essa solução não permite que o sistema possa ser facilmente escalado, gerando problemas muito complexos com o aumento da quantidade de dispositivos conectados, como sobrecarga.

Além das vantagens do ROSRemote, os resultados dos testes provaram que o projeto é viável, principalmente se for utilizado em um servidor local. Em todo caso, é necessário que a conexão possua um nível de qualidade que permita a utilização de aplicações remotas e que consiga trafegar os dados com uma velocidade mais alta (PEREIRA; BASTOS, 2017).

Como uma das principais vantagens do *SpaceBrew* é poder realizar e desfazer conexões em tempo de execução, em todos os testes esse procedimento foi realizado a fim de avaliar se a aplicação apresentaria alguma instabilidade. Essa vantagem do *SpaceBrew* foi confirmada em todos os testes, pois ao desconectar um dispositivo, o recebimendo de informação do *Master* selecoinado era cessada e quando a conexão era refeita, o recebimento dos dados era retomado normalmente. A aplicação não apresentou problemas ao receber dados remotos nem quando o usuário estava realizando o controle remoto do robô.

O primeiro teste foi realizado em dois locais diferentes para verificar se a velocidade na internet poderia interferir na aplicação. No primeiro local, a velocidade da internet foi medida, sendo constatado que para download era de 8,57 Mbps e para upload de 0,48 Mbps, ou seja, a velocidade para envio dos dados não é alta e portanto, o tempo gasto para o tráfego dos dados foi relativamente alto.

Em cerca de 10 testes realizados com essa velocidade, o tempo de resposta mais baixo obtido foi 0,0834 segundo, mas houveram casos em que o tráfego dos dados durou cerca de 0,8 segundo. Devido a velocidade de upload baixa, o tempo de resposta foi um pouco alto, tendo como média cerca de 0,595 segundo. Desse teste é possível concluir que, em uma rede de velocidade baixa, o projeto pode ser utilizado para recuperar informações que não necessitem urgência. Já no caso de instruções que possam danificar o robô não é indicado que o usuário utilize o ROSRemote juntamente com uma internet que possua essa velocidade.

O mesmo teste foi realizado na rede da Unifei que possui uma internet mais rápida, cerca de 50 Mbps para download e 35 Mbps para upload, como descrito na seção 4.1, ainda utilizando o servidor gratuito do *SpaceBrew*. No laboratório de robótica foram realizados 100 repetições e a média de tempo entre todas foi de 0,1472 segundo. O tempo de tráfego dos dados pode ser visto no gráfico da Figura 4.2.1, que mostra que o menor tempo registrado foi de 0,09338 segundo e o maior de 0,69591 segundo.

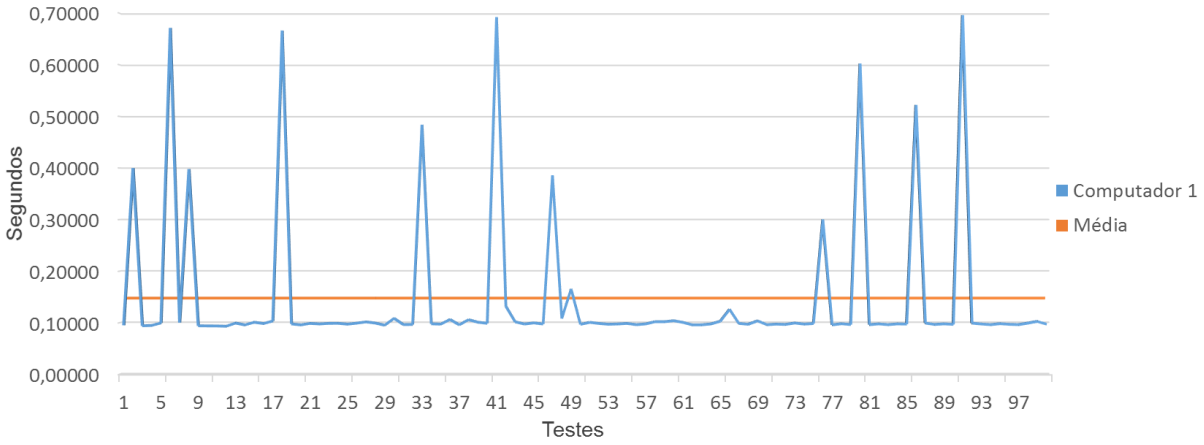

Figura 4.2.1 – Tempo de tráfego dos dados utilizando um computador e o servidor do *SpaceBrew*.

Para esse teste também foi medido o "ping", que é o tempo necessário para uma informação chegar ao destino e voltar ao remetente. Com ele foi possível notar que o tempo necessário para os dados trafegarem até o servidor grátis do *SpaceBrew* é alto, como mostrado no gráfico da Figura 4.2.2, no qual a média de tempo foi de aproximadamente 300ms. Com esse resultado é possível mostrar que a velocidade da internet afeta a velocidade final do ROSRemote.

Para uma aplicação que possa oferecer risco de danos ao robô ou a outros envolvidos, utilizar o servidor gratuito do *SpaecBrew* não é a melhor solução. No entanto para aplicações em que seja somente necessário recuperar os dados sem que seja obrigatório o envio de resposta em um curto período de tempo, o ROSRemote pode ser utilizado e atenderá às necessidades do usuário.

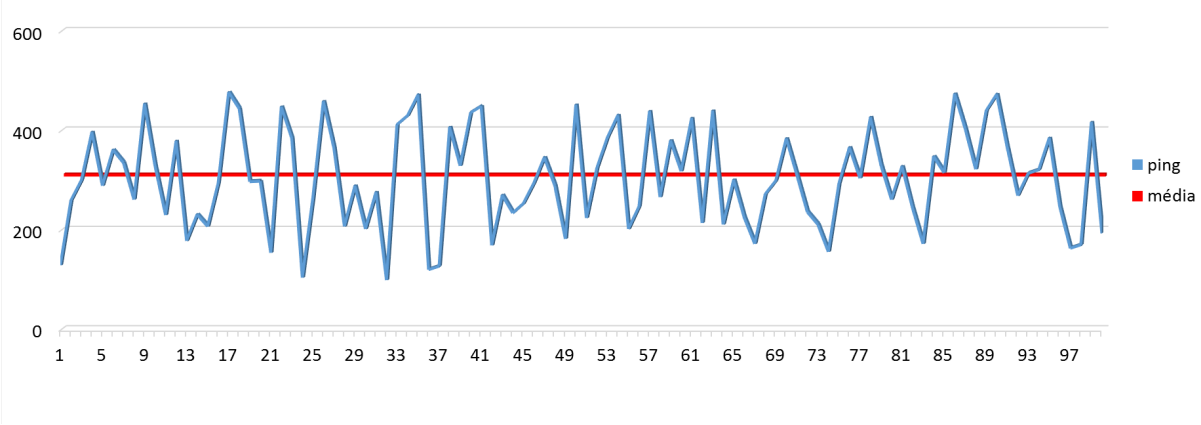

Figura 4.2.2 – Tempo de tráfego dos dados medidos por meio do utilitário “Ping”.

No segundo teste os resultados também foram muito satisfatórios, foram enviados comandos tele operacionais simples ao robô e a resposta foi praticamente instantânea, assim como no primeiro teste, o tempo foi de menos de 1 segundo, cerca de 0,4 segundo para o robô se movimentar de acordo com o comando enviado. Com esse resultado é possível perceber que a utilização do ROSRemote para comandar robôs a distância pode ser viável, contando que não haja obstáculos que possam causar danos sérios à ele e que devam ser avistados e desviados com antecedência.

Mesmo quando vários comandos seguidos foram enviados ao robô, ele respondeu a todos com um pequeno atraso devido à velocidade da internet. Vale ressaltar que o comando é enviado para o servidor do *SpaceBrew* e posteriormente transmitido ao robô, ocasionando um pequeno aumento no tempo total.

O terceiro teste, realizado com três máquinas, gerou o resultado esperado. Todos os computadores obtiveram um tempo de resposta baixo e as informações chegaram de forma correta, resultado que já era previsto já que o *SpaceBrew* utiliza o protocolo TCP e este garante a entrega de forma correta.

No entanto, os tempos registrados não se diferenciaram muito do primeiro teste, a média foi apenas 0,02 segundo acima do teste com apenas um computador, cerca de 0,16 segundo. Esse teste obteve tempos mínimos de 0,09854 segundo para um computador e 0,09200 para outro, bem como tempos máximos de 0,75853 e 0,69354, respectivamente. O gráfico da Figura 4.2.3 mostra como foi o desempenho do tráfego de dados.

Com esse resultado é possível verificar que mesmo em uma situação em que haja vários computadores interligados, é possível que todos enviem e recebam comandos simultâneos sem alterar a integridade dos dados. Isso permite que vários *Masters* possam ser conectados ao mesmo tempo contando que seus IPs públicos sejam diferentes.

O último teste, que envolveu três máquinas e um robô, foi o mais complexo e dele também foi obtido o resultado esperado. O mesmo comando é enviado para todos os clientes do *SpaceBrew* simultaneamente e se o *Master* possuísse um robô conectado, esse deveria se movimentar. Com esse teste foi constatado que além da movimentação ocorrer

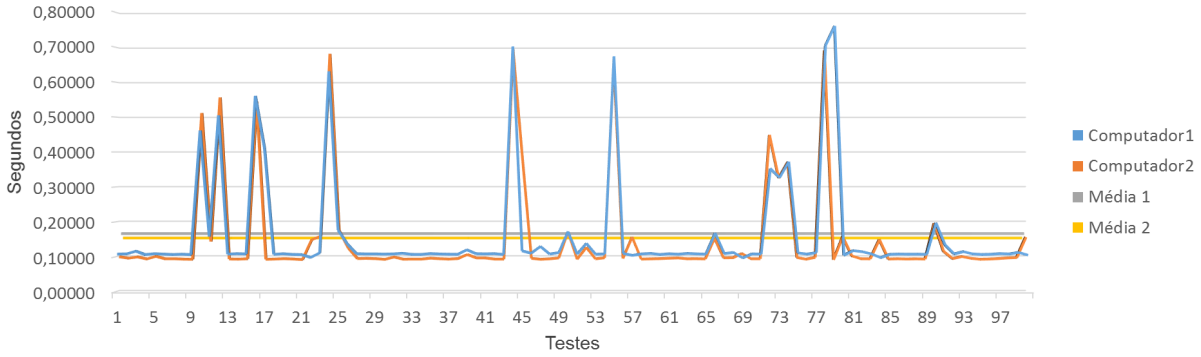

Figura 4.2.3 – Tempo de tráfego dos dados utilizando dois computadores e o servidor do *SpaceBrew*.

corretamente, o computador que não estava conectado ao robô devolveu ao usuário uma mensagem indicando que não era possível realizar a movimentação devido ao fato de não haver nenhum robô conectado a ele.

Dessa forma, os testes realizados utilizando o servidor disponibilizado gratuitamente pelo *SpaceBrew* permitiram concluir que o ROSRemote é viável e pode ser utilizado em aplicações nas quais não haja risco de danos ao robô ou pessoas que possam estar perto dele. Se a conexão com a internet for de boa qualidade, os atrasos poderão ser imperceptíveis ao usuário.

Em todos os quatro testes, a variação de tempo na transferência dos dados foi relativamente alta e com isso é possível constatar que a oscilação da internet pode influenciar bastante no tráfego dos dados. É possível inferir que a variação do tempo de tráfego dos dados foi alta devido ao fato de que o desvio padrão obtido com os dados do teste também foi alto. A Tabela 4.2.1 mostra a média e o desvio padrão em cada um deles. O gráfico da Figura 4.2.1 demonstra que algumas medidas que chegaram perto de 0,7 segundo foram casos isolados. Portanto, utilizando o desvio padrão, é possível que essas medidas possam ser descartadas no resultado final. Em todo caso, a média de tempo não foi alta, portanto o controle de robôs com o ROSRemote nestes testes pode ser realizada.

Tabela 4.2.1 – Média e Desvio Padrão do teste com o ROSRemote utilizando o servidor grátis do *SpaceBrew* e 100 repetições.

	Média	Desvio Padrão	Intervalo de Confiança $\alpha = 99\%$
Teste 1	Máquina 1 0,1472	Máquina 1 0,1426	Máquina 1 0,03674
Teste 2	Máquina 1 0,1656 Máquina 2 0,1519	Máquina 1 0,1480 Máquina 2 0,1431	Máquina 1 0,03812 Máquina 2 0,0368

Os resultados utilizando um servidor local foram melhores do que o esperado. Os tempos medidos foram menores dos que os obtidos com o servidor gratuito do *SpaceBrew* e durante alguns picos de velocidade da internet, foi possível obter tempos muito abaixo dos obtidos nos outros testes. O menor tempo registrado foi 0,006 segundo e a média também foi baixa, cerca de 7 milissegundos.

No teste com apenas um computador conectado ao servidor criado dentro da rede da Unifei, foi possível identificar uma média de 0,0068 segundo, abaixo da média utilizando-se o servidor gratuito disponível pelos desenvolvedores do *SpaceBrew*, de 0,14 segundo. Nesse teste, também foi medido o tempo de tráfego de dados utilizando o "ping" e o tempo médio foi de menos de 5ms. Ou seja, um servidor local com tempo de "ping" baixo resultou em um tempo baixo para a aplicação do ROSRemote. Isso permite concluir que os dados trafegados pelo *SpaceBrew* são enviados em uma velocidade alta e com pouca latência. Além disso, o tempo mínimo foi de 0,006 segundo e o tempo máximo de 0,14 segundo. Esses resultados podem ser vistos no gráfico da Figura 4.2.4.

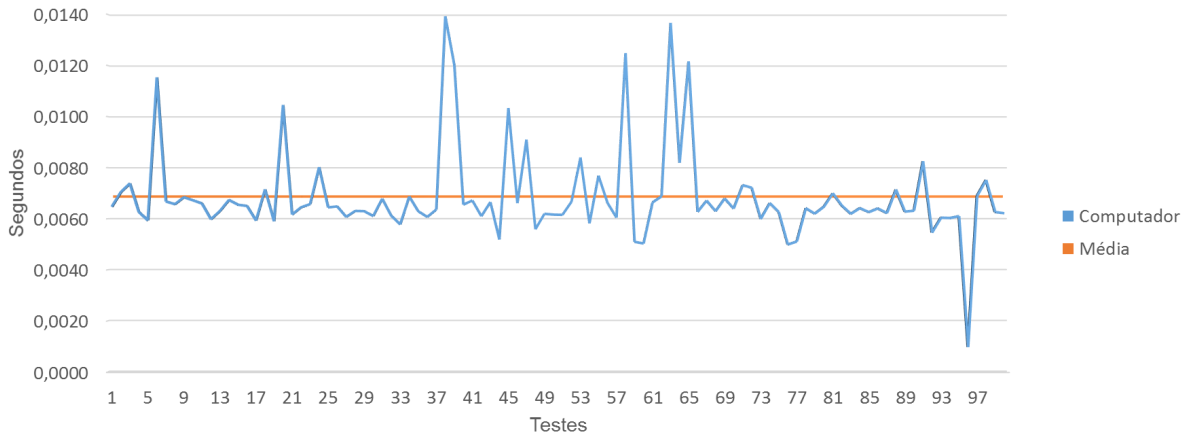

Figura 4.2.4 – Tempo de tráfego dos dados utilizando um computador e o servidor criado na Unifei.

No último teste realizado, foram utilizados três computadores conectados em uma rede, porém em diferentes sub-redes, onde também se localizava o servidor. Os dados foram muito semelhantes com os do teste anterior, houveram picos de velocidade nos quais o tempo foi extremamente baixo, cerca de 0,006 segundo, mas em alguns momentos, o tempo de resposta chegou a 0,389 segundo, como pode ser visto no gráfico da Figura 4.2.5.

É possível perceber que no servidor local, que possuía um tempo médio de tráfego de dados de cerca de 4ms (medido através do "ping"), a média de tempo foi melhor. A oscilação da internet não foi alta, como pode ser visto na Tabela 4.2.2. Utilizando o desvio padrão neste teste, não é necessário descartar nenhum resultado mostrado no gráfico da Figura 4.2.5. A média de tráfego de dados obteve um tempo aceitável para uma aplicação de controle de robôs e isso permite assegurar que o ROSRemote é viável, principalmente quando o servidor possui uma taxa de transferência alta e latência baixa, como é o caso desse criado no laboratório de robótica.

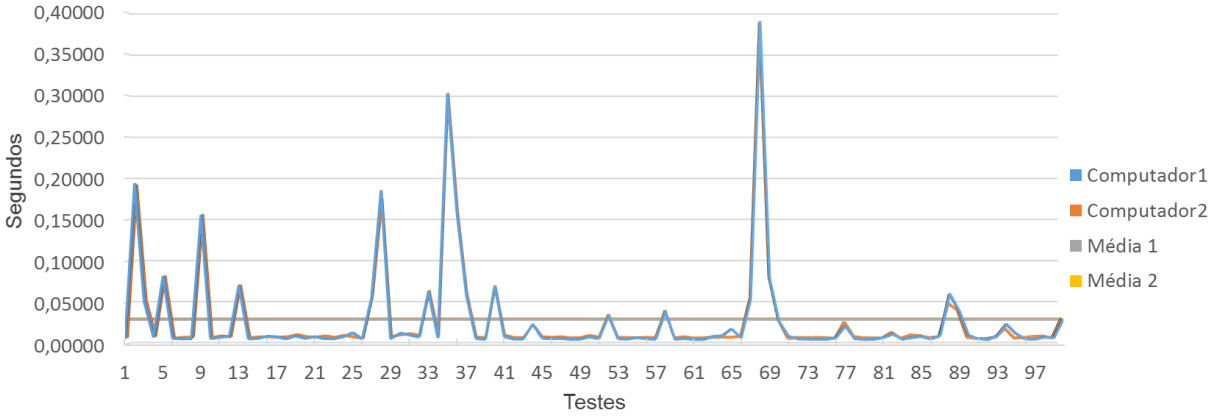

Figura 4.2.5 – Tempo de tráfego dos dados utilizando dois computadores e o servidor criado na Unifei e 100 repetições.

Tabela 4.2.2 – Média e Desvio Padrão do teste com o ROSRemote utilizando o servidor criado no laboratório de robótica da Unifei.

	Média	Desvio Padrão	Intervalo de Confiança $\alpha = 99\%$
Teste 1	Máquina 1 0,0068	Máquina 1 0,0017	Máquina 1 0,00046
Teste 2	Máquina 1 0,0281 Máquina 2 0,0292	Máquina 1 0,0583 Máquina 2 0,0579	Máquina 1 0,01501 Máquina 2 0,01492

Com esses dados, a conclusão é que o processamento do *SpaceBrew* é rápido, visto que tanto no servidor remoto (com média de tempo de tráfego de 300ms) quanto no servidor local (com média de tempo de tráfego de 4ms) o tempo de resposta do ROSRemote não foi muito alto. Assim foi encontrado o primeiro resultado importante para o trabalho. Para o controle de robôs, o *SpaceBrew* pode ser indicado e possivelmente, seja uma ferramenta que possa ser utilizada para quaisquer aplicações na web. Ele é simples de se utilizar, muito poderoso e permite criar e desfazer conexões em tempo de execução.

A segunda conclusão é com relação ao ROSRemote, ele é uma ferramenta que cumpre o que promete, porém ainda possui alguns problemas que podem ser resolvidos. Por exemplo, não é possível utilizar o comando *rostopic echo* em mais de um Tópico ao mesmo tempo, pois como não é possível utilizar mais de um terminal para a aplicação, é necessário cessar o recebimento de informações de um Tópico a fim de iniciar o recebimento das informações sobre outro. Outra desvantagem do ROSRemote é que é preciso que o ROS e o pacote ROSRemote estejam sendo executados no computador remoto e não há como inicializar esses processos a distância através do ROSRemote.

Uma das principais vantagens é a facilidade de realizar e desfazer conexões em tempo

de execução da aplicação, o que pode auxiliar bastante o usuário, visto que pode ser necessário parar de receber informações de uma das máquinas remotas por um curto período de tempo e voltar a receber esses dados normalmente. Além disso, é possível dar nomes para as aplicações do ROSRemote, eles aparecerão no *Web Admin Page* do *SpaceBrew*, permitindo ao usuário descobrir de qual servidor se originou a informação, bem como auxiliando-o na realização das conexões com os destinos desejados.

Por último, é possível utilizar o ROSRemote em qualquer lugar sem a necessidade de realizar configurações que, em alguns casos, são impossíveis. Também é possível utilizá-lo em qualquer plataforma, em qualquer dispositivo e para qualquer aplicação do ROS.

4.3 Resultados do SSH

Os mesmos testes realizados utilizando-se o ROSRemote foram feitos com o SSH e os resultados também foram bons.. A velocidade de tráfego de dados pelo SSH é bastante alta, mas ele possui algumas desvantagens como a necessidade de se realizar uma configuração prévia que não é complicada, mas necessita de acesso a roteadores e isso pode ser um ponto muito importante pois nem sempre o usuário tem acesso a esses recursos. Além disso, caso o usuário necessite realizar conexões com um número N de servidores, é necessário que essa configuração seja realizada em todos eles, bem como o redirecionamento de portas em todas as redes.

O primeiro teste utilizou apenas dois computadores enviando dados entre si e o tráfego foi bastante rápido, praticamente simultâneo. Isso se deve ao fato de que, no SSH os dois computadores são conectados diretamente, eliminando a necessidade de um servidor para realizar essa conexão. Com esse teste foi possível verificar uma desvantagem do SSH em relação ao ROSRemote, para realizar a conexão é necessário conhecer o IP público do servidor, bem como a porta liberada para a conexão. É possível configurar um DDNS (Dinamic DNS) e dar um nome ao servidor, porém da mesma forma, o usuário deve saber o nome de todos os servidores com os quais deseja se conectar.

Mesmo o SSH sendo uma ferramenta bastante rápida, esse protocolo necessita criptografar os dados antes de enviá-los e por isso, acaba sendo mais lento do que uma ferramenta que envia os dados sem criptografia, como o ROSRemote. Esse ponto é vantajoso para usuários que necessitam de segurança no tráfego dos dados mas caso contrário, isso afeta o tempo de resposta. Nos testes realizados com o SSH foi constatado que o tempo médio de tráfego dos dados, que pode ser visto no gráfico da Figura 4.3.1, foi de 15 vezes maior do que o tempo do ROSRemote que obteve cerca de 0,0068 contra um tempo médio de 0,1054 segundo do SSH.

Assim como o ROSRemote, a variação na velocidade do tráfego dos dados foi alta, e o desvio padrão foi maior do que o teste do ROSRemote utilizando o servidor gratuito do *SpaceBrew*. Enquanto o segundo obteve um desvio padrão de 0,0017, o teste com o

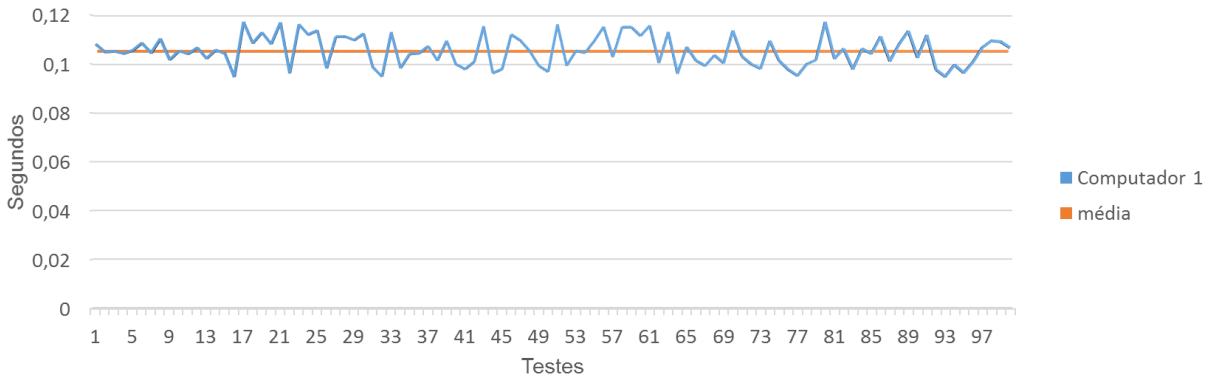

Figura 4.3.1 – Tempo de tráfego dos dados utilizando dois computadores e o SSH.

SSH obteve um desvio padrão de 0,0062. Com isso é possível concluir que o SSH possui uma latência maior do que o ROSRemote. Porém ambas as ferramentas obtiveram um resultado satisfatório para aplicações na web.

O segundo teste também obteve um resultado tão rápido quanto o primeiro, a velocidade no tráfego dos dados foi praticamente simultânea e precisa, visto que o SSH também funciona sobre o protocolo TCP, que evita perda de dados. Porém, durante realização do segundo teste foi constatada outra desvantagem do SSH, não é possível realizar a conexão com dois computadores simultaneamente de forma tão simples. É necessário configurar um servidor SSH em cada um dos computadores com os quais deseja se conectar e então, a conexão deve ser realizada de forma diferente da conexão para um computador. É necessário utilizar ferramentas como o PSSH (Paralel Secure Shell) ou criar *scripts* que contenham os IPs ou nomes dos destinos e iterar sobre eles para realizar a conexão um a um. Quando esse *script* é utilizado no terminal, dá-se a impressão de que a conexão foi feita de forma simultânea porém, internamente, a conexão é realizada separadamente com cada computador remoto.

O terceiro e quartos testes, que envolviam realizar a movimentação de robôs de forma remota, também obtiveram bons resultados em questão de velocidade. Por outro lado ocorreram alguns problemas quando, ao mesmo tempo, foi utilizado o comando *rostopic echo* e o tele operacional do robô. O primeiro comando foi utilizado para imprimir os dados publicados pelo Tópico cmd_vel, que mostra a velocidade do robô. Em um primeiro momento, foi notado que o terminal não mostrava corretamente todas as velocidades, mesmo quando elas já haviam sido alteradas, outras vezes imprimia a velocidade antiga e ainda ocorreram casos em que nada era impresso no terminal para o usuário. Após esse problema, também foi constatado que da mesma forma que a impressão do tópico era paralisada, o robô também não respondia aos comandos tele operacionais.

Após alguns estudos e pesquisas foi constatado que o problema de se abrir dois terminais conectados ao mesmo tempo com SSH se deve a uma limitação da própria ferramenta.

Para realizar uma conexão por meio desse protocolo é preciso se conectar a uma máquina com um usuário, como mostrado na seção 2.7.1.1. No entanto, a cada vez que o usuário se conecta, o SSH cria um novo túnel responsável pelo tráfego dos dados, conforme Figura 4.3.2. É possível criar vários túneis, contando que eles possuam origem ou destino diferentes, ou ainda, que sejam conectados por usuários diferentes. Caso um usuário de uma mesma máquina se conecte duas vezes ao mesmo usuário de uma mesma máquina remota, ao se enviar comandos em concorrência, o SSH pode confundir o túnel pelo qual deve enviar as respostas e elas se perderão, conforme Figura 4.3.3.

Figura 4.3.2 – Forma correta de se utilizar o tunelamento SSH.

Figura 4.3.3 – Forma incorreta de se utilizar o tunelamento SSH.

Por fim, é possível concluir que o SSH é rápido, mas ainda assim obteve um tempo 15 vezes maior do que o tempo do ROSRemote nas mesmas condições de testes. Além disso, o SSH pode apresentar alguns problemas que não ocorrem no ROSRemote, como a paralização do terminal em caso de envio de dados concorrentes. Outra desvantagem do SSH é a necessidade do usuário conhecer todos os endereços aos quais quer se conectar. Por outro lado tudo isso não justifica a não utilização do SSH, apesar de ser necessário realizar uma configuração ao utilizá-lo, após essa primeira configuração, ele é rápido e devido às criptografias, seguro.

4.4 Resultados da VPN

Os mesmos testes realizados utilizando-se o ROSRemote e o SSH foram feitos com a VPN e, entre todas as ferramentas, essa foi a que resultou nos piores tempos. A velocidade de

tráfego de dados em uma VPN pode ser alta, porém devido à criptografia que é inerente ao protocolo, o tempo no tráfego de dados foi mais alto que as outras ferramentas. Além disso, a VPN também possui como desvantagem a necessidade de se realizar uma configuração prévia bastante complexa e também necessita de acesso a roteadores. Portanto, da mesma forma que o SSH, esse é um problema que pode fazer com que alguns usuários não consigam configurar uma rede VPN devido ao fato de não terem acesso aos roteadores.

Configurar uma rede VPN não é uma tarefa muito simples. É necessário na maioria dos casos, utilizar um software capaz de criar um servidor de VPN que possa manter essa conexão, existem softwares gratuitos para esse fim, porém os softwares que possuem maiores recursos, como auxiliar envio de dados em *broadcast*, são pagos. Apesar de também ser necessário configurar um servidor para o *SpaceBrew* (caso o usuário não deseje utilizar o servidor disponibilizado pelos desenvolvedores), sua utilização ainda é mais simples, necessitando apenas configurar o servidor para receber conexões externas com alguns simples comandos no terminal do Ubuntu, sem a necessidade de abrir portas.

O segundo problema é que, caso o usuário necessite enviar e receber informações de vários computadores ao mesmo tempo, ele teria um grande problema ao realizar essa tarefa através de uma VPN. Enquanto o *SpaceBrew* aceita que vários dispositivos se conectem entre si de forma rápida e fácil, configurar uma VPN com mais de dois dispositivos é uma tarefa árdua e ainda necessita de softwares de terceiros.

Outro problema que pode tornar a utilização de VPNs para conectar muitos dispositivos inviável é que são necessários hardwares específicos para esse tipo de conexão como roteadores e switches que forneçam suporte à essa tecnologia. Para a utilização do ROSRemote só é necessário o dispositivo com o ROS e outro com o *SpaceBrew*, caso o usuário opte por criar seu próprio servidor.

Porém, mesmo com todas as desvantagens, ainda foram realizados testes utilizando uma conexão VPN e como já descrito, ela obteve o pior tempo entre todas as ferramentas testadas. Enquanto as médias do ROSRemote e do SSH foram de 0,0068 e 0,1054 segundo, respectivamente, a VPN obteve uma média de 0,1361. Além disso, a VPN foi a ferramenta que gerou maior oscilação no tempo de envio dos dados, com um desvio padrão de 0,007, contra 0,017 do ROSRemote e 0,0062 do SSH. Os tempos de resposta da VPN podem ser vistos no gráfico da Figura 4.4.1.

4.5 Resultados do *Rapyuta*

O framework *Rapyuta* promete ser uma ferramenta para auxiliar no processamento dos dados obtidos pelos robôs, podendo trabalhar diretamente com o ROS e facilitando o trabalho do desenvolvedor, pois permite que os pacotes do ROS sejam utilizados sem modificações. Porém, todas essas vantagens não passaram da teoria, na prática, o *framweork Rapyuta* se mostra bastante complexo logo na instalação.

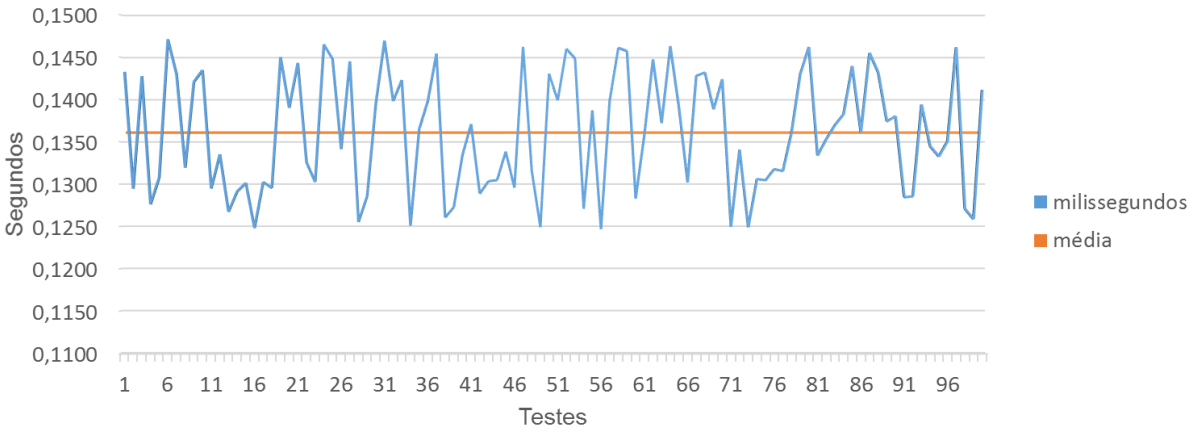

Figura 4.4.1 – Tempo de tráfego dos dados utilizando dois computadores e a VPN.

A princípio, o site não diz que o framework funciona somente em algumas versões do Ubuntu e do ROS, esse detalhe só é notado durante a instalação. Nesse processo é fornecido ao usuário a possibilidade de escolher entre as versões *Precise* (12.04), *Quantal* (12.12) ou *Raring* (13.04) do Ubuntu e entre as versões, *Groovy*, *Fuerte* ou *Hydro* do ROS. Ao tentar instalar essa aplicação na versão 14.04 (Trusty) do Ubuntu, o instalador apontou um erro exatamente devido à versão na qual estava ocorrendo o processo. Isso, em si, é uma desvantagem com relação as outras ferramentas, pois limita o usuário a trabalhar somente com algumas versões suportadas.

Durante a instalação foi encontrado outro problema grave que impossibilitou a utilização do *Rapyuta* que é a dependência *Twisted* (mecanismo dirigido a eventos que deve ser instalada para o correto funcionamento). Essa dependência gera um erro em Python durante a instalação que, para ser solucionada necessita de um *Downgrade* da mesma. Porém ao instalar uma versão mais antiga, um segundo erro é apontado e para solucioná-lo é necessário realizar um *upgrade* dessa mesma dependência, repetindo esse círculo que não permite a instalação da ferramenta.

Como descrito na seção 2.7.2.1, foram encontradas poucas aplicações que utilizam o *Rapyuta* e, metade delas foi feita pelos próprios criadores da ferramenta. Todos os artigos que citam o *Rapyuta* não dizem nada sobre a instalação, apenas citam que existe essa ferramenta sem mencionar maiores detalhes. Isso pode ser um indício de que, apesar da aplicação ser muito útil na teoria, na prática talvez não auxilie muito o usuário, principalmente se ele possuir uma versão mais nova do Ubuntu ou do ROS.

Por fim, não é possível realizar o controle de robôs utilizando essa aplicação, ela somente serve para que o processamento do robô seja feito fora dele, diminuindo seu custo e melhorando o tempo de resposta do mesmo. O *Rapyuta* realiza a movimentação do robô somente de forma automática. O *rapyuta* recebe os dados, realiza o processamento e envia o comando ao robô automaticamente, de acordo com a aplicação desenvolvida para cada caso. Contudo não é possível realizar a movimentação de robôs pelo teclado, como

no ROS ou no ROSRemote. Portanto, apesar de ser uma aplicação de robótica muito similar ao ROSRemote, não pode ser utilizada como comparação para o mesmo. Além disso, ainda não foi possível finalizar a instalação para poder realizar alguns e avaliar a utilidade e a viabilidade da ferramenta.

4.6 Resultados do ROSLink

Da mesma forma que o *Rapyuta*, o ROSLink promete ser uma aplicação que permite ao usuário utilizar vários recursos úteis. Sua instalação é simples e necessita apenas da realização do download do código fonte e posteriormente, a compilação de seu código utilizando a ferramenta *catkin*. Apesar da instalação ser simples, a utilização não é tão atrativa.

Em um primeiro momento é possível perceber que o tutorial para sua utilização é falho e apesar de mostrar vários exemplos da ferramenta funcionando, não descreve um ponto crucial: como realizar a conexão entre os dispositivos.

Além de não ser possível realizar o teste com essa aplicação, assim como com o *Rapyuta*, com ela também não é possível realizar o controle de robôs via teclado, como é possível no ROSRemote. O ROSLink apenas permite que o usuário consiga programar uma aplicação utilizando as estruturas de dados pré criadas pelo aplicativo. A lista de todas essas mensagens pode ser vista em Koubaa (2018).

4.7 Comparação Entre Ferramentas

Todos os testes resultaram em tempos que permitem a utilização do ROS para contorle remoto aos robôs, porém entre todos, o ROSRemote foi o que obteve o melhor tempo, tanto utilizando um servidor criado localmente quanto com o servidor gratuito do *SpaceBrew*. Esse resultado se deve ao fato de que tanto a VPN quanto o SSH criptografam os dados antes do envio e descriptografam após o recebimento e o ROSRemote trabalha com dados crus, sem criptografia. Isso resulta em uma vantagem e uma desvantagem, caso o usuário necessite de segurança para o tráfego dos dados, o ROSRemote pode não ser a melhor opção, já que a VPN e o SSH oferecem mais segurança nesse ponto. Por outro lado, se o usuário não se estiver interessado em segurança e sim em velocidade, o ROSRemote vai lhe atender melhor, pois foi 5 vezes mais rápido do que o SSH e quase 7 vezes mais rápido que a VPN. A Tabela 4.7.1 mostra a diferença de tempo entre os testes utilizando o ROSRemote, o SSH e a VPN. É possível notar que, quando conectados utilizando a mesma configuração, o ROSRemote foi superior as outras ferramentas em questão de tempo.

Na questão da configuração das ferramentas, o ROSRemote também é o mais simples e mais funcional. Para se utilizar o SSH e a VPN o usuário necessita realizar o chamado

Tabela 4.7.1 – Comparação entre as médias das ferramentas utilizadas nos testes.

	Média	Desvio Padrão
ROSRemote	0,0068	0,0017
SSH	0,3300	0,1603
VPN	0,4414	0,1810

redirecionamento de portas no roteador ou switch e isso não é possível no caso do usuário estar fazendo uso de internet 3G/4G ou desconhecer a senha do roteador. Além disso, algumas empresas fornecedoras de internet não permitem que o usuário realize esse tipo de configuração e algumas inserem multas contratuais caso o usuário realize essa configuração com a rede.

O *SpaceBrew* possui uma característica que ao mesmo tempo em que pode ser considerada um vantagem também pode ser considerada desvantagem, que é a forma como as conexões com os *Masters* remotos são realizadas. No SSH e na VPN, é necessário que o usuário saiba todos os nomes ou endereços dos destinos com os quais deseja conectar e realizar as conexões uma a uma, ou criar um *script* que realize esse trabalho. No ROSRemote isso não ocorre pois o usuário consegue ver todos os clientes que foram criados com o nome da aplicação e o endereço. Isso auxilia o usuário pois, dessa forma, ele não precisa saber cada um dos destinos, é possível nomear as aplicações de formas que o ajude a conhecer a origem e então conectar apenas os clientes desejados. Por outro lado, caso sejam necessários conectar vários clientes ao mesmo tempo, o usuário terá um grande trabalho realizando todas as conexões uma a uma. Como descrito, será necessário realizar $2*(N-1)^2$ cliques com o mouse, com N sendo a quantidade de clientes criados, caso o usuário deseje conectar todos os clientes.

Apesar de ter algumas desvantagens, existem duas vantagens que o ROSRemote possui e que facilitam bastante o trabalho do desenvolver ou do usuário, que são:

- *Multicast*: conectar vários usuários em *multicast*, fazendo com que todos os clientes recebam informações de todos os outros sem intervenção direta, é bastante simples no ROSRemote. Utilizando o *SpaceBrew* basta o usuário conectar todos os publicadores a todos os assinantes. Como descrito anteriormente, é um trabalho extra que o usuário terá, mas após a realização das conexões pela primeira vez, o *SpaceBrew* as armazena e em uma próxima utilização, o usuário não necessitará realizar as conexões novamente;

- Realizar conexões em tempo de execução: com o *SpaceBrew* é possível que o usuário faça e desfaça conexões com quaisquer clientes em tempo de execução. Isso garante que o usuário não necessita realizar a interrupção da sua aplicação para desfazer uma conexão. É possível cessar o recebimento de dados de um cliente e quando necessário, realizar a conexão novamente para voltar a receber os dados.

Todas as ferramentas testadas possuem uma mesma vantagem importante para o usuário, que é a possibilidade de trabalhar com vários *Masters*. Isso é interessante para o usuário pois permite que várias plataformas diferentes possam se conectar sem que seja necessário nenhuma configuração diferente. Qualquer dispositivo (smartfones, notebooks, computadores) e qualquer sistema operacional (linux e macOS) que possa utilizar ROS pode utilizar as ferramentas e trocar informações entre si.

Além das vantagens já citadas, o ROSRemote possui uma que o coloca em posição de destaque que é a possibilidade de enviar dados para um terceiro computador sem que seja necessário configurar um novo servidor. Suponha que um usuário se encontra em um computador A, requisita um dado de um computador B e essa resposta deve ser enviada para um terceiro computador C, todos em regiões geograficamente distintas. Com o ROSRemote é possível solicitar o dado e fazer com que ele seja automaticamente encaminhado para o terceiro computador, como detalhado na Figura 4.7.1. Com o SSH e a VPN, o usuário deve receber a resposta de B e então encaminhar para C, realizando duas tarefas ao invés de apenas uma, como mostra a Figura 4.7.2. Além disso, para se conectar através de um SSH, é necessário que o usuário tenha realizado uma configuração nos computadores B e C antes de iniciar o envio dos dados.

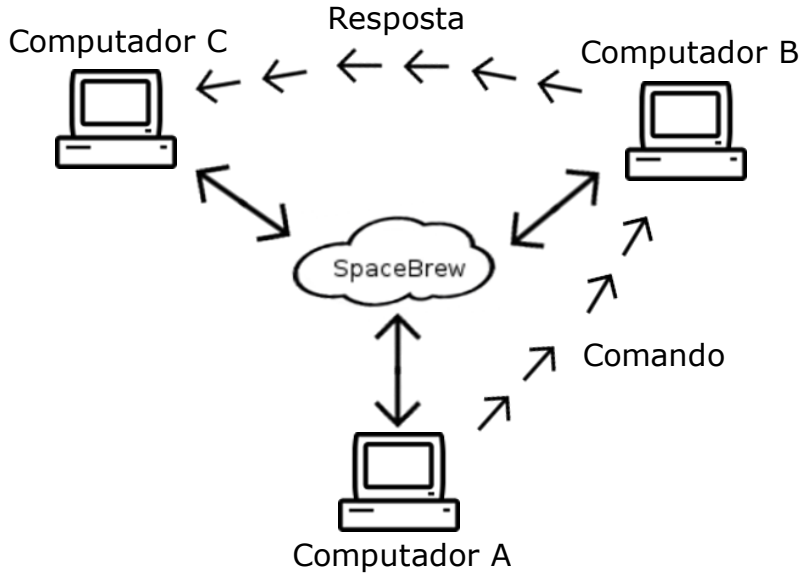

Figura 4.7.1 – Os dados do ROSRemote são trafegados através do servidor que se encontra na nuvem, mas o usuário tem a impressão de que os computadores B e C estão conectados diretamente.

Por outro lado, o ROSRemote também possui problemas e uma das principais é depender de um servidor centralizado, o que se torna um grande ponto de falha. O SSH e a VPN também utilizam servidores, porém os servidores são as próprias máquinas nas

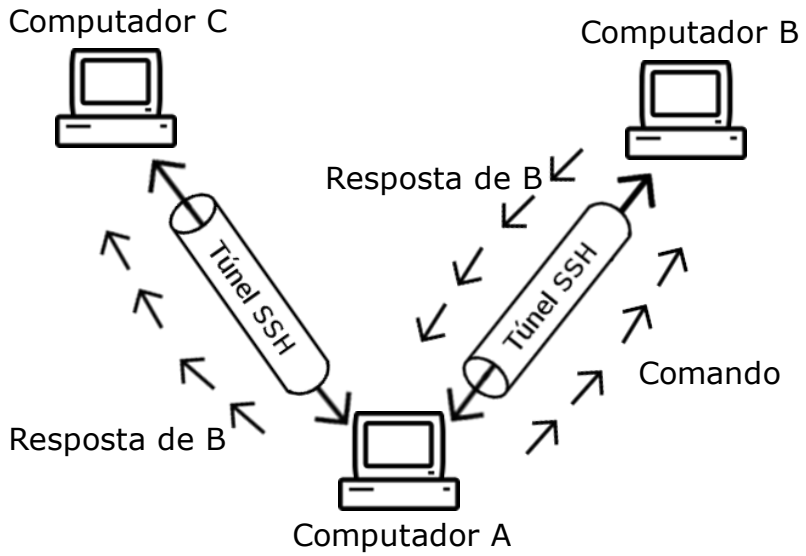

Figura 4.7.2 – Os computadores B e C não trocam informação entre si automaticamente, é necessária a interferência do usuário para encaminhar os dados.

quais o usuário irá se conectar. Desse modo, se houver uma falha na máquina o usuário também ficará sem conexão. No ROSRemote isso é um problema mais grave, visto que se o servidor parar, todas as conexões serão desfeitas mas os *Masters* remotos ainda estarão enviando comandos ao robô e o usuário não terá condições de parar esses envios, podendo ocasionar sérios danos aos robôs ou a pessoas que estiverem perto dele.

O ROSRemote também possui algumas limitações como não ser possível visualizar, ao mesmo tempo, o que dois tópicos estão publicando. Porém, nesse ponto o SSH também gerou um problema ao tentar movimentar o robô e imprimir um tópico ao mesmo tempo em dois terminais diferentes, pois ambos ficaram travados. Isso ocorre porque o SSH não permite que sejam criadas duas conexões de uma mesma origem para um mesmo destino utilizando o mesmo usuário e não é possível, nesse teste, utilizar dois usuários diferentes, pois os dados criados por um não estariam visíveis para o outro. Nesse ponto a VPN é a única ferramenta que funciona corretamente pois ela é apenas responsável pelo envio dos dados.

4.8 Considerações Finais

De acordo com os testes realizados, o ROSRemote se mostrou ser uma ferramenta mais rápida e simples do que outras que podem realizar o mesmo trabalho. Enquanto outras ferramentas obtiveram tempo de tráfego de dados de cerca de 0,3 segundo, o ROSRemote atingiu tempos de 0,006 segundo e apesar de possuir algumas desvantagens, é uma fer-

ramenta que possui vantagens que permitem concluir que ela pode ser utilizada para o controle e monitoramento de robôs a distância.

O próximo capítulo apresenta uma conclusão do trabalho e descreve possíveis trabalhos futuros.

5 Conclusão

O ROS é uma biblioteca amplamente difundida nos dias de hoje, porém só permite a utilização de dispositivos que estejam conectados em uma mesma rede. Baseado nisso, o ROSRemote foi desenvolvido utilizando-se outro *framework*, o *SpaceBrew*, e a união das duas permitiu criar uma ferramenta capaz de receber comandos localmente e enviar para um computador remoto.

A facilidade de instalação, configuração e utilização são grandes atrativos para o ROSRemote. Além disso, a possibilidade não só de utilizar os recursos já inclusos nele mas também de inserir novas funcionalidades, permitem que esse FaaS (*Framework as a Service*) auxilie nas tarefas remotas que o ROS, até hoje, não permite realizar.

Como o *SpaceBrew*, além de ser simples, permite a criação de um servidor local para realizar as conexões, isso garante que o tráfego dos dados possa ser mais seguro. Essa segurança depende do desenvolvedor porém, hoje em dia, existem muitas tecnologias de criptografia e segurança de dados que facilitam esse trabalho.

Após vários testes nos quais o tempo médio de transferência de dados foi de cerca de alguns milissegundos, é possível concluir que para a maioria dos casos o ROSRemote é um *framework* bastante viável para ser utilizado tanto para recuperar informações estáticas quanto para o controle de robôs de forma remota. Talvez, futuramente, o ROSRemote possa auxiliar na expansão das aplicações que o ROS irá abranger.

Durante a medição de velocidade e tempo de resposta, foi verificado que, mesmo o tempo de resposta não sendo muito rápido ao testá-lo com o "ping" (cerca de 300ms), o tempo total da aplicação não foi tão alto, concluindo que o processamento do *SpaceBrew* é rápido e pode ser utilizado tanto para o ROSRemote quanto para outras aplicações.

O SSH e a VPN, outras aplicações utilizadas para a transferência de dados, exigem que o usuário saiba todos os endereços e/ou nomes com os quais se deseja conectar e não permite que o usuário realize uma nova conexão ou desconecte uma máquina em tempo de execução da aplicação. Além disso, mesmo com a velocidade dessas duas ferramentas sendo alta, devido a necessidade de criptografar os dados, o ROSRemote é mais rápido que ambas.

O *Rapyuta* é uma aplicação que promete auxiliar muito no desenvolvimento de aplicações, pois permite que processamentos pesados sejam feitos fora dos robôs, diminuindo o custo deles e os tornando mais acessíveis às pessoas. Porém, existem alguns problemas de compatibilidade que dificultam que ela seja testada, sendo assim, ela pode não ser a melhor opção no momento.

Para trabalhos futuros é sugerido que, a princípio, novas funcionalidades sejam acrescentadas ao *framework* ROSRemote, tanto as que necessitem somente do *Master* quanto as que possam realizar várias tarefas utilizando robôs. Também é interessante, no caso do

usuário necessitar que suas informações sejam trafegadas de forma segura, unir o ROSRemote com o SSH ou a VPN para o envio de dados de forma criptografada. Além disso essa junção entre ROSRemote e VPN ou SSH é interessante pois retira uma limitação do *SpaceBrew*, que é não poder conectar dispositivos que se encontram em uma mesma rede. Ao se trabalhar com o ROSRemote e uma VPN, por exemplo, é possível trafegar os dados através da VPN na rede local e, quando for necessário enviar os comandos ou dados para o *Master* remoto, utilizar o *SpaceBrew*. Por fim, como esse trabalho é totalmente baseado em internet, também é interessante pesquisas em melhorias na velocidade e segurança no tráfego de dados, bem como a prevenção de perdas e a garantia que toda a comunicação seja realizada da melhor forma possível.

Como contribuições desse trabalho foi criado o pacote ROSRemote, disponível no GitHub (PEREIRA, 2017a), também houve a publicação de um artigo na conferência ICAR (*18th International Conference on Advanced Robotics*), que pode ser visto em (PEREIRA; BASTOS, 2017).

Referências

ADJIH, C. *et al.* Fit iot-lab: A large scale open experimental iot testbed. In: IEEE. *Internet of Things (WF-IoT), 2015 IEEE 2nd World Forum on*. [S.l.], 2015. p. 459–464.

ALMADA-LOBO, F. The industry 4.0 revolution and the future of manufacturing execution systems (mes). *Journal of Innovation Management*, v. 3, n. 4, p. 16–21, 2016.

AMAZON. *Amazon Prime Air*. 2017. Disponível em: <https://www.amazon.com/Amazon-Prime-Air/b?ie=UTF8&node=8037720011>. Acesso em: 1 julho 2017.

ANSWERS, R. 2011. Disponível em: <http://answers.ros.org/question/10330/whats-the-best-way-to-convert-a-ros-message-to-a-string-or-xml/>. Acesso em: 15 novembro 2016.

ARÉVALO, V. *et al.* Laboratorio remoto de automática. una solución de bajo coste basada en raspberry pi y arduino. Grupo Docente de la ETS de ICCP de la Universidad de Granada, 2017.

ARMBRUST, M. *et al.* A view of cloud computing. *Communications of the ACM*, ACM, v. 53, n. 4, p. 50–58, 2010.

ARUMUGAM, R. *et al.* Davinci: A cloud computing framework for service robots. In: IEEE. *Robotics and Automation (ICRA), 2010 IEEE International Conference on*. [S.l.], 2010. p. 3084–3089.

ASIMOV, I. Runaround. *Astounding Science Fiction*, v. 29, n. 1, p. 94–103, 1942.

ASSUNÇÃO, M. D. *et al.* Big data computing and clouds: Trends and future directions. *Journal of Parallel and Distributed Computing*, Elsevier, v. 79, p. 3–15, 2015.

BARRETT, D. J. *et al. SSH, The Secure Shell: The Definitive Guide: The Definitive Guide*. [S.l.]: "O'Reilly Media, Inc.", 2005.

BERTACCHINI, F. *et al.* Shopping with a robotic companion. *Computers in Human Behavior*, Elsevier, 2017.

BILLARD, A. Robota: Clever toy and educational tool. *Robotics and Autonomous Systems*, Elsevier, v. 42, n. 3, p. 259–269, 2003.

BILLARD, A. Robota: Clever toy and educational tool. *Robotics and Autonomous Systems*, v. 42, n. 3, p. 259 – 269, 2003. Socially Interactive Robots.

BLACKSTOCK, M.; LEA, R. Toward a distributed data flow platform for the web of things (distributed node-red). In: ACM. *Proceedings of the 5th International Workshop on Web of Things*. [S.l.], 2014. p. 34–39.

BOGUE, R. Domestic robots: Has their time finally come? *Industrial Robot: An International Journal*, Emerald Publishing Limited, v. 44, n. 2, p. 129–136, 2017.

BOREN, J.; COUSINS, S. Exponential growth of ros [ros topics]. *IEEE Robotics & Automation Magazine*, IEEE, v. 18, n. 1, p. 19–20, 2011.

BRADSKI, G. The opencv library. *Dr. Dobb's Journal: Software Tools for the Professional Programmer*, Miller Freeman Inc., v. 25, n. 11, p. 120–123, 2000.

BRADY, M. Artificial intelligence and robotics. *Artificial intelligence*, Elsevier, v. 26, n. 1, p. 79–121, 1985.

BRADY, M. Artificial intelligence and robotics. *Artificial Intelligence*, v. 26, n. 1, p. 79 – 121, 1985.

BYRNE, D. *et al.* The rise of cloud computing: Minding your p's and q's (and k's). In: NBER/CRIW CONFERENCE, MEASURING AND ACCOUNTING FOR INNOVATION IN THE 21ST CENTURY, WASHINGTON, DC, MARCH. [S.l.], 2017.

CABITZA, F. *et al.* End-user development in ambient intelligence: a user study. In: ACM. *Proceedings of the 11th Biannual Conference on Italian SIGCHI Chapter*. [S.l.], 2015. p. 146–153.

CALHEIROS, R. N. *et al.* Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and experience*, Wiley Online Library, v. 41, n. 1, p. 23–50, 2011.

CAPEK, K. *et al.* Rossum's universal robots. *Prague, CZ*, v. 1, 1920.

CARDARELLI, E. *et al.* Cooperative cloud robotics architecture for the coordination of multi-agv systems in industrial warehouses. *Mechatronics*, Elsevier, v. 45, p. 1–13, 2017.

CARMOUCHE, J. H. *IPsec virtual private network fundamentals*. [S.l.]: Pearson Education India, 2007.

CHEN, X. Decentralized computation offloading game for mobile cloud computing. *IEEE Transactions on Parallel and Distributed Systems*, IEEE, v. 26, n. 4, p. 974–983, 2015.

CHERNOUSKO, F. Locomotion principles for mobile robotic systems. *Procedia Computer Science*, Elsevier, v. 103, p. 613–617, 2017.

COMBS, G. *et al.* Wireshark-network protocol analyzer. *Version 0.99*, v. 5, 2008.

CYFRONET. *Between SaaS and PaaS: Framework as a Service*. 2012. Disponível em: <http://dice.cyfronet.pl/blog/between-saas-and-paas-framework-as-a-service>. Acesso em: 10 março 2017.

DESOLDA, G. *et al.* End-user composition of interactive applications through actionable ui components. *Journal of Visual Languages & Computing*, Elsevier, v. 42, p. 46–59, 2017.

DIAKOPOULOS, D.; KAPUR, A. Netpixl: Towards a new paradigm for networked application development. In: *NIME*. [S.l.: s.n.], 2013. p. 206–209.

DORIYA, R. *et al.* A review on cloud robotics based frameworks to solve simultaneous localization and mapping (slam) problem. *arXiv preprint arXiv:1701.08444*, 2017.

ESTRANY, B. *et al.* Multimodal human-machine interface devices in the cloud. *Journal on Multimodal User Interfaces*, Springer, p. 1–19, 2017.

FORNERO, A. K. K. Extending your business network through a virtual private network (vpn). 2016.

GAI, K. *et al.* Sa-east: security-aware efficient data transmission for its in mobile heterogeneous cloud computing. *ACM Transactions on Embedded Computing Systems (TECS)*, ACM, v. 16, n. 2, p. 60, 2017.

GHERARDI, L. *et al.* A software product line approach for configuring cloud robotics applications. In: IEEE. *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*. [S.l.], 2014. p. 745–752.

GHOSH, D. *Design and Implementation of a Unified Programming Framework for Things, Web and Cloud*. Tese (Doutorado) — McGill University, 2014.

GROUP, R. *About SpaceBrew*. 2014. Disponível em: <http://docs.spacebrew.cc/about>. Acesso em: 13 outubro 2015.

GROUP, R. *Getting Started*. 2014. Disponível em: <http://docs.spacebrew.cc/gettingstarted/>. Acesso em: 9 maio 2016.

GUNAWI, H. S. *et al. Failure as a Service (FaaS): A Cloud Service for Large-Scale, Online Failure Drills*. [S.l.], 2011. Disponível em: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-87.html>.

HAMET, P.; TREMBLAY, J. Artificial intelligence in medicine. *Metabolism-Clinical and Experimental*, Elsevier, v. 69, p. S36–S40, 2017.

HARRINGTON, B. D.; VOORHEES, C. The challenges of designing the rocker-bogie suspension for the mars exploration rover. Pasadena, CA: Jet Propulsion Laboratory, National Aeronautics and Space Administration, 2004, 2004.

HEMMERT, F.; JOOST, G. On the other hand: Embodied metaphors for interactions with mnemonic objects in live presentations. In: ACM. *Proceedings of the TEI'16: Tenth International Conference on Tangible, Embedded, and Embodied Interaction*. [S.l.], 2016. p. 211–217.

HIRATA, K. *et al.* Development of experimental fish robot. In: *Sixth International Symposium on Marine Engineering (ISME 2000), Tokyo, Japan, Oct.* [S.l.: s.n.], 2000. p. 23–27.

HU, G. *et al.* Cloud robotics: architecture, challenges and applications. *IEEE network*, IEEE, v. 26, n. 3, 2012.

INABA, M. Remote-brained robots. In: *IJCAI*. [S.l.: s.n.], 1997. p. 1593–1606.

JACKSON, J. Microsoft robotics studio: A technical introduction. *IEEE robotics & automation magazine*, IEEE, v. 14, n. 4, 2007.

KAMEI, K. *et al.* Cloud networked robotics. *IEEE Network*, IEEE, v. 26, n. 3, 2012.

KASTAN, B. Autonomous weapons systems: A coming legal singularity. *U. Ill. JL Tech. & Pol'y*, HeinOnline, p. 45, 2013.

KEAHEY, K. *et al.* Science clouds: Early experiences in cloud computing for scientific applications. *Cloud computing and applications*, v. 2008, p. 825–830, 2008.

KERR, J.; NICKELS, K. Robot operating systems: Bridging the gap between human and robot. In: IEEE. *System Theory (SSST), 2012 44th Southeastern Symposium on*. [S.l.], 2012. p. 99–104.

KOSUGE, K. *et al.* Mobile robot helper. In: *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*. [S.l.: s.n.], 2000. v. 1, p. 583–588 vol.1.

KOUBAA, A. *ROSLINK Common Message Set*. 2018. Disponível em: <http://wiki.coins-lab.org/roslink/ROSLINKCommonMessageSet.pdf>. Acesso em: 03 fevereiro 2018.

KOUBAA, A. *et al.* Roslink: Bridging ros with the internet-of-things for cloud robotics. In: _____. *Robot Operating System (ROS): The Complete Reference (Volume 2)*. [S.l.]: Springer International Publishing, 2017. p. 265–283.

LAM, M.-L.; LAM, K.-Y. Path planning as a service ppaas: Cloud-based robotic path planning. In: IEEE. *Robotics and Biomimetics (ROBIO), 2014 IEEE International Conference on*. [S.l.], 2014. p. 1839–1844.

LANKES, M. *et al.* An eye for an eye: Gaze input in competitive online games and its effects on social presence. In: ACM. *Proceedings of the 13th International Conference on Advances in Computer Entertainment Technology*. [S.l.], 2016. p. 17.

LI, A. W.; BASTOS, G. S. A hybrid self-adaptive particle filter through kld-sampling and samcl. In: IEEE. *Advanced Robotics (ICAR), 2017 18th International Conference on*. [S.l.], 2017. p. 106–111.

LI, Y. *et al.* Intelligent cryptography approach for secure distributed big data storage in cloud computing. *Information Sciences*, Elsevier, v. 387, p. 103–115, 2017.

LITMAN, T. Autonomous vehicle implementation predictions. *Victoria Transport Policy Institute*, v. 28, 2014.

LIU, X. *et al.* Wids checker: Combating bugs in distributed systems. In: *NSDI*. [S.l.: s.n.], 2007. p. 19–19.

LORENCIK, D.; SINCAK, P. Cloud robotics: Current trends and possible use as a service. In: IEEE. *Applied Machine Intelligence and Informatics (SAMI), 2013 IEEE 11th International Symposium on*. [S.l.], 2013. p. 85–88.

LU, H. *et al.* Brain intelligence: go beyond artificial intelligence. *Mobile Networks and Applications*, Springer, p. 1–8, 2017.

LUBBERS, P. *HTML5 WebSocket: Full-duplex, Real-time Web Communication*. [S.l.]: DZone, 2011.

LUND, H. H.; NIELSEN, J. An edutainment robotics survey. In: CITESEER. *HART2002*. [S.l.], 2002.

MANZI, A. *et al.* Design of a cloud robotic system to support senior citizens: The kubo experience. *Autonomous Robots*, Springer, v. 41, n. 3, p. 699–709, 2017.

MARKOFF, J. Google cars drive themselves, in traffic. *The New York Times*, v. 10, n. A1, p. 9, 2010.

MAYER, J. T. A. *PySpacebrew*. 2013. Disponível em: <https://github.com/Spacebrew/pySpacebrew>. Acesso em: 05 junho 2016.

MCGOVERN, S. *et al.* Fast bender actuators for fish-like aquatic robots. In: INTERNATIONAL SOCIETY FOR OPTICS AND PHOTONICS. *Electroactive Polymer Actuators and Devices (EAPAD) 2008*. [S.l.], 2008. v. 6927, p. 69271L.

MELL, P. *et al.* The nist definition of cloud computing. Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology Gaithersburg, 2011.

MERIZALDE, J. L. P. *et al. Estudio del modelo de referencia del Internet de las Cosas (IoT), con la implementación de un prototipo domótico*. Dissertação (B.S. thesis) — Quito, 2016., 2016.

MICROSOFT. *VPN Tunneling Protocols*. 2017. Disponível em: <https://technet.microsoft.com/en-us/library/6e3cd69c-dc8c-483e-98bc-8d2e7e76e048>. Acesso em: 1 novembro 2017.

MOHANARAJAH, G. *et al.* Rapyuta: A cloud robotics platform. *IEEE Transactions on Automation Science and Engineering*, IEEE, v. 12, n. 2, p. 481–493, 2015.

MOHANARAJAH, G. *et al.* Cloud-based collaborative 3d mapping in real-time with low-cost robots. *IEEE Transactions on Automation Science and Engineering*, IEEE, v. 12, n. 2, p. 423–431, 2015.

MURER, M. *et al.* Torquescreen: Actuated flywheels for ungrounded kinaesthetic feedback in handheld devices. In: ACM. *Proceedings of the Ninth International Conference on Tangible, Embedded, and Embodied Interaction*. [S.l.], 2015. p. 161–164.

NASA. *Mars Exploration Rover Mission*. 2016. Disponível em: <http://mars.nasa.gov/mer/home/>. Acesso em: 10 agosto 2016.

NEHMZOW, U. *Mobile robotics: a practical introduction*. [S.l.]: Springer Science & Business Media, 2012.

NILSSON, N. J. *Shakey the robot*. [S.l.], 1984.

ODIYO, B.; DWARKANATH, M. Virtual private network. *Uppsala universitet (accessed on November 2011)*, 2011.

O'KANE, J. M. *A gentle introduction to ROS*. [S.l.]: Jason M. O'Kane, 2014. Acesso em: 13 janeiro 2018.

OTERO, L. E. A. Uma arquitetura para a implantação automática de serviços em infraestruturas de nuvem. Universidade Federal de Pernambuco, 2013.

PANDEY, S. *et al.* A particle swarm optimization-based heuristic for scheduling workflow applications in cloud computing environments. In: IEEE. *Advanced information networking and applications (AINA), 2010 24th IEEE international conference on*. [S.l.], 2010. p. 400–407.

PANDIT, D. D. V.; POOJARI, A. A study on amazon prime air for feasibility and profitability–a graphical data analysis. *IOSR Journal of Business and Management*, v. 16, n. 11, p. 06–11, 2014.

PEREIRA, A. *CloudRos*. 2017. Disponível em: <https://github.com/alysonmp/ROSRemote>. Acesso em: 20 novembro 2017.

PEREIRA, A. *ROSRemote*. 2017. Disponível em: <http://wiki.ros.org/ROSRemote>. Acesso em: 20 novembro 2017.

PEREIRA, A. B.; BASTOS, G. S. Rosremote, using ros on cloud to access robots remotely. In: IEEE. *Advanced Robotics (ICAR), 2017 18th International Conference on*. [S.l.], 2017. p. 284–289.

PIMENTEL, V.; NICKERSON, B. G. Communicating and displaying real-time data with websocket. *IEEE Internet Computing*, IEEE, v. 16, n. 4, p. 45–53, 2012.

POCZTER, S. L.; JANKOVIC, L. M. The google car: driving toward a better future? *Journal of Business Case Studies (Online)*, The Clute Institute, v. 10, n. 1, p. 7, 2014.

QUIGLEY, M. *et al.* Ros: an open-source robot operating system. In: KOBE. *ICRA workshop on open source software*. [S.l.], 2009. v. 3, n. 3.2, p. 5.

RAIBERT, M. H. Legged robots. *Communications of the ACM*, ACM, v. 29, n. 6, p. 499–514, 1986.

RAPYUTA. *Rapyuta Robotics*. 2015. Disponível em: <http://www.rapyuta.org/>. Acesso em: 20 dezembro 2017.

REN, F. Robotics cloud and robotics school. In: IEEE. *Natural Language Processing andKnowledge Engineering (NLP-KE), 2011 7th International Conference on*. [S.l.], 2011. p. 1–8.

REYNOLDS, P. *et al.* Pip: Detecting the unexpected in distributed systems. In: *NSDI*. [S.l.: s.n.], 2006. v. 6, p. 9–9.

RIAZUELO, L.; ONTHERS. C 2 tam: A cloud framework for cooperative tracking and mapping. *Robotics and Autonomous Systems*, Elsevier, v. 62, n. 4, p. 401–413, 2014.

RIMAL, B. P. *et al.* A taxonomy and survey of cloud computing systems. In: *Proceedings of the 2009 Fifth International Joint Conference on INC, IMS and IDC*. Washington, DC, USA: IEEE Computer Society, 2009. (NCM '09), p. 44–51. ISBN 978-0-7695-3769-6. Disponível em: <http://dx.doi.org/10.1109/NCM.2009.218>.

ROBOTICS, C. *ROS 101: Intro to the Robot Operating System*. 2014. Disponível em: <http://robohub.org/ros-101-intro-to-the-robot-operating-system/>. Acesso em: 4 agosto 2016.

ROS. *ROS Concepts*. 2007. Disponível em: <http://wiki.ros.org/ROS/Concepts>. Acesso em: 9 maio 2016.

ROS. 2016. Disponível em: <http://wiki.ros.org/rosbash#rosrun/>. Acesso em: 4 agosto 2016.

ROS, W. *Master*. 2016. Disponível em: <http://wiki.ros.org/Master>. Acesso em: 4 agosto 2016.

ROS, W. *Nodes*. 2016. Disponível em: <http://wiki.ros.org/Nodes>. Acesso em: 4 agosto 2016.

ROS, W. *rqt_graph*. 2016. Disponível em: <http://wiki.ros.org/rqt_graph>. Acesso em: 4 agosto 2016.

ROS, W. *Services*. 2016. Disponível em: <http://wiki.ros.org/Services>. Acesso em: 4 agosto 2016.

ROS, W. *srv*. 2016. Disponível em: <http://wiki.ros.org/srv>. Acesso em: 4 agosto 2016.

ROS, W. *Topics*. 2016. Disponível em: <http://wiki.ros.org/Topics>. Acesso em: 4 agosto 2016.

ROSA, J.; ROCHA, R. P. Exportation to the cloud of distributed robotic tasks implemented in ros. In: ACM. *Proceedings of the Symposium on Applied Computing*. [S.l.], 2017. p. 235–240.

ROSSUM, G. V. *et al.* Python programming language. In: *USENIX Annual Technical Conference*. [S.l.: s.n.], 2007. v. 41, p. 36.

SADOOGHI, I. *et al.* Understanding the performance and potential of cloud computing for scientific applications. *IEEE Transactions on Cloud Computing*, IEEE, v. 5, n. 2, p. 358–371, 2017.

SÁNCHEZ-MARTÍN, F. *et al.* Historia de la robótica: de arquitas de tarento al robot da vinci.(parte i). *Actas Urológicas Españolas*, Elsevier, v. 31, n. 2, p. 69–76, 2007.

SANTOS, J. M. *et al.* An evaluation of 2d slam techniques available in robot operating system. In: IEEE. *Safety, Security, and Rescue Robotics (SSRR), 2013 IEEE International Symposium on*. [S.l.], 2013. p. 1–6.

SCOTT, C. *et al. Virtual private networks*. [S.l.]: "O'Reilly Media, Inc.", 1999.

SEGRERA, F. *et al.* Avatar medium: Disembodied embodiment, fragmented communication. 2013.

SHINDE, S. *et al.* Abhikaha: Aerial collision avoidance in quadcopter using cloud robotics. *International Journal of Research In Science and Engineering*, 2016.

SIEGWART, R. *et al. Introduction to autonomous mobile robots*. [S.l.]: MIT press, 2011.

SULTAN, N. Cloud computing for education: A new dawn? *International Journal of Information Management*, Elsevier, v. 30, n. 2, p. 109–116, 2010.

TODD, D. J. *Walking machines: an introduction to legged robots*. [S.l.]: Springer Science & Business Media, 2013.

TOMOYA, A. *et al.* A mobile robot for following, watching and detecting falls for elderly care. *Procedia Computer Science*, Elsevier, v. 112, p. 1994–2003, 2017.

TURNBULL, L.; SAMANTA, B. Cloud robotics: Formation control of a multi robot system utilizing cloud infrastructure. In: IEEE. *Southeastcon, 2013 Proceedings of IEEE*. [S.l.], 2013. p. 1–4.

TZAFESTAS, S. G. *Introduction to mobile robot control*. [S.l.]: Elsevier, 2013.

VALADÃO, C. T. *et al.* Robot toys for children with disabilities. In: *Computing in Smart Toys*. [S.l.]: Springer, 2017. p. 55–84.

VECCHIOLA, C. *et al.* High-performance cloud computing: A view of scientific applications. In: IEEE. *Pervasive Systems, Algorithms, and Networks (ISPAN), 2009 10th International Symposium on*. [S.l.], 2009. p. 4–16.

WAIBEL, M. *et al.* Roboearth. *IEEE Robotics & Automation Magazine*, IEEE, v. 18, n. 2, p. 69–82, 2011.

WALTER, W. G. An electromechanical animal. Dialectica, v. 4, n. 3, p. 206–213, 1950.

WAN, J. *et al.* Cloud robotics: current status and open issues. *IEEE Access*, IEEE, v. 4, p. 2797–2807, 2016.

WAN, J. *et al.* Cloud robotics: Current status and open issues. *IEEE Access*, IEEE, v. 4, p. 2797–2807, 2016.

WEAVER, J. N. *et al.* Uav performing autonomous landing on usv utilizing the robot operating system. In: CITESEER. *Proc. of the ASME District F-Early Career Technical Conference*. [S.l.], 2013.

WELCH, A. A cost-benefit analysis of amazon prime air. University of Tennessee at Chattanooga, 2015.

WOOD, T. *et al.* The case for enterprise-ready virtual private clouds. In: *HotCloud*. [S.l.: s.n.], 2009.

WOWWEE. *Friendship at your fingertips*. 2018. Disponível em: <https://wowwee.com/>.

WRIGHT, C. *et al.* Design of a modular snake robot. In: IEEE. *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on*. [S.l.], 2007. p. 2609–2614.

YANG, C. *et al.* Big data and cloud computing: innovation opportunities and challenges. *International Journal of Digital Earth*, Taylor & Francis, v. 10, n. 1, p. 13–53, 2017.

YLONEN, T.; LONVICK, C. The secure shell (ssh) protocol architecture. 2006.

ZDEŠAR, A. *et al.* Engineering education in wheeled mobile robotics. *IFAC-PapersOnLine*, Elsevier, v. 50, n. 1, p. 12173–12178, 2017.

ZHANG, Q. *et al.* Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, Springer, v. 1, n. 1, p. 7–18, 2010.

ZHOU, K. *et al.* Industry 4.0: Towards future industrial opportunities and challenges. In: IEEE. *Fuzzy Systems and Knowledge Discovery (FSKD), 2015 12th International Conference on*. [S.l.], 2015. p. 2147–2152.

ZWEIGLE, O. *et al.* Roboearth: connecting robots worldwide. In: ACM. *Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human*. [S.l.], 2009. p. 184–191.