

Josué Oliveira de Araújo

Sistema alocador de tarefas para robôs de entrega utilizando ROS

Natal – RN

Novembro de 2019

Josué Oliveira de Araújo

Sistema alocador de tarefas para robôs de entrega utilizando ROS

Trabalho de Conclusão de Curso de Engenharia de Computação da Universidade Federal do Rio Grande do Norte, apresentado como requisito parcial para a obtenção do grau de Bacharel em Engenharia de Computação

Orientador: Prof. Dsc. Bruno Marques
Ferreira da Silva

Universidade Federal do Rio Grande do Norte – UFRN
Departamento de Engenharia de Computação e Automação – DCA
Curso de Engenharia de Computação

Natal – RN
Novembro de 2019

Josué Oliveira de Araújo

Sistema alocador de tarefas para robôs de entrega utilizando ROS

Trabalho de Conclusão de Curso de Engenharia de Computação da Universidade Federal do Rio Grande do Norte, apresentado como requisito parcial para a obtenção do grau de Bacharel em Engenharia de Computação

Orientador: Prof. Dsc. Bruno Marques
Ferreira da Silva

Trabalho aprovado. Natal – RN, 27 de novembro de 2019:

Prof. DSc. Bruno Marques Ferreira da Silva - Orientador
UFRN

Prof. DSc. Luiz Marcos Garcia Gonçalves
UFRN

Prof. DSc. Orivaldo Vieira de Santana Júnior
UFRN

MSc. Davi Henrique dos Santos
UFRN

Natal – RN
Novembro de 2019

À minha família, por sempre acreditarem e investirem em mim. Amo vocês.

AGRADECIMENTOS

Não poderia iniciar de outra forma, em primeiro lugar agradeço a Deus por me dar muito mais do que eu mereço. Sem Ele nada sou.

Agradeço aos meus pais, Marinilton e Josineide, e aos meus irmãos, Matheus e Julianne. Toda confiança depositada, os imensuráveis esforços feitos e o carinho demonstrado foram essenciais para conclusão dessa etapa.

Agradeço à minha esposa Allanda por todo amor, carinho e cuidado de sempre. Sem o seu incentivo eu não teria chegado até aqui.

Agradeço aos meus amigos, tanto de Mossoró quanto de Natal, por serem fonte de alívio e regozijo nos momentos difíceis.

Agradeço ao meu orientador, Bruno Silva, por ter me feito trocar a mecânica pela computação e por estar sempre disposto a me ajudar, com muita paciência e atenção.

Agradeço a Universidade Federal do Rio Grande do Norte, seu corpo docente e administrativo, por me proporcionarem anos de muito aprendizado e crescimento pessoal.

*“Ao Rei eterno, ao Deus único, imortal e invisível,
sejam honra e glória para todo o sempre.
Amém.”(1 Tm 1:17)*

RESUMO

A proposta desse trabalho é implementar um sistema capaz de controlar múltiplos robôs, alocando tarefas para os mesmos. O serviço desejado é o de entrega de cafés. Para tanto, utilizar-se-á o ROS *Robot Operating System*, que abstrai os conceitos necessários para, por exemplo, fazer um robô se locomover. Dessa forma, pode-se concentrar os esforços no desenvolvimento do sistema em si, pois o ROS trará as ferramentas necessárias para tratar os envios de mensagens, controle dos robôs, cálculo de rotas e demais processos que ocorrem em baixo nível. A linguagem utilizada para o desenvolvimento do sistema foi Python. O sistema foi testado em um ambiente simulado no Gazebo utilizando turtlebots.

Palavras-chaves: ROS. Múltiplos robôs. Robô entregador. Move base.

ABSTRACT

The proposal of this work is to implement a system capable of controlling multiple robots, allocating tasks for them. The desired service is the coffee delivery. To do so, will be used the ROS *Robot Operating System*, which abstract the necessary concepts to make the robot move, for example. That way, efforts can be concentrated on the development of the system, because the ROS will bring the necessary tools to handle messaging, robots control, route calculation and other low-level processes. The language used for the development of this system was Python. The system was tested in a simulated environment in Gazebo using turtlebots.

Keywords: ROS. Multiple robots. Delivery robot. Move base

LISTA DE ILUSTRAÇÕES

| | |
|--|----|
| Figura 1 – Exemplo básico de um grafo ROS | 17 |
| Figura 2 – Exemplo complexo de um grafo ROS | 18 |
| Figura 3 – Exemplo de uma TF tree | 21 |
| Figura 4 – <i>Print screen</i> do terminal listando a action | 22 |
| Figura 5 – Comportamento esperado do move_base | 23 |
| Figura 6 – Estrutura de organização dos arquivos launch | 26 |
| Figura 7 – Diagrama da arquitetura do software | 28 |
| Figura 8 – <i>Print screen</i> do smach-viewer mostrando a máquina de estados de um dos robôs | 30 |
| Figura 9 – <i>Print screen</i> do Gazebo apresentando o <i>floorplan.world</i> | 31 |
| Figura 10 – <i>Print screen</i> do rviz apresentando o mapa do ambiente | 32 |
| Figura 11 – <i>Print screen</i> do rviz e do smach-viewer mostrando os robôs em rota de entrega | 33 |
| Figura 12 – <i>Print screen</i> do gazebo mostrando os robôs em rota de entrega | 33 |

LISTA DE ABREVIATURAS E SIGLAS

| | |
|--------|--|
| ROS | <i>Robot Operating System</i> |
| AMCL | <i>Adaptive Monte Carlo Localization</i> |
| TCP/IP | <i>Transmission Control Protocol/Internet Protocol</i> |
| UDP | <i>User Datagram Protocol</i> |
| SLAM | <i>Simultaneous Localization and Mapping</i> |
| P2P | <i>Peer-to-peer</i> |

SUMÁRIO

| | | |
|------------|---|-----------|
| 1 | INTRODUÇÃO | 12 |
| 1.1 | Motivação | 12 |
| 1.2 | Contribuições | 12 |
| 1.3 | Estrutura do Trabalho | 12 |
| 2 | ROS | 14 |
| 2.1 | O que é? | 14 |
| 2.2 | Importância | 15 |
| 2.3 | Funcionamento | 16 |
| 2.3.1 | Nós | 16 |
| 2.3.2 | Tópicos | 16 |
| 2.3.3 | Grafo | 16 |
| 2.3.4 | Arquivos launch | 18 |
| 2.3.5 | Namespaces | 19 |
| 2.3.6 | TF | 19 |
| 2.4 | Actions | 21 |
| 2.5 | Pacotes utilizados | 22 |
| 2.5.1 | AMCL | 22 |
| 2.5.2 | move_base | 22 |
| 3 | TRABALHOS RELACIONADOS | 24 |
| 4 | IMPLEMENTAÇÃO | 25 |
| 4.1 | Instalando pré-requisitos | 25 |
| 4.2 | Configuração do ambiente | 25 |
| 4.2.1 | Entendendo os arquivos launch | 26 |
| 4.3 | Arquitetura do software | 27 |
| 4.4 | Robot Delivery Action | 28 |
| 4.5 | Máquina de estados | 29 |
| 5 | EXPERIMENTOS | 31 |
| 5.1 | Ambiente | 31 |
| 5.1.1 | Gazebo | 31 |
| 5.1.2 | Rviz | 32 |
| 5.2 | Simulações | 32 |
| 5.3 | Eventuais problemas e limitações | 34 |

| | | |
|------------|---|-----------|
| 5.3.1 | Comunicação | 34 |
| 5.3.2 | Colisão | 34 |
| 5.3.3 | Otimização | 35 |
| 5.4 | Robôs reais | 35 |
| 6 | CONCLUSÃO | 36 |
| | Referências | 37 |
| | APÊNDICE A – ROBOTS_GAZEBO_RVIZ.LAUNCH | 39 |
| | APÊNDICE B – ROBOTS.LAUNCH | 40 |
| | APÊNDICE C – ROBOT.LAUNCH | 41 |
| | APÊNDICE D – MOVE_BASE.LAUNCH | 44 |
| | APÊNDICE E – AMCL.LAUNCH | 46 |

1 INTRODUÇÃO

Tarefas simples do cotidiano vêm sendo repensadas utilizando robôs, como limpar a casa [24] ou entregar materiais em hospitais [9]. Pode-se inferir o rápido crescimento em sistemas como esses à middlewares que abstraem as tarefas de baixo nível do robô, agilizando o desenvolvimento de aplicações, como o ROS [12]. O alto nível de aceitação e de entusiasmo com essa ferramenta pode ser expresso através do seu crescimento exponencial [3]. A base desse trabalho será o ROS. Será configurado um ambiente utilizando cenários previamente construídos e já mapeados [10].

1.1 Motivação

Trazer mordomia e conforto aos utilizadores seria o foco desse trabalho, tendo em vista que, mesmo não sendo uma tarefa difícil de ser realizada por pessoas, receber o café em sua mesa com apenas um clique parece razoável. Esse sistema se encaixaria em diversos ambientes diferentes, desde universidades até edifícios empresariais.

Além disso, esse trabalho poderá servir como base para outros semelhantes que desejem alocar simples tarefas como essa para robôs. Um sistema que controle robôs que levem documentos de uma sala à outra em um prédio empresarial, seria um exemplo disso.

1.2 Contribuições

O foco será então o desenvolvimento de um sistema alocador de tarefas para múltiplos robôs, onde a tarefa seria a de entregar cafés. Deseja-se implementar um sistema completo, desde o software do cliente solicitando entregas de cafés em determinada localização, passando pelo software central, que receberá os pedidos e os despachará para os entregadores, que serão turtlebots [11] equipados com câmeras RGB-D do tipo Kinect e controlados pelo programa que gerencia o comportamento dos robôs, utilizando máquinas de estados.

1.3 Estrutura do Trabalho

Este trabalho apresenta o passo a passo dessa implementação, desde uma visão geral sobre o ROS que é a base do sistema no Capítulo 2 e mostrando trabalhos relacionados no Capítulo 3. Todo o processo de implementação estará descrito no Capítulo 4, desde a instalação de pré-requisitos até a arquitetura do software. No Capítulo 5, estão disponíveis

informações acerca dos experimentos realizados. Por fim, tem-se a conclusão no Capítulo 6, onde pode se ver um panorama do que foi alcançado nesse trabalho bem como possibilidades de trabalhos futuros.

2 ROS

Neste capítulo será tratado acerca do ROS *Robot Operating System* que é a base para o software que será desenvolvido nesse trabalho. Assimilar os conceitos que serão expostos a seguir é fundamental para entender a implementação do software do qual se trata esse trabalho.

2.1 O que é?

Na página inicial da Wiki do ROS, encontra-se a seguinte descrição:

"ROS (Sistema Operacional de Robôs) fornece bibliotecas e ferramentas para ajudar desenvolvedores de software a criar aplicações para robôs. Ele fornece abstração de hardware, drivers de dispositivos, bibliotecas, visualizadores, passagem de mensagens, gerenciamento de pacotes e muito mais. ROS está sob a licença BSD de código aberto." [19, tradução nossa].

Segundo Quigley [13], o ROS é um projeto enorme com muitos colaboradores. Ele surgiu a partir da necessidade sentida pela comunidade de pesquisa em robótica de um framework de colaboração aberta. Vários projetos da Universidade de Stanford em meados dos anos 2000 como o Stanford AI Robot (STAIR) e o Personal Robots (PR), criaram protótipos e sistemas de softwares. Já em 2007, a Willow Garage [7] disponibilizou uma quantidade significativa de recursos para ampliar muito mais esses conceitos e criar implementações bem testadas. Esse esforço foi impulsionado por incontáveis pesquisadores que contribuíram com o seu tempo e conhecimento acerca da base do ROS e seus pacotes de software fundamentais. Com o tempo, ele passou a ser extensamente utilizado pela comunidade de pesquisa em robótica [4].

Cinco filosofias do ROS foram descritas em [12], são elas:

- Peer-to-peer

O ROS consiste em vários pequenos programas de computador que se conectam entre si e trocam mensagens continuamente. Essas mensagens são enviadas diretamente de um programa para outro, sem passar por um serviço central de distribuição de mensagens.

- Tools-based

Diferente de muitos outros frameworks de robótica que possuem um ambiente de desenvolvimento e execução engessados, o ROS, por ser baseado em uma arquitetura Unix, onde sistemas de software complexos são criados a partir de vários pequenos

programas genéricos, utiliza-se de programas separados para realizar tarefas como navegar na árvore do código fonte, visualizar as interconexões do sistema, plotar gráficos de fluxo de dados, gerar documentação, registrar dados e etc. Isso encoraja a criação de novas versões desses pequenos programas, com implementações melhoradas, tendo em vista que facilmente pode-se substituir o programa que realiza determinada tarefa.

- Multilingual

Muitas implementações de software são mais facilmente alcançadas utilizando linguagens de "alta produtividade" como Python ou Ruby. Assim como, para tarefas onde as exigências de performance são maiores, é melhor utilizar C++. Também há várias razões que levam alguns programadores a preferirem linguagens como Lisp ou MATLAB. Sabendo que há várias opiniões divergentes dos programadores e que cada linguagem tem suas particularidades em diferentes contextos, o ROS tem uma dimensão multilingual. Os módulos de software ROS podem ser escritos em qualquer linguagem que tenha a biblioteca cliente escrita. Essas bibliotecas existem para as linguagens C++, Python, LISP, Java, JavaScript, MATLAB, Ruby, Haskell, R, Julia, dentre outras.

- Thin

As convenções ROS encorajam os contribuidores a criar softwares de forma independente do ROS e depois envolvê-los em um wrapper de modo que possam enviar e receber mensagens de outros módulos ROS. Essa camada extra tem a intenção de permitir a reutilização do software fora do ROS em outras aplicações e isso também contribui para facilitar a criação de testes automatizados utilizando ferramentas de integração contínuas.

- Free and open source

O core do ROS está sob licença BSD, que permite o uso comercial ou não. Isso permite o seu uso tanto em ambientes acadêmicos onde geralmente os códigos são completamente abertos, bem como em produtos comerciais onde geralmente são feitos de forma privada. Todos os usos são válidos sob a licença do ROS.

2.2 Importância

O desenvolvimento de softwares para robôs era uma tarefa extremamente difícil, pois exige um vasto conjunto de conhecimento, desde controladores para drivers de diferentes tipos de robôs, até algoritmos de percepção. A quantidade de conhecimento necessária é tamanha, que uma equipe era necessária para tratar todas as minúcias. Contudo, a chegada do ROS acarretou uma diminuição no esforço despendido para realizar essa tarefa,

pois a curva de aprendizado foi extremamente reduzida, não sendo necessário conhecer profundamente todo o processo como antes, devido ao nível de abstração que o ROS traz. Com isso, é possível focar mais no desenvolvimento do software em si, do que no processo necessário para fazer o robô se localizar em um mapa, por exemplo.

2.3 Funcionamento

Para entender o funcionamento do ROS, é conveniente subdividi-lo e explicar cada parte específica, conforme disposto a seguir.

2.3.1 Nós

Os pequenos programas descritos na seção 2.1, são chamados de nós ROS. Esse termo poderia ser substituído por módulos de softwares, contudo, tendo em vista que esses programas se comunicam entre si de forma peer-to-peer, acaba gerando uma estrutura de grafo (subseção 2.3.3). Por isso, é conveniente chamá-los de nós do grafo ROS. Todo nó pode publicar mensagens em tópicos (subseção 2.3.2) para que outros nós que se interessem pelos dados dessa mensagem se inscrevam nesse tópico e recebam essas mensagens.

2.3.2 Tópicos

São os canais por onde os nós ROS podem conversar. Um tópico tem um determinado tipo de mensagem, denominado ROS Messages, que pode ser desde uma simples string até um tipo de mensagem customizada que carrega as informações desejadas pelo usuário. Em cada tópico pode haver dois tipos de nós: os que publicam, e os assinantes, que escutam as mensagens. Essa ideia poderá ser melhor compreendida ao ler a subseção seguinte.

2.3.3 Grafo

Como já foi citado anteriormente, o ROS se baseia em vários programas pequenos que se comunicam entre si através de mensagens P2P. Assim, forma-se uma estrutura chamada de Grafo ROS, onde os nós do grafo são os programas e as arestas são os links P2P.

A Figura 1 mostra um exemplo simples de Grafo ROS onde a comunicação está ocorrendo em apenas uma direção sempre, formando um pipeline. Há um nó *microphone* que capta o som e publica essa mensagem em um tópico. Então o segundo nó, *speech recognition*, que está inscrito nesse mesmo tópico, recebe essa mensagem e realiza o processamento de reconhecimento de fala. Feito isso, ela publica o resultado desse processamento em um novo tópico que tem o nó *dialog manager* inscrito nele. Esse nó recebe a mensagem, realiza o processamento do diálogo e publica o resultado em um novo tópico e assim por diante.

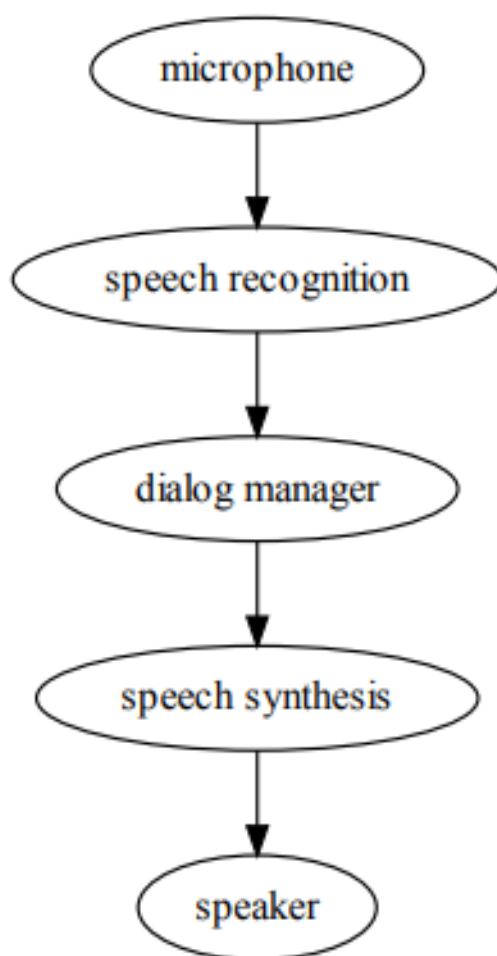


Figura 1 – Exemplo básico de um grafo ROS

Fonte – [12, p. 3]

Trata-se de um exemplo bem simples de um grafo, contudo, de forma geral, os grafos ROS são bem maiores. A Figura 2 mostra um exemplo mais complexo de Grafo ROS, onde há muitos nós e muitos deles publicam em vários tópicos diferentes, assim como há nós que também estão inscritos em múltiplos tópicos.

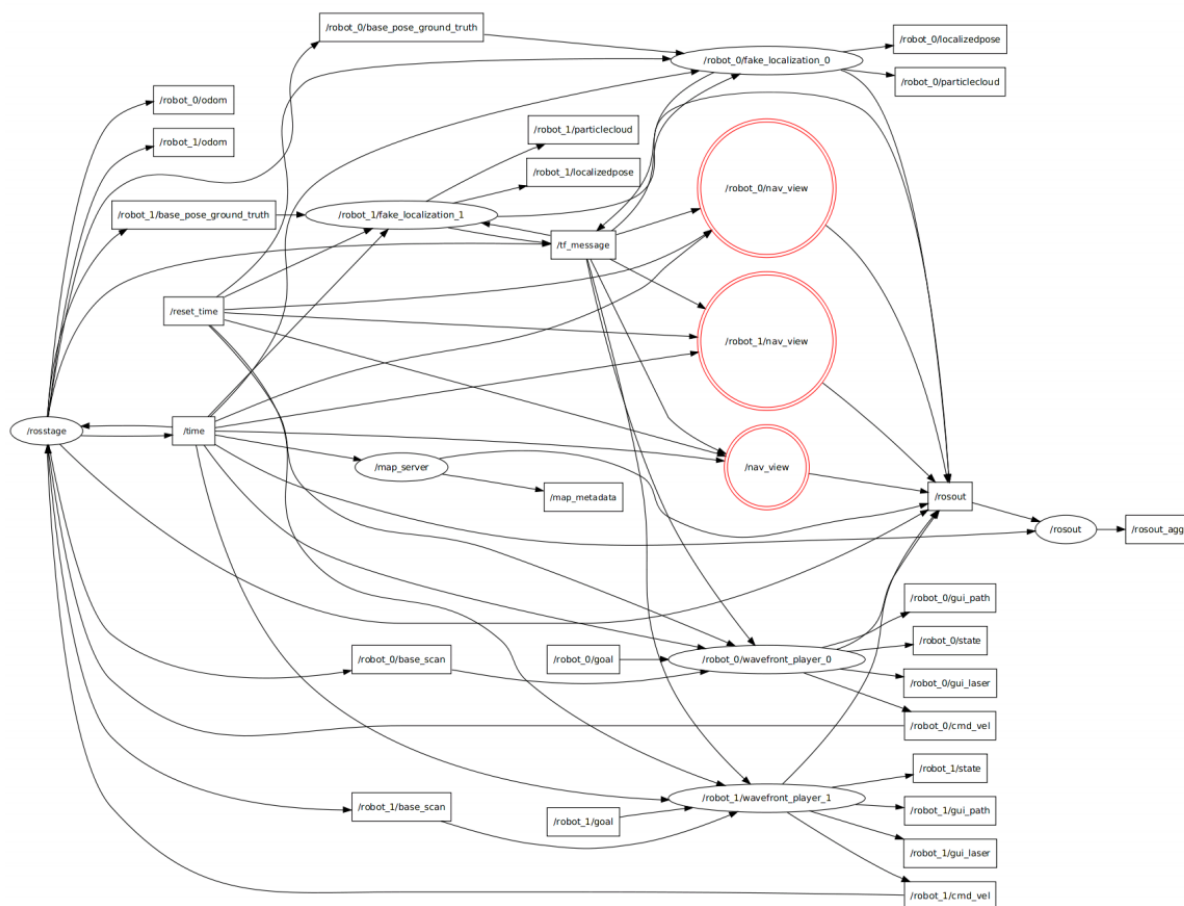


Figura 2 – Exemplo complexo de um grafo ROS

Fonte – [12, p. 5]

2.3.4 Arquivos launch

A inicialização de um nó ROS pode ser feita através de linha de comando no terminal, contudo, em um sistema mais complexo, onde há uma incontável quantidade de nós, essa prática se torna completamente inviável. Os arquivos launch são a solução para essa tarefa, eles são arquivos escritos em um formato XML específico cuja extensão é *.launch*.

No launch file é possível configurar os parâmetros que cada nó precisa. Por exemplo, o nó `map_server` precisa receber como parâmetro um arquivo de um mapa para poder ser iniciado. O caminho para este arquivo deve ser configurada no arquivo `.launch`. Outro exemplo de parâmetro a ser configurado é a posição inicial dos robôs no ambiente de simulação ou então o tipo de sensor que o `turtlebot` vai utilizar.

2.3.5 Namespaces

Quando se deseja trabalhar com apenas um robô, esse conceito não é tão importante, contudo, como trata-se de uma aplicação utilizando múltiplos robôs, entender o namespace do ROS é fundamental. Ele será o responsável por distinguir o que se refere a cada robô.

Para facilitar a compreensão, pode-se imaginar uma situação hipotética onde inicialmente há apenas um robô e os seguintes nós foram iniciados:

```
/map  
/amcl  
/scan  
/move_base
```

Todos os nós foram iniciados no namespace global, ou seja na *raiz* /. Se por ventura, outro robô vier a ser instanciado dessa mesma forma, um problema irá ocorrer. A leitura dos lasers dos dois robôs estarão sendo publicados no mesmo tópico */scan*, ambos os robôs responderão a mesma instância do */move_base* e só haverá um *amcl* para os dois robôs. O correto seria cada robô ter o seu próprio namespace, ficando da seguinte forma:

```
/map  
  
/robot0/amcl  
/robot0/scan  
/robot0/move_base  
  
/robot1/amcl  
/robot1/scan  
/robot1/move_base
```

Assim, cada robô terá o seu próprio namespace. Vale ressaltar que o map continua no namespace global pois só é necessário um mapa para os dois robôs, ou seja, o nó *map_server* está publicando um mapa no tópico */map* e os dois robôs estão inscritos nesse tópico.

Na Figura 2 é possível perceber que alguns nós possuem o nome do robô no início da sua nomenclatura. Isso se dá por conta do namespace. O namespace pode ser configurado nos arquivos launch (subseção 2.3.4).

2.3.6 TF

Gerenciar os sistemas de coordenadas é uma tarefa crucial para o funcionamento de um robô. Tendo em vista a necessidade de, a todo momento, precisar saber a posição relativa de cada parte do robô à uma outra. Por exemplo, para que o robô saiba com

exatidão a distância em que ele se encontra de determinado obstáculo, ele precisa saber exatamente em qual posição da sua base o laser está posicionado. Imaginando uma situação onde o laser do robô está posicionado na parte de trás da base, mas para o robô ele está posicionado na parte da frente da sua base, inevitavelmente ocorrerá uma colisão, pois haverá uma incongruência no cálculo da posição do robô em relação à posição real do mesmo. Outro exemplo comum, seria um robô com um braço mecânico tentando pegar um objeto. Se a posição relativa do gancho mecânico não estiver em sintonia com a real posição do robô, o movimento realizado para pegar o objeto não irá chegar no destino correto.

Imaginando então um cenário onde o robô está constantemente se movimentando e o braço mecânico também está constantemente se mexendo, conseguir calcular as posições relativas é essencial para que o robô possa desempenhar todas as suas tarefas com êxito. No ROS o `tf` é a ferramenta que torna isso possível. Inúmeras transformações são publicadas o tempo todo, formando uma árvore chamada de TF Tree. O nó `amcl`, por exemplo, que é responsável pela localização do robô, publica a transformação do mapa para o `odom` de cada robô, que é uma mensagem contendo a posição, orientação, velocidade linear e angular e aceleração linear e angular, ou seja, o estado atual do robô.

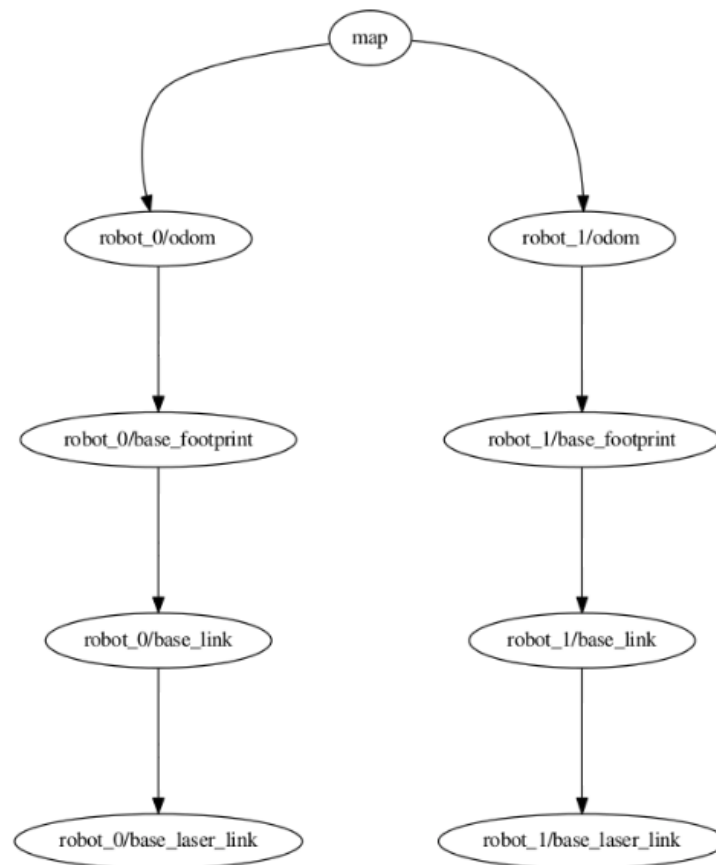


Figura 3 – Exemplo de uma TF tree

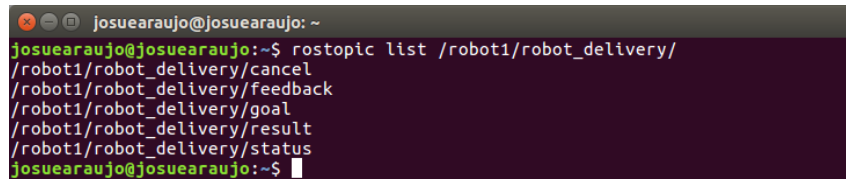
Fonte – [2, Editada pelo autor]

De forma análoga ao namespace (subseção 2.3.5), também há a ideia de um espaço de nomes para cada robô na árvore tf, ele é denominado de `tf_prefix`. Na Figura 3 é possível perceber cada robô com seu prefixo, tendo suas transformações partindo do `map`.

2.4 Actions

Em qualquer sistema baseado em ROS, existirão casos onde um nó enviará uma solicitação para outro e também receberá a resposta do mesmo. Esse é exatamente o funcionamento do ROS Service [18]. Contudo, o ROS disponibiliza uma ferramenta ainda mais poderosa, que são as actions [17].

Ao inicializar uma action, cinco tópicos com nomes pré-definidos são criados: *Goal*, *Feedback*, *Result*, *Status* e *Cancel*. Em um action, além de poder solicitar o pedido através do tópico *Goal* e receber a resposta através do tópico *Result*, quando a solicitação demora algum tempo, o nó que fez a solicitação pode cancelar o pedido através do tópico *Cancel*. Também é possível saber o andamento do pedido através do tópico *Feedback*. Por último, é possível saber o status do action server através do tópico *Status*.



```
josuearaujo@josuearaujo: ~  
josuearaujo@josuearaujo:~$ rostopic list /robot1/robot_delivery/  
/robot1/robot_delivery/cancel  
/robot1/robot_delivery/feedback  
/robot1/robot_delivery/goal  
/robot1/robot_delivery/result  
/robot1/robot_delivery/status  
josuearaujo@josuearaujo:~$
```

Figura 4 – *Print screen* do terminal listando a action

Fonte – Elaborada pelo autor.

A Figura 4 apresenta uma print screen do terminal executando o comando *rostopic list /robot1/robot_delivery/*. Esse comando retorna uma lista com todos os tópicos que estão sendo utilizados no momento no ROS. Ao adicionar o parametro */robot1/robot_delivery/*, só serão listados tópicos que estiverem dentro desse caminho. Essa action foi implementada nesse trabalho, como pode ser visto na seção 4.4. Ao iniciar o action server, os cinco tópicos referentes a ele passam a ser listados no *rostopic list*.

2.5 Pacotes utilizados

Como já foi bastante frisado, o ROS abstrai muitos conceitos, disponibilizando pacotes a serem utilizados pelos desenvolvedores. Nesse trabalho foram utilizados alguns desses pacotes, dentre eles o AMCL e o *move_base*.

2.5.1 AMCL

O AMCL *Adaptive Monte Carlo Localization* é um algoritmo utilizado por robôs para se localizar [14]. Diferente do SLAM *Simultaneous Localization and Mapping*, que mapeia uma área e se localiza ao mesmo tempo, ele apenas se localiza em um ambiente que já foi previamente mapeado utilizando, por exemplo, o gmapping. Para realizar essa tarefa, o algoritmo utiliza apenas o scan laser.

2.5.2 *move_base*

O pacote *move_base* fornece uma implementação de uma action que, ao receber uma mensagem através do tópico *goal*, irá tentar levar o robô até esse destino [16]. Essa mensagem é de um tipo específico *MoveBaseGoal*, sendo composta por informações da pose (posição e orientação) que o robô deverá assumir e em qual sistema de coordenadas ele deverá se referenciar.

Para conseguir chegar até o destino, ele utiliza dois planejadores, global e local, que estão diretamente relacionados à dois mapas de custos, também chamados de global e local. Esses mapas de custos trazem informações dos obstáculos, por isso são fundamentais para o cálculo da rota. O mapa de custos global é obtido através dos obstáculos já previstos no

mapa, já o mapa de custos local é obtido através da leitura do scan laser. A rota utilizada pelo robô é determinada da seguinte forma: um caminho global é calculado pelo planejador global baseado no mapa de custos global e então, enquanto o robô vai se movimentando, o planejador local recalcula o tempo todo a rota baseado nos obstáculos vistos no mapa de custos local. O planejador local leva em consideração alguns fatores para determinar o novo caminho a ser seguido, dentre eles estão a proximidade com os obstáculos, com o destino e a com a rota global previamente calculada.

Quando, por algum motivo, o robô fica preso, o próprio `move_base` realiza uma sequência de procedimentos para tentar remover o impedimento e chegar no goal. Como pode ser visto na Figura 5.

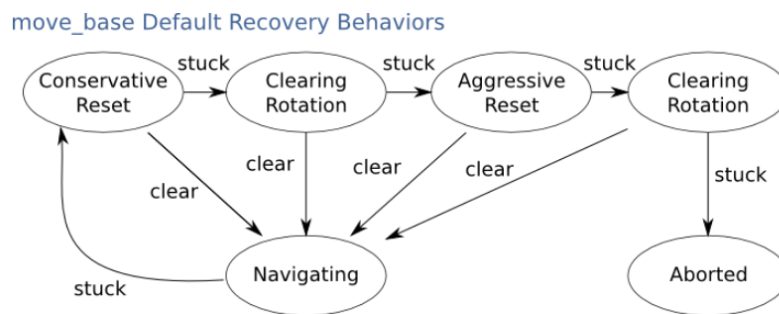


Figura 5 – Comportamento esperado do `move_base`

Fonte – [16]

3 TRABALHOS RELACIONADOS

Sistemas conhecidos como service robots (robôs de serviço) vem sendo estudados e utilizados já há bastante tempo. Os cenários são os mais diversos possíveis: ambientes domésticos, escritórios, hospitais, dentre outros.

Mesmo antes de se existir o ROS, robôs entregadores já eram desenvolvidos, como pode ser visto em [8]. Trata-se de um robô entregador que utilizava um sistema de câmeras espalhadas no ambiente, denominado de iGPS, para fornecer dados para um sistema supervisor do robô, possibilitando o cálculo de rotas. Dois testes foram realizados nesse sistema: o primeiro era se locomover do terceiro ao quarto andar de um prédio, utilizando o elevador e o segundo era um teste de duração, para verificar a confiabilidade. O robô cumpriu a tarefa de se locomover do terceiro ao quarto andar e no teste de duração que foi realizado durante três meses, sendo quatro horas por dia e quatro dias por semana, robô percorreu uma distância superior a 100km, à uma velocidade máxima de 0.4m/s.

Em [9], tem-se um exemplo de trabalho voltado para um ambiente hospitalar. Trata-se de um simulador para analisar a performance do sistema com múltiplos robôs realizando serviços de entrega em um hospital. O trabalho pressupõe que os robôs operam de modo STA *single-task-allocation*, pois trata-se de um ambiente onde não há conexão wireless em todo prédio, somente na base do robô. Com isso, cada robô precisa necessariamente voltar para a base ao finalizar uma tarefa para então receber uma nova. A análise feita traz como resultado tabelas que apresentam a quantidade de tarefas realizadas em uma determinada quantidade de tempo, o tempo médio de espera para cada tarefa e quantas tarefas deixam de ser entregadas, variando a quantidade de robôs na simulação.

Um sistema capaz de controlar robôs heterogêneos foi proposto por [5], trata-se do RoboServ. Ele traz a perspectiva de se controlar e monitorar de forma genérica qualquer robô através de serviços REST, utilizando o ROS. Esse trabalho apresenta como resultado uma simulação feita utilizando dois robôs diferentes, Turtlebot e Pioneer, sendo executados simultaneamente e controlados remotamente através de uma aplicação WEB. Nessa interface é possível escolher para qual dos robôs deseja-se enviar os comandos de teleoperação e monitorar a câmera.

O trabalho desenvolvido utiliza o ROS como base para sua implementação, utilizando as ferramentas que ele disponibiliza para realizar o serviço de robôs de entrega e permitindo que as solicitações de entrega sejam feitas através de um software cliente. Tudo sendo controlado por um software central, que é o servidor dessa aplicação, e tendo as comunicações entre as partes do programa realizadas através do ROS.

4 IMPLEMENTAÇÃO

Cada parte do sistema alocador de tarefas, bem como o passo a passo da sua implementação, estará disposto nas seções a seguir. Ele foi desenvolvido em Python utilizando como base o ROS.

4.1 Instalando pré-requisitos

O primeiro passo foi instalar o ROS e todos os pacotes adicionais que serão necessários. O sistema operacional utilizado foi o Ubuntu, na versão 16.04. Na própria Wiki do ROS há um tutorial [22] para instalar a versão Kinetic, que está sendo utilizada nesse trabalho. Algumas dependências adicionais foram necessárias. Elas podem ser obtidas executando os seguintes comandos.

```
apt install ros-kinetic-turtlebot*  
apt install ros-kinetic-joy*
```

Outro programa utilizado foi o smach-viewer, que é uma interface gráfica para visualizar as máquinas de estado utilizadas no programa. Para instalá-lo, basta acessar o github do criador [21] e pegar o código fonte. Trata-se de um pacote do ROS, ele precisa ser colocado em um catkin workspace. Conhecer esse workspace é fundamental para a implementar sistemas para o ROS, pois é através deles que o programa implementado estará acessível aos comandos ROS. O tutorial de como configurar um workspace também está na Wiki do ROS [15]. Essa ferramenta exige uma dependência em uma versão mais antiga do que a que é instalada por padrão atualmente. Para obtê-la basta executar os seguintes comandos:

```
add-apt-repository ppa:nilarimogard/webupd8  
apt-get update  
apt-get install python-wxgtk2.8
```

4.2 Configuração do ambiente

Para inicializar os nós ROS que compõem o ambiente onde o software desenvolvido será executado, faz-se necessário configurar os parâmetros de inicialização dos mesmos. Os pacotes utilizados, explicados anteriormente, precisam ter uma série de configurações iniciais estabelecidas, como por exemplo, posição inicial dos robôs, mapa a ser utilizado e variáveis dos sensores. Alguns nós iniciados pelos launch files são genéricos, ou seja,

servem para os dois robôs da simulação, como é o exemplo do map server. Já outros nós são específicos de cada robô, como por exemplo, o AMCL, tendo em vista que cada robô necessita ter a sua localização calculada no ambiente.

Para que o ambiente funcione corretamente com múltiplos robôs, cada um deles deve ter o seu próprio namespace e seu prefixo na árvore tf, para que o AMCL possa calcular a transformação corretamente. Sendo assim, os nós que são específicos de cada robô, estarão sob um namespace, como por exemplo: `/robot1/amcl`. De forma análoga, na árvore de transformação tf, cada robô terá o seu próprio tf_prefix.

4.2.1 Entendendo os arquivos launch

Os arquivos `.launch` utilizados para iniciar todos os nós necessários para o funcionamento do sistema, estão disponíveis nos Apêndices de A a E. Uma breve explicação do que cada um está fazendo será dada para facilitar o entendimento.

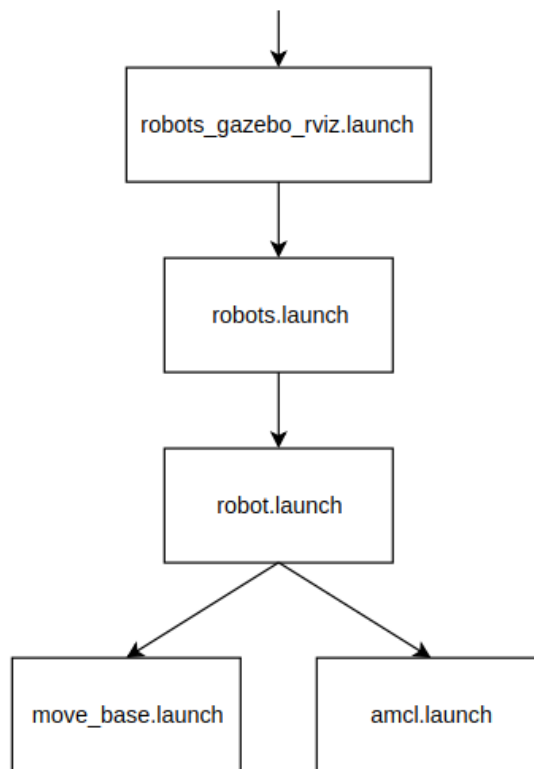


Figura 6 – Estrutura de organização dos arquivos launch

Fonte – Elaborada pelo autor

Como pode ser visto na figura Figura 6, o arquivo principal, ou seja, que inclui os demais, é o `robots_gazebo_rviz.launch`. Ele é inicializado utilizando o comando `roslaunch` no terminal. Ele inclui o arquivo `robots.launch`, que por sua vez inclui o `robot.launch` e assim sucessivamente. Essas inclusões são feitas através da tag `<include/>`, como por

exemplo na linha 28 do código do Apêndice A, e elas são usadas para diminuir o tamanho dos arquivos e facilitar o debug de possíveis erros nos parâmetros. Além disso, os arquivos estão estrategicamente separados por funções específicas.

- `robots_gazebo_rviz.launch` (Apêndice A)

É nesse arquivo onde os nós globais são instanciados, como por exemplo o `map_server`. Só é necessário um `map_server` para ambos os robôs.

- `robots.launch` (Apêndice B)

Nenhum nó é iniciado nesse arquivo, ele serve apenas para organizar os namespaces dos robôs. Caso queira instanciar outro robô na simulação, basta replicar esse trecho de código, mudando o namespace e a posição inicial do robô.

- `robot.launch` (Apêndice C)

Nesse arquivo são instanciados todos os nós inerentes a cada um dos robôs. Todo o nó iniciado nesse arquivo ou nos que forem incluídos por ele, estarão sob o namespace de cada um dos robôs, tendo em vista a configuração que foi realizada no arquivo *robots.launch*.

- `move_base.launch` (Apêndice D)

Uma série de parâmetros do `move_base` de cada um dos robôs são definidos através desse arquivo, onde acontece a inicialização dos nós do `move_base` para cada um dos robôs.

- `amcl.launch` (Apêndice E)

Uma série de parâmetros necessários para a inicialização correta do `amcl` são definidos nesse arquivo, dentre eles estão os prefixos dos robôs que serão utilizados na árvore de transformação `tf`.

4.3 Arquitetura do software

O programa está dividido em três partes:

- Cliente

Responsável por realizar os pedidos de café. Trata-se de enviar para o servidor coordenadas no mapa onde o robô deverá realizar a entrega. Essas coordenadas estão previamente definidas em uma lista de posições válidas e o software aleatoriamente seleciona um elemento dessa lista. O software implementado apenas desempenha uma função semelhante do que seria o software real do cliente, provavelmente uma aplicação web.

- Servidor

Trata-se de um produtor/consumidor. Em uma thread, o servidor ouve as solicitações advindas do cliente e adiciona-as em uma fila. Em outra thread, o programa consome essa fila, sempre que houver um robô disponível para receber um novo pedido de entrega.

- Robôs

Responsável por receber um pedido de entrega do servidor, executá-lo e retornar o resultado para o servidor. Uma ênfase na sua implementação é dada na seção 4.5.

A Figura 7 mostra o diagrama do software, onde se pode ver essas três partes. A comunicação entre o cliente e o servidor foi implementada utilizando tópicos. Já a comunicação entre o servidor e as máquinas de estados, acontece através de actions. O código fonte de cada uma das partes do sistema pode ser encontrado no github do autor [1].

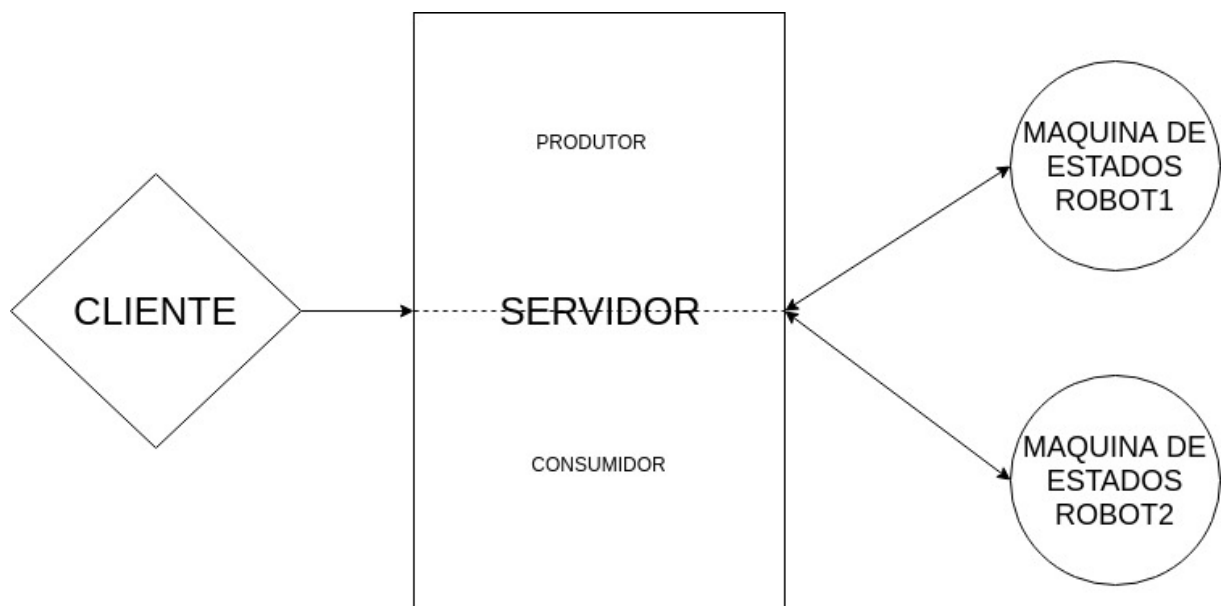


Figura 7 – Diagrama da arquitetura do software

Fonte – Elaborada pelo autor.

4.4 Robot Delivery Action

Como foi dito anteriormente, a comunicação entre o servidor e os robôs é feita através de uma action (RobotDeliveryAction), isto é, o servidor envia um goal (RobotDeliveryGoal) para os robôs e este, após efetuar a entrega, envia uma resposta (RobotDeliveryResult) de

volta para o servidor. Essas mensagens são definidas em um arquivo *.action* e precisam ser compiladas pelo `catkin_make` para serem utilizadas.

```
1 # This is an action definition file, which has three parts
2 # Part 1: the goal.
3 #
4 # The coordinates x and y, where the robot should delivery the coffe
5 float64 x
6 float64 y
7 ---
8 # Part 2: the result, sent by the robot upon completion
9 #
10 # The number of the robot who executed this action and the status of the
    requested delivery.
11 int32 robot
12 int32 status
13 ---
14 # Part 3: the feedback, to be sent periodically by action server (robot)
15 # We don't have any feedback for this action
```

Como pode ser visto acima, é nesse arquivo onde as três mensagens podem ser definidas. No caso dessa action implementada, não há mensagem de feedback, pois o servidor também não trata esse feedback. Já as mensagens de goal e result estão definidas e podem ser facilmente compreendidas.

4.5 Máquina de estados

Segundo a wiki do ROS [20], o SMACH é recomendado quando se pretende que o robô desempenhe tarefas complexas, nas quais os estados e suas possíveis transições possam ser descritos explicitamente. Ele proporciona uma rápida prototipagem, com uma sintaxe simples, agilizando o processo de implementação. Facilita a depuração do código, pois os estados da aplicação estão bem definidos, bem como permite a utilização do smach viewer, possibilitando a visualização da máquina de estados em uma interface gráfica.

Na Figura 8 pode-se ver como a máquina de estados do robô foi definida. São 4 estados:

- COZINHA (estado inicial)

O robô aguarda receber um `RobotDeliveryGoal` e, quando recebe, ele prepara o `MoveBaseGoal` que será passado para o próximo estado da máquina.

- REALIZANDO_ENTREGA

Após receber o objeto do tipo `MoveBaseGoal`, ele o envia para o action server. E então ele aguarda o `MoveBaseResult`. Se o resultado for *SUCCEEDED*, quer dizer que ele chegou no destino, então a saída do estado atual resultará na transição para o estado *DESTINO*. Se o resultado for diferente de *SUCCEEDED*, algo deu errado durante a entrega, então o robô é enviado de volta para a cozinha através da segunda saída possível para o estado atual, que o levará para o estado *RETORNANDO_COZINHA*.

- DESTINO

Nesse estado o robô está no seu destino, aguardando o cliente pegar o café que estará na bandeja do robô, contudo, por se tratar de uma simulação, nesse estado apenas está implementado uma pausa de cinco segundos, representando o tempo em que o usuário pegaria o café e mandaria o robô voltar.

- RETORNANDO_COZINHA

Nesse estado o robô tem a tarefa de voltar para a cozinha, que é a sua base. Há duas saídas possíveis para esse estado. Uma delas é quando o `MoveBaseResult` retorna *SUCCEEDED*, ou seja, ele chegou na cozinha e então a máquina de estados seguirá para o estado *COZINHA* onde o ciclo será reiniciado. Contudo, caso o `MoveBaseResult` retorne algo diferente de *SUCCEEDED*, indica que algo aconteceu durante o trajeto e o robô não conseguiu chegar na cozinha, então o robô deverá repetir o estado *RETORNANDO_COZINHA*.

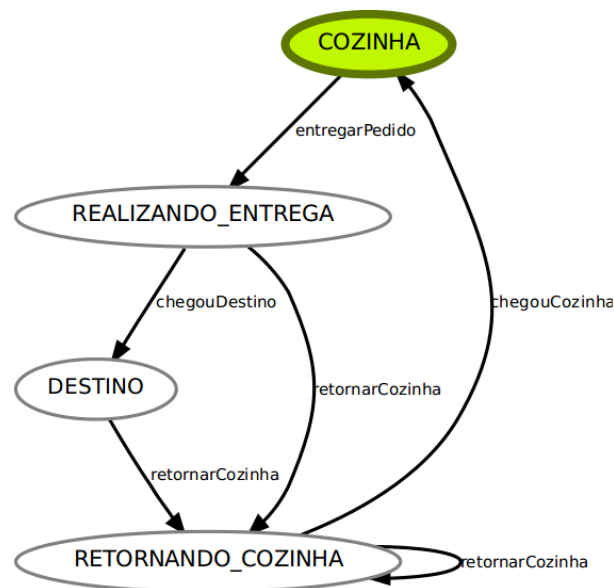


Figura 8 – *Print screen* do smach-viewer mostrando a máquina de estados de um dos robôs

5 EXPERIMENTOS

Nesse capítulo será exposto a forma como foram realizados os testes, os arquivos de configuração do ambiente de simulação, além dos links para visualização do sistema em execução. Também estarão dispostos nesse capítulo eventuais problemas e limitações desse trabalho.

5.1 Ambiente

O sistema foi testado utilizando um ambiente simulado e, para isso, dois softwares foram fundamentais: o Gazebo e o Rviz.

5.1.1 Gazebo

Para realizar os experimentos, foi utilizado o simulador de robótica 3D Gazebo. Nele é possível criar um ambiente da forma como desejar: paredes, objetos e etc. Os testes foram realizados utilizando um ambiente de competição criado pela Open Robotics em conjunto com a Hitachi [10]. O ambiente de simulação completo é iniciado através do arquivo *servicesim.world*, contudo, há uma versão reduzida desse ambiente, que pode ser inicializada através do arquivo *world floorplan.world*. Tendo em vista que o próprio pacote do *servicesim* já disponibiliza o mapa dessa versão reduzida e que o intuito desse trabalho não é mapear uma área, essa versão é considerada a mais adequada e por isso, foi utilizada neste trabalho.

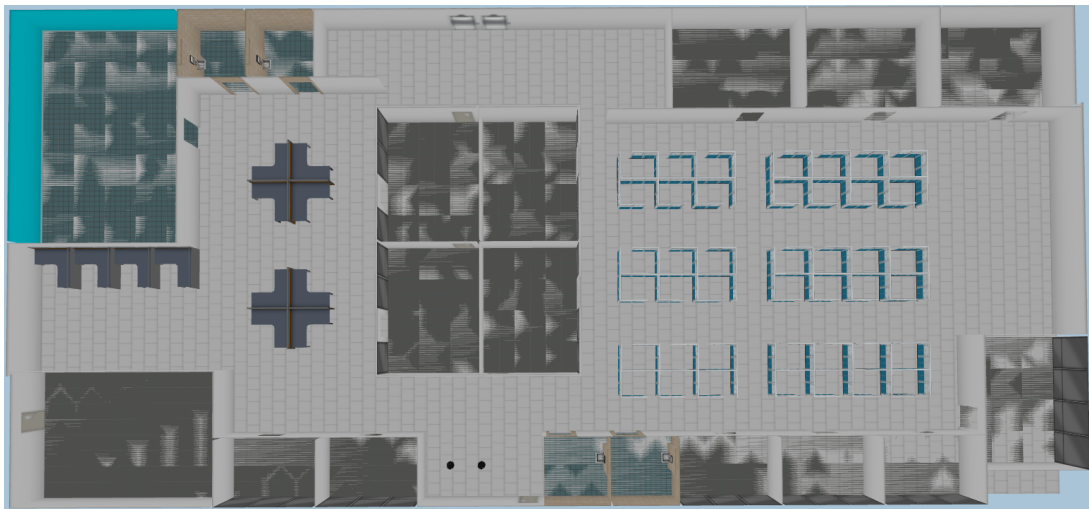


Figura 9 – *Print screen* do Gazebo apresentando o *floorplan.world*

Fonte – Elaborada pelo autor

A Figura 9 mostra o ambiente que foi utilizado nas simulações. Trata-se de um ambiente de escritório com várias salas e estações de trabalho, as quais serão os destinos do robô entregador de café. Na parte de baixo da imagem é possível visualizar os robôs turtlebots.

5.1.2 Rviz

O rviz foi outra ferramenta utilizada para obter visualizações 3D, contudo, diferente do Gazebo que apresenta uma simulação do mundo real, o Rviz permite enxergar a perspectiva do robô: os sensores, localização, mapa, imagens de câmera, tf, nuvem de pontos, dentre outras possibilidades. Como foi dito anteriormente, o pacote servicesim já disponibiliza um mapa para esse ambiente de simulação utilizado. Então um nó `map_server` foi inicializado com o mesmo.

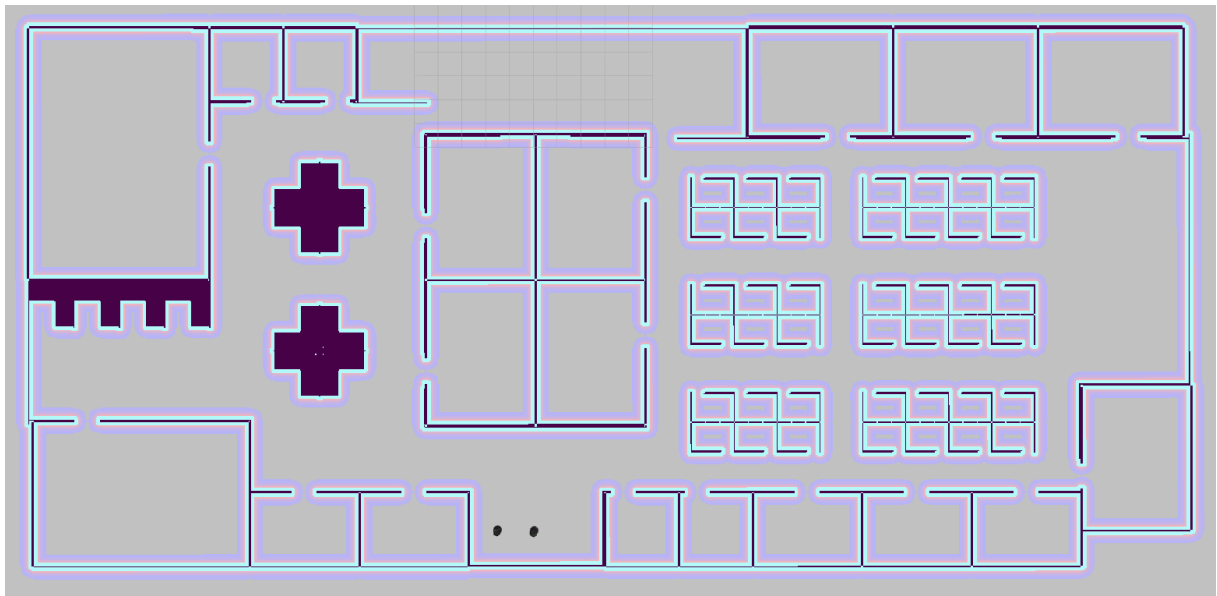


Figura 10 – *Print screen* do rviz apresentando o mapa do ambiente

Fonte – Elaborada pelo autor

A Figura 10 mostra o mapa disponibilizado junto com o pacote do servicesim. Assim como na Figura 9, também pode-se ver os robôs na parte de baixo da imagem.

5.2 Simulações

As simulações foram realizadas utilizando 30 possíveis pontos para entrega de café. Essas coordenadas foram coletadas manualmente. Para tornar a simulação mais realista, as coordenadas escolhidas foram as salas de escritório e as estações de trabalho onde supostamente o café deverá ser entregue.

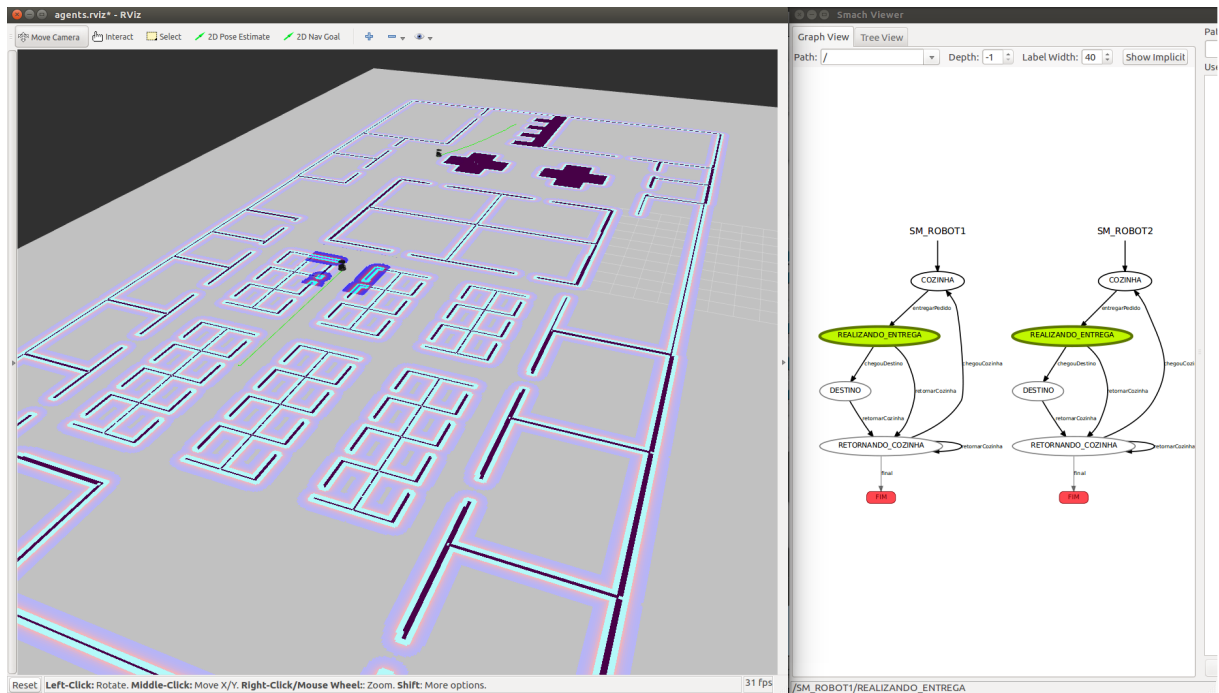


Figura 11 – *Print screen* do rviz e do smach-viewer mostrando os robôs em rota de entrega

Fonte – Elaborada pelo autor

A Figura 11 foi tirada durante uma das simulações. Como pode ser visto no smach-viewer, os robôs estão realizando entrega. A rota calculada pelo move_base para realizar a entrega pode ser vista no rviz, trata-se da linha verde que há para cada robô.

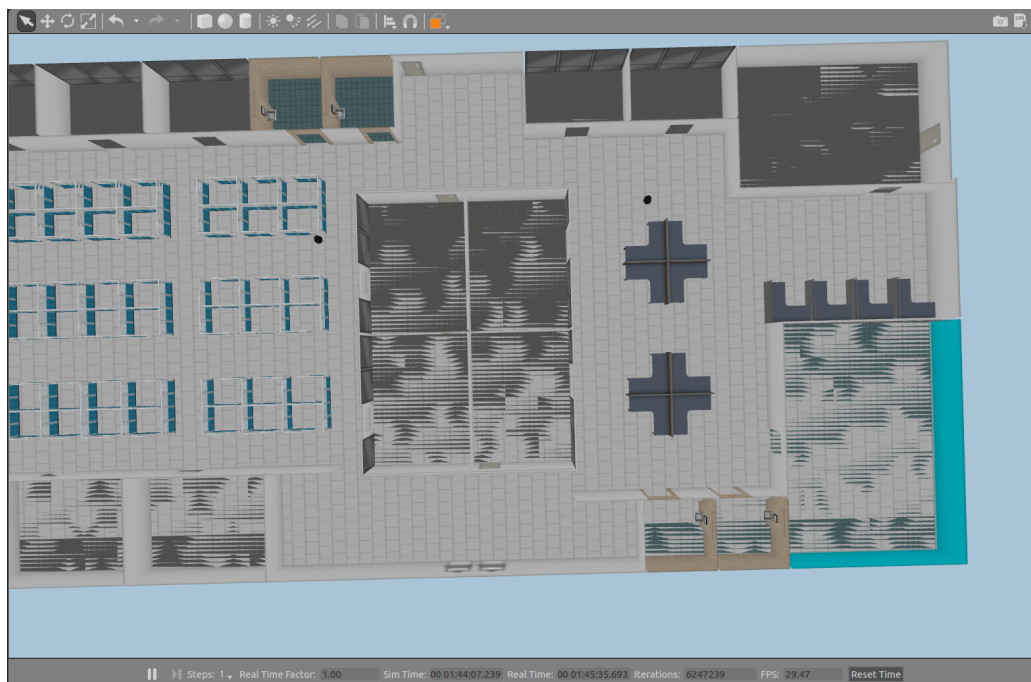


Figura 12 – *Print screen* do gazebo mostrando os robôs em rota de entrega

Fonte – Elaborada pelo autor

A Figura 12 foi tirado no mesmo instante da Figura 11. Ela apresenta a perspectiva do gazebo dos robôs entregando café.

Algumas simulações podem ser vistas nos seguintes links:

- Visualização do gazebo, rviz, smach-viewer, cliente e servidor

<https://www.youtube.com/watch?v=WKHKqFszIp4>

- Visualização do gazebo, rviz, cliente, servidor e máquinas de estados

https://www.youtube.com/watch?v=N_OyoeiHZF8

De acordo com as simulações realizadas, uma média de 48 cafés são entregues por hora, utilizando dois robôs entregadores.

5.3 Eventuais problemas e limitações

5.3.1 Comunicação

Como já foi citado, a comunicação entre o cliente e o servidor foi feito utilizando um tópico. Nas simulações, isso não trouxe nenhum problema, contudo, em um ambiente real, podendo haver limitações no sinal da rede, programas sendo executados em máquinas diferentes, esse modo de comunicação poderá trazer problemas. Por se tratar de uma comunicação baseada no protocolo UDP, não há garantias que as mensagens irão chegar no servidor, com isso, o cliente não terá garantia alguma que o seu café será entregue. O ideal seria implementar essa comunicação da mesma forma que foi feita entre o servidor e os robôs: utilizando actions, que aplicam o protocolo TCP/IP.

5.3.2 Colisão

Em alguns instantes específicos das simulações, onde um dos robôs está indo em uma direção e o outro está voltando na direção oposta, dependendo da posição onde eles se cruzem, se tiverem pouco espaço para desviar, ou estiverem fazendo uma curva, eles podem apresentar dificuldades para ultrapassar um ao outro. Utilizando um mapa mais aberto, com corredores mais espaçosos, era notório que essa passagem era mais fácil. Uma solução seria forçar, enviando um mapa específico para cada robô, que cada um deles utilizasse rotas específicas. Por exemplo, se cada um dos robôs utilizasse um lado do corredor para calcular as suas rotas, isso impediria que ambos trafegassem exatamente pela mesma rota, removendo completamente a chance de haver colisões.

5.3.3 Otimização

Outra limitação desse trabalho é que não há uma otimização nas rotas. Por exemplo, cada robô poderia sair para entregar mais de um café ao mesmo tempo, levando em consideração as coordenadas de entrega dos cafés que estão na fila, otimizando assim o tempo de entrega.

5.4 Robôs reais

A intenção inicial do trabalho era de, além das simulações, realizar testes reais utilizando como cenário o DCA da UFRN, o que não foi possível por falta de tempo. Contudo, poucas adaptações precisariam ser feitas para realizar tais testes. A primeira mudança está relacionada ao mapa: ao invés de utilizar o mapa do servicesim, seria necessário apontar para o mapa do DCA, essa configuração é feita no launch file, no nó `map_server`. Além disso, as poses iniciais de cada robô também precisariam ser alteradas, tendo em vista que elas dependem do mapa utilizado. Esses valores deveriam ser alterados no launch de cada robô, no nó do AMCL e também na máquina de estados de cada robô, pois as coordenadas de onde seria a cozinha, também teria mudado. Além disso, no programa do cliente, há uma lista de possíveis locais onde o café pode ser entregue. Sendo assim, ao mudar o mapa para o do DCA, essa lista de coordenadas também deve ser alterada.

6 CONCLUSÃO

Implementar um sistema como esse que foi apresentado nesse trabalho, à primeira vista, demonstraria demandar uma quantidade de esforço considerável, tendo em vista a necessidade de ter o mapa de uma área, localizar o robô nesse ambiente, calcular rotas, comunicação entre softwares, dentre outras minúcias que estão envolvidas nesse processo. No entanto, ao se estudar o ROS e descobrir o nível de abstração que ele traz, a tarefa se torna imensuravelmente mais fácil.

O trabalho foi desenvolvido com êxito, como apresentado no Capítulo 5, os robôs foram postos em simulação durante várias horas seguidas e entregando café aleatoriamente em trinta destinos possíveis. Mesmo em alguns determinados momentos os robôs tendo algumas dificuldades de cruzarem um ao outro, como foi citado na subseção 5.3.2, os robôs em quase 100% das vezes conseguiam se localizar novamente no mapa e seguir seus trajetos. Contudo, vale ressaltar o que foi posto nessa mesma subseção: uma possível melhoria poderia ser feita se fossem levantadas as localizações críticas do mapa onde os robôs encontram mais dificuldades para que ajustes pudessem ser feitos. Dessa forma, esse risco estaria eximido.

Esse trabalho abre inúmeras possibilidades de novos projetos, em diversas perspectivas diferentes, como por exemplo:

- Algoritmos de SLAM [6] e de Odometria Visual [23] utilizando múltiplos robôs poderiam ser pensados, tendo em vista que esse trabalho prepara um ambiente com múltiplos robôs.
- Melhorar o projeto atual, implementando uma otimização das rotas para os robôs.

REFERÊNCIAS

- [1] Josué Araújo. multi_robot_sim. https://github.com/josuearaujo/multi_robot_sim, 2019. Acessado em: 2019-11-21. 28
- [2] ROS User arttp2. Tf tree image example. <https://answers.ros.org/upfiles/14773947913085009.png>. Acessado em: 2019-11-21. 21
- [3] J. Boren and S. Cousins. Exponential growth of ROS [ROS topics]. *IEEE Robotics Automation Magazine*, 18(1):19–20, March 2011. 12
- [4] Brian Gerkey. Ros, the robot operating system, is growing faster than ever, celebrates 8 years. <https://spectrum.ieee.org/automaton/robotics/robotics-software/ros-robot-operating-system-celebrates-8-years>, 2015. 14
- [5] L. F. Costa and L. M. G. Gonçalves. Roboserv: A ROS based approach towards providing heterogeneous robots as a service. In *2016 XIII Latin American Robotics Symposium and IV Brazilian Robotics Symposium (LARS/SBR)*, pages 169–174, Oct 2016. 24
- [6] H. Durrant-Whyte and Tim Bailey. Simultaneous localization and mapping: part I. *IEEE Robotics and Automation Magazine*, 13(2):99–110, 2006. 36
- [7] Willow Garage. Willow garage. <http://www.willowgarage.com/>. Acessado em: 2019-11-20. 14
- [8] Y. Hada, H. Gakuhari, K. Takase, and E. I. Hemeldan. Delivery service robot using distributed acquisition, actuators and intelligence. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 3, pages 2997–3002 vol.3, Sep. 2004. 24
- [9] S. Jeon and J. Lee. The simulator for performance analysis of multiple robots for hospital delivery. In *2016 13th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)*, pages 746–748, Aug 2016. 12, 24
- [10] OSRF OpenSourceRoboticsFoundation. Servicesim. <https://bitbucket.org/osrf/servicesim/wiki/Home>. Acessado em: 2019-11-17. 12, 31
- [11] OSRF OpenSourceRoboticsFoundation. Turtlebot 2. <https://www.turtlebot.com/turtlebot2/>. Acessado em: 2019-11-19. 12

- [12] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, page 5, 2009. 12, 14, 17, 18
- [13] Morgan Quigley, Brian Gerkey, and William D. Smart. *Programming Robots with ROS: A Practical Introduction to the Robot Operating System*. O'Reilly Media, Inc., 1st edition, 2015. 14
- [14] ROS. Amcl. <http://wiki.ros.org/amcl>. Acessado em: 2019-11-21. 22
- [15] ROS. Catkin workspace. http://wiki.ros.org/catkin/Tutorials/create_a_workspace. Acessado em: 2019-11-15. 25
- [16] ROS. Move base. http://wiki.ros.org/move_base. Acessado em: 2019-11-21. 22, 23
- [17] ROS. Ros actions. <http://wiki.ros.org/actionlib>. Acessado em: 2019-11-21. 21
- [18] ROS. Ros services. <http://wiki.ros.org/Services>. Acessado em: 2019-11-21. 21
- [19] ROS. ROS Wiki. <http://wiki.ros.org>. Acessado em: 2019-11-20. 14
- [20] ROS. Smach. <http://wiki.ros.org/smach>. Acessado em: 2019-11-16. 29
- [21] ROS. Smach-viewer. https://github.com/ros-visualization/executive_smach_visualization. Acessado em: 2019-11-15. 25
- [22] ROS. Tutorial ros installation. <http://wiki.ros.org/kinetic/Installation/Ubuntu>. Acessado em: 2019-11-15. 25
- [23] D. Scaramuzza and F. Fraundorfer. Visual odometry [tutorial]. *IEEE Robotics Automation Magazine*, 18(4):80–92, Dec 2011. 36
- [24] Youngkak Ma, Seungwoo Kim, Dongik Oh, and Youngwan Cho. A study on development of home mess-cleanup robot mcbot. In *2008 IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, pages 114–119, July 2008. 12

APÊNDICE A

– ROBOTS_GAZEBO_RVIZ.LAUNCH

```

1 <launch>
2   <param name="/use_sim_time" value="true"/>
3   <arg name="world_file" default="$(find multi_robot_sim)/worlds/
4     floorplan.world"/>
5
6   <!-- start world -->
7   <include file="$(find gazebo_ros)/launch/empty_world.launch">
8     <env name="GAZEBO_RESOURCE_PATH" value="$(find multi_robot_sim)/
9       worlds:$(find multi_robot_sim)"/>
10    <env name="GAZEBO_MODEL_PATH" value="$(find multi_robot_sim)/models"
11      />
12    <arg name="world_name" value="$(arg world_file)" />
13    <arg name="paused" value="false"/>
14    <arg name="use_sim_time" value="true"/>
15    <arg name="gui" value="true"/>
16    <arg name="verbose" value="true"/>
17  </include>
18
19  <!-- Map server -->
20  <arg name="map_file" default="$(find multi_robot_sim)/maps/servicesim.
21    yaml"/>
22  <node name="map_server" pkg="map_server" type="map_server" args="$(arg
23    map_file)" >
24    <param name="frame_id" value="/map"/>
25  </node>
26
27  <!-- Rviz -->
28  <node pkg="rviz" type="rviz" name="rviz" args="-d $(find
29    multi_robot_sim)/rviz/agents.rviz"
30    output="screen" />
31
32  <!-- include our robots -->
33  <include file="$(find multi_robot_sim)/launch/include/robots.launch.
34    xml"/>
35</launch>

```


APÊNDICE B – ROBOTS.LAUNCH

```
1 <launch>
2   <!-- BEGIN ROBOT 1-->
3   <group ns="robot1">
4     <include file="$(find multi_robot_sim)/launch/include/robot.launch.
      xml" >
5       <arg name="initial_pose_x" value="0" />
6       <arg name="initial_pose_y" value="21" />
7       <arg name="initial_pose_z" value="0" />
8       <arg name="initial_pose_yaw" value="3.1415926" />
9       <arg name="robot_name" value="robot1" />
10    </include>
11  </group>
12
13  <!-- BEGIN ROBOT 2-->
14  <group ns="robot2">
15    <include file="$(find multi_robot_sim)/launch/include/robot.launch.
      xml" >
16      <arg name="initial_pose_x" value="1.5" />
17      <arg name="initial_pose_y" value="21" />
18      <arg name="initial_pose_z" value="0" />
19      <arg name="initial_pose_yaw" value="0" />
20      <arg name="robot_name" value="robot2" />
21    </include>
22  </group>
23 </launch>
```

APÊNDICE C – ROBOT.LAUNCH

```

1 <launch>
2   <arg name="initial_pose_x"/>
3   <arg name="initial_pose_y"/>
4   <arg name="initial_pose_z"/>
5   <arg name="initial_pose_yaw"/>
6   <arg name="robot_name"/>
7
8   <arg name="init_pose" value="-x $(arg initial_pose_x) -y $(arg
   initial_pose_y) -z $(arg initial_pose_z) -Y $(arg initial_pose_yaw)"
   />
9
10  <arg name="base" value="$(optenv TURTLEBOT_BASE kobuki)"/> <!--kobuki(
   hexagons) create(circles), roomba -->
11  <arg name="battery" value="$(optenv TURTLEBOT_BATTERY /proc/acpi/
   battery/BAT0)"/> <!-- /proc/acpi/battery/BAT0 -->
12  <arg name="stacks" value="$(optenv TURTLEBOT_STACKS hexagons)"/> <!--
   circles, hexagons -->
13  <arg name="3d_sensor" value="$(optenv TURTLEBOT_3D_SENSOR kinect)"/>
   <!-- kinect, asus_xtion_pro -->
14
15  <arg name="urdf_file" default="$(find xacro)/xacro.py '$(find
   turtlebot_description)/robots/$(arg base)_$(arg stacks)_$(arg 3
   d_sensor).urdf.xacro' prefix=$(arg robot_name)" />
16  <param name="robot_description" command="$(arg urdf_file)" />
17
18  <!-- Gazebo model spawner -->
19  <node name="spawn_turtlebot_model" pkg="gazebo_ros" type="spawn_model"
   args="$(arg init_pose) -unpause -urdf -param robot_description -
   model $(arg robot_name)" respawn="false" output="screen"/>
20
21  <!-- Velocity mux nodelet -->
22  <node pkg="nodelet" type="nodelet" name="mobile_base_nodelet_manager"
   args="manager"/>
23  <node pkg="nodelet" type="nodelet" name="cmd_vel_mux"
24     args="load yocs_cmd_vel_mux/CmdVelMuxNodelet
   mobile_base_nodelet_manager">
25     <param name="yaml_cfg_file" value="$(find turtlebot_bringup)/param/
   mux.yaml" />
26     <remap from="cmd_vel_mux/output" to="mobile_base/commands/velocity"
   />
27 </node>

```

```

28
29 <!-- bumper2pc.launch.xml -->
30 <node pkg="nodelet" type="nodelet" name="bumper2pointcloud" args="load
    kobuki_bumper2pc/Bumper2PcNodelet nodelet_manager">
31   <param name="pointcloud_radius" value="0.24"/>
32   <remap from="bumper2pointcloud/pointcloud" to="sensors/
    bumper_pointcloud"/>
33   <remap from="bumper2pointcloud/core_sensors" to="sensors/core"/>
34 </node>
35
36 <!-- Robot State Publisher -->
37 <node pkg="robot_state_publisher" type="robot_state_publisher" name="
    robot_state_publisher">
38   <param name="publish_frequency" type="double" value="30.0" />
39   <param name="tf_prefix" type="string" value="$(arg robot_name)" />
40 </node>
41
42 <arg name="min_range" default="0.45" />
43 <!-- fake laser -->
44 <node pkg="nodelet" type="nodelet" name="laserscan_nodelet_manager"
    args="manager"/>
45 <node pkg="nodelet" type="nodelet" name="depthimage_to_laserscan"
46   args="load depthimage_to_laserscan/DepthImageToLaserScanNodelet
    laserscan_nodelet_manager">
47   <param name="scan_height" value="3"/>
48   <param name="output_frame_id" value="$(arg robot_name)/
    camera_depth_frame"/>
49   <param name="range_min" value="$(arg min_range)"/>
50   <remap from="image" to="camera/depth/image_raw"/>
51   <remap from="scan" to="scan"/>
52 </node>
53
54 <!-- Velocity smoother -->
55 <node pkg="nodelet" type="nodelet" name="navigation_velocity_smoother"
56   args="load yocs_velocity_smoother/VelocitySmootherNodelet
    mobile_base_nodelet_manager">
57   <rosparam file="$(find turtlebot_bringup)/param/defaults/smooth
    er.yaml" command="load"/>
58   <remap from="navigation_velocity_smoother/smooth_cmd_vel" to="
    cmd_vel_mux/input/navi"/>
59   <remap from="navigation_velocity_smoother/odometry" to="odom"/>
60   <remap from="navigation_velocity_smoother/robot_cmd_vel" to="
    mobile_base/commands/velocity"/>
61 </node>
62
63 <!-- Safety controller -->
64 <node pkg="nodelet" type="nodelet" name="kobuki_safety_controller"

```

```

65     args="load kobuki_safety_controller/SafetyControllerNodelet
        nodelet_manager">
66     <remap from="kobuki_safety_controller/cmd_vel" to="cmd_vel_mux/input
        /safety_controller"/>
67     <remap from="kobuki_safety_controller/events/bumper" to="events/
        bumper"/>
68     <remap from="kobuki_safety_controller/events/cliff" to="events/cliff
        "/>
69     <remap from="kobuki_safety_controller/events/wheel_drop" to="events/
        wheel_drop"/>
70 </node>
71
72 <!-- The odometry estimator, throttling, fake laser etc. go here -->
73 <!-- All the stuff as from usual robot launch file -->
74 <include file="$(find multi_robot_sim)/launch/include/move_base.launch
        .xml" >
75     <arg name="robot_name" value="$(arg robot_name)" />
76     <arg name="init_pose" value="$(arg init_pose)" />
77 </include>
78
79 <!--AMCL-->
80 <include file="$(find multi_robot_sim)/launch/include/amcl.launch.xml"
        >
81     <arg name="initial_pose_x" value="$(arg initial_pose_x)"/>
82     <arg name="initial_pose_y" value="$(arg initial_pose_y)"/>
83     <arg name="initial_pose_a" value="$(arg initial_pose_yaw)"/>
84     <arg name="odom_frame_id" value="/$(arg robot_name)/odom"/>
85     <arg name="base_frame_id" value="/$(arg robot_name)/base_footprint"
        />
86     <arg name="scan_topic" value="/$(arg robot_name)/scan"/>
87 </include>
88
89 <!--Load agent parameters-->
90 <rosparam file="$(find multi_robot_sim)/param/agent_params.yaml"
        command="load"/>
91 </launch>

```

APÊNDICE D – MOVE_BASE.LAUNCH

```

1 <launch>
2   <arg name="robot_name"/>
3   <arg name="init_pose"/>
4
5   <!-- Define your move_base node -->
6   <node pkg="move_base" type="move_base" respawn="false" name="move_base
7     " output="screen">
8     <rosparam file="$(find multi_robot_sim)/param/costmap_common_params.
9       yaml" command="load" ns="global_costmap" />
10    <rosparam file="$(find multi_robot_sim)/param/costmap_common_params.
11      yaml" command="load" ns="local_costmap" />
12    <rosparam file="$(find multi_robot_sim)/param/local_costmap_params.
13      yaml" command="load" />
14    <rosparam file="$(find multi_robot_sim)/param/global_costmap_params.
15      yaml" command="load" />
16    <rosparam file="$(find multi_robot_sim)/param/move_base_params.yaml"
17      command="load" />
18    <rosparam file="$(find multi_robot_sim)/param/
19      base_local_planner_params.yaml" command="load"/>
20    <param name="local_costmap/robot_base_frame" value="/$(arg
21      robot_name)/base_footprint"/>
22    <param name="global_costmap/robot_base_frame" value="/$(arg
23      robot_name)/base_footprint"/>
24    <param name="local_costmap/point_cloud_sensor/topic" value="/$(arg
25      robot_name)/camera/depth/points"/>
26    <param name="global_costmap/point_cloud_sensor/topic" value="/$(arg
27      robot_name)/camera/depth/points"/>
28    <param name="local_costmap/point_cloud_sensor/sensor_frame" value="/
29      $(arg robot_name)/camera_depth_frame"/>
30    <param name="global_costmap/point_cloud_sensor/sensor_frame" value="
31      /$(arg robot_name)/camera_depth_frame"/>
32    <param name="local_costmap/scan/sensor_frame" value="/$(arg
33      robot_name)/camera_depth_frame"/>
34    <param name="global_costmap/scan/sensor_frame" value="/$(arg
35      robot_name)/camera_depth_frame"/>
36    <param name="local_costmap/scan/topic" value="/$(arg robot_name)/
37      scan"/>
38    <param name="global_costmap/scan/topic" value="/$(arg robot_name)/
39      scan"/>
40    <remap from="odom" to="odom"/>
41    <remap from="map" to="/map"/>

```

```
25     <remap from="cmd_vel" to="navigation_velocity_smoother/raw_cmd_vel"
    />
26   </node>
27 </launch>
```

APÊNDICE E – AMCL.LAUNCH

```

1 <launch>
2   <arg name="use_map_topic" default="false"/>
3   <arg name="scan_topic" default="scan" />
4   <arg name="initial_pose_x" default="0.0"/>
5   <arg name="initial_pose_y" default="0.0"/>
6   <arg name="initial_pose_a" default="0.0"/>
7   <arg name="odom_frame_id" default="odom"/>
8   <arg name="base_frame_id" default="base_footprint"/>
9   <arg name="global_frame_id" default="map"/>
10
11   <node pkg="amcl" type="amcl" name="amcl">
12     <param name="use_map_topic" value="$(arg use_map_topic)"/>
13     <!-- Publish scans from best pose at a max of 10 Hz -->
14     <param name="odom_model_type" value="diff"/>
15     <param name="odom_alpha5" value="0.1"/>
16     <param name="gui_publish_rate" value="10.0"/>
17     <param name="laser_max_beams" value="60"/>
18     <param name="laser_max_range" value="12.0"/>
19     <param name="min_particles" value="500"/>
20     <param name="max_particles" value="2000"/>
21     <param name="kld_err" value="0.05"/>
22     <param name="kld_z" value="0.99"/>
23     <param name="odom_alpha1" value="0.2"/>
24     <param name="odom_alpha2" value="0.2"/>
25     <!-- translation std dev, m -->
26     <param name="odom_alpha3" value="0.2"/>
27     <param name="odom_alpha4" value="0.2"/>
28     <param name="laser_z_hit" value="0.5"/>
29     <param name="laser_z_short" value="0.05"/>
30     <param name="laser_z_max" value="0.05"/>
31     <param name="laser_z_rand" value="0.5"/>
32     <param name="laser_sigma_hit" value="0.2"/>
33     <param name="laser_lambda_short" value="0.1"/>
34     <param name="laser_model_type" value="likelihood_field"/>
35     <!-- <param name="laser_model_type" value="beam"/> -->
36     <param name="laser_likelihood_max_dist" value="2.0"/>
37     <param name="update_min_d" value="0.25"/>
38     <param name="update_min_a" value="0.2"/>
39     <param name="odom_frame_id" value="$(arg odom_frame_id)"/>
40     <param name="base_frame_id" value="$(arg base_frame_id)"/>
41     <param name="global_frame_id" value="$(arg global_frame_id)" />

```

```
42     <param name="child_frame_id" value="$(arg base_frame_id)" />
43     <param name="resample_interval" value="1"/>
44     <!-- Increase tolerance because the computer can get quite busy -->
45     <param name="transform_tolerance" value="1.0"/>
46     <param name="recovery_alpha_slow" value="0.0"/>
47     <param name="recovery_alpha_fast" value="0.0"/>
48     <param name="initial_pose_x" value="$(arg initial_pose_x)"/>
49     <param name="initial_pose_y" value="$(arg initial_pose_y)"/>
50     <param name="initial_pose_a" value="$(arg initial_pose_a)"/>
51     <remap from="map" to="/map"/>
52     <remap from="scan" to="$(arg scan_topic)"/>
53     <remap from="static_map" to="/static_map" />
54 </node>
55 </launch>
```