UNIVERSIDADE DO ESTADO DO AMAZONAS
ESCOLA SUPERIOR DE TECNOLOGIA
ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Rafael Goulart Nogueira da Silva

SISTEMA ROBÓTICO PARA VIGILÂNCIA DE AMBIENTES INTERNOS UTILIZANDO UM ROBÔ MÓVEL AUTÔNOMO COM ACIONAMENTO DIFERENCIAL

Manaus
2015

Rafael Goulart Nogueira da Silva

SISTEMA ROBÓTICO PARA VIGILÂNCIA DE AMBIENTES INTERNOS UTILIZANDO UM ROBÔ MÓVEL AUTÔNOMO COM ACIONAMENTO DIFERENCIAL

Trabalho de Conclusão de Curso submetido à Coordenação do curso de Engenharia de Controle e Automação da Universidade do Estado do Amazonas como parte dos requisítos necessários para a obtenção do grau de Engenheiro. em conformidade com as normas ABNT.

Orientador Me. Almir Kimura Junior
Coorientador: Dr. Israel Mazaira Morales

Manaus
2015

Rafael Goulart Nogueira da Silva

SISTEMA ROBÓTICO PARA VIGILÂNCIA DE AMBIENTES INTERNOS UTILIZANDO UM ROBÔ MÓVEL AUTÔNOMO COM ACIONAMENTO DIFERENCIAL

Trabalho de Conclusão de Curso submetido à Coordenação do curso de Engenharia de Controle e Automação da Universidade do Estado do Amazonas como parte dos requisítos necessários para a obtenção do grau de Engenheiro. em conformidade com as normas ABNT.

Aprovado em de de 2015.

BANCA EXAMINADORA

Me. Almir Kimura Junior
Orientador

Dr. Walter Andres Vermehren Valenzuela
Professor

Me. Marcelo Albuquerque
Convidado 1

Marivan Gomes
Convidado 2

Manaus
2015

AGRADECIMENTOS

Primeiramente, aos meus pais, Celso e Rita, por terem me educado, ajudando na formação do meu carater, e por terem investindo o máximo que podiam em mim.

Agradeço a Isabelle, minha parceira de longa caminhada, que mesmo com minha insensibilidade em determinadas situações, não desistiu de mim e sempre acreditou no meu potencial. Ela foi, sem dúvidas, a pessoa mais importante para a realização deste trabalho, testando o sistema inúmeras vezes, e me motivando a prosseguir com o desenvolvimento desta ideia.

Ao corpo docente do curso de Engenharia de Controle e Automação da UEA, e em especial ao professor Almir Kimura Junior pela sua orientação durante esta ardua jornada chamada TCC, e que, mesmo com os problemas da vida, me guiou na elaboração e redação deste trabalho.

Agradecimentos especiais são direcionados à todos os voluntários e grande amigos que testaram o Horus Patrol: Isabelle, Celso, Lucas, Dilermando, Máximo, Eiji, Diannny, professor Kimura e professor Mazaira.

A Universidade do Estado do Amazonas, que proporcionou a oportunidade de participar desta graduação e que contribuiu para minha carreira profissional.

Por fim, agradeço a todos os brasileiros que, por meio dos impostos, investiram no Ciências sem Fronteira e este, por sua vez, permitiu que eu adquirisse os conhecimentos necessários para criar o Horus Patrol.

"Não é a escola que faz o aluno,
é o aluno que faz a escola."
(Ana Rita Arruda, Fundação Nokia de Ensino)

RESUMO

Robôs móveis vêm sendo usados cada vez mais para a execução de inúmeros serviços à sociedade. Dentre estes serviços, destaca-se seu papel em sistemas de segurança onde o robô é responsável por realizar a vigilância do ambiente visando a remoção do ser humano desta atividade considerada de risco a saúde humana, bem como otimizar a reação do sistema mediante alguma ameaça.

Assim, nesse contexto, este trabalho propõe-se a desenvolver um sistema robótico, chamado de Horus Patrol, para realizar a vigilância de um ambiente a fim de reduzir o contato humano com esta atividade de risco. Este sistema consiste em um robô móvel com acionamento diferencial, o Turtlebot 2, operando em conjunto com uma interface gráfica. O sistema realiza a vigilância do ambiente por meio de patrulhas, visando garantir que qualquer possível ameaça seja detectada. A patrulha é executada por meio de rotas pré-definidas, onde o robô é o responsável por realizá-las de forma autônoma e reportar quaisquer anomalias detectadas para uma central de comando, monitorada por um vigilante que é o responsável por filtrar falso positivos e pela tomada de decisões em caso de perigo. Este trabalho foi testado por meio de uma série de robustas simulações no simulador 3D Gazebo, onde o Horus Patrol realizou patrulhas em um ambiente virtual a fim de verificar o comportamento de suas funcionalidades. Após os testes, constatou-se que o sistema é eficaz na realização de patrulhas em ambientes simulados e que sua interface gráfica está pronta para ser aplicada em um ambiente real.

Palavras-chaves: ROS. Sistemas robóticos. Robôs móveis. Denvolvimento de GUI. Sistemas de vigilância.

ABSTRACT

Mobile robots have been increasingly used for running numerous services to society. Among these services, stands out their role in security systems where the robot is responsible for performing environmental patrolling to remove the human being of this activity, considered risky, as well as optimize the system reaction by any threat event.
Therefore, in this context, this project proposes to develop a robotic system, called Horus Patrol, to conduct surveillance of an indoor environment in order to reduce human contact with this risk activity. This system consists of a mobile differential drive robot, the Turtlebot 2, operating in conjunction with a graphical user interface. The system performs environmental monitoring through patrols in order to ensure that any possible threat is detected. The patrol is performed through pre-defined routes where the robot is responsible for carrying them out autonomously and report any abnormal findings to a central command, monitored by a security guard who is responsible for filtering false positives and for the decision-making in case of danger.
This work was tested by a series of robust simulations in the Gazebo simulator, where Horus Patrol patrolled a virtual environment in order to verify its functionalities. After testing, it was found that the system is effective in carrying out patrols in simulated environments and its graphical user interface is ready to be applied in a real environment.

Key-words:ROS. Robotic systems. Mobile robots. GUI development. Surveillance system.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Sistemas de vigilância sempre estiveram presentes na humanidade a fim de proteger bens ou pessoas, dentro de um ambiente, de possíveis perigos. Por meio desta definição, considera-se que o primeiro sistema de vigilância foi ainda na pré-história quando o homem vigiava a entrada da sua caverna para não ser surpreendido por animais ferozes ou outros humanos. Com o surgimento das primeiras sociedades, os riscos foram aumentando e, no século XVI, oficializou-se o uso do termo vigilante como profissão. Estas pessoas escolhidas por serem hábeis na luta e no uso da espada, e eram utilizados na vigilância de armazéns ou outras áreas de importância para seus empregadores, os senhores feudais.

Atualmente, muitos destes sistemas de vigilância são constituídos de dois componentes: seres humanos (vigilante ou vigias), que são responsáveis por monitorar o ambiente tomando as necessárias ações mediante a uma possível ameaça, e dispositivos eletrônicos (câmeras, sensores), que auxiliam estes vigilantes agindo como uma extensão das capacidades físicas dos mesmos. Um exemplo muito comum deste tipo de sistema é o chamado de *Closed-circuit television* (CCTV), ou circuito fechado de televisão (CFTV), consistindo de um conjunto de câmeras distribuídas em pontos de interesse de uma área onde as imagens captadas são gravadas e monitoradas em tempo real remotamente.

Com o advento da robótica, novos modelos de sistemas de vigilância começam a surgir que são mais eficientes e seguros em comparação com um sistema de vigilância convencional. Tais sistemas empregam robôs móveis para auxiliar na vigilância. Dentre as vantagens de se utilizar robôs móveis destacam-se o tempo de reação inferior durante uma situação de perigo e a redução da presença de um ser humano em uma situação de perigo, pois o mesmo é substituído por um robô, que além de proteger vidas, reduz consideravelmente a existência do erro humano durante execução do sistema.

Neste trabalho, foi desenvolvido um protótipo de um sistema de vigilância por meio de patrulhas empregando a robótica móvel, chamado de Horus Patrol, onde o objetivo é o de substituir o vigilante responsável pela execução das patrulhas do sistema, por um robô móvel, o Turtlebot 2. Este robô realiza patrulhas pré-definidas em um ambiente conhecido interno (lugar fechado por seis paredes), buscando sempre que possível presença de um intruso, reportando suas detecções em tempo real à uma central de comando, constituída por uma interface gráfica, que contém ferramentas necessárias para a gestão do sistema e permite que o vigilante interaja com o robô.

Para validar seu funcionamento e viabilidade, o Horus Patrol foi testado em um ambiente gerado pelo simulador 3D Gazebo, que utilizou um modelo virtual do Turtlebot 2.

1.1 Motivação

O uso de robôs especializados no provimento de serviços é cada vez mais corrente em vários setores de atividades humanas. Dentre estas atividades, destacam-se as tediosas, repetitivas, insalubres ou de risco para o ser humano. Dentre estes tipos de atividades, as consideradas de risco a vida são as mais importantes a serem estudadas a fim de reduzir ou remover, por meio da robótica, a presença do ser humano. Um exemplo que representa este cenário de atividade de risco é a vigilância de ambiente realizada por meio de patrulhas, onde o ser humano, o vigilante, está exposto constantemente a ameaças de intrusos. Deste modo, a robótica móvel se mostra como uma solução viável para resolver este problema, onde os robôs são os responsáveis por desempenhar o papel de vigilantes.

O desenvolvimento de um trabalho científico voltado aos conhecimentos adquiridos durante a participação no programa de intercâmbio Ciência sem Fronteiras do governo federal é uma das maneiras de se contribuir para o avanço da sociedade, que é o principal objetivo do programa.

1.2 Justificativa

O presente trabalho se justifica pela diminuição do contato humano com situações de possível risco inerentes da atividade de um vigilante, pelo aumento da eficácia na proteção do patrimônio de uma empresa, pela redução de custos com um sistema de segurança e pelo aumento da velocidade de reação mediante alguma situação de perigo.

1.3 Objetivo

Desenvolver um sistema robótico de baixo custo para monitorar um ambiente interno por meio de patrulhas. Este sistema robótico será implementado em um ambiente simulado e contará com o robô móvel Turtebot 2 e uma interface gráfica, que se comunicará com o robô via rede *wireless*, para a gestão do sistema de segurança e interação com o robô.

1.3.1 Objetivo específico

- Desenvolver um algoritmo para controlar os movimentos e os dados coletados por sensores;
- Determinar as técnicas mais compatíveis para realizar o mapeamento, localização e navegação de robôs autônomos;
- Desenvolver um algoritmo para detecção de presença de ameaças utilizando as informações oriundas dos sensores do robô móvel virtual;

- Estudar o funcionamento do sistema ROS e seus pacotes para controle de robôs e desenvolver os "nós" necessários para controlar o sistema robótico;
- Desenvolver uma interface gráfica que possua as ferramentas necessárias para a gestão do sistema de segurança;
- Montar uma área virtual para testes do sistema robótico e fazer todas as simulações necessárias.

1.4 Metodologia

A metodologia de desenvolvimento deste trabalho é dividida em quatro etapas:

Inicialmente, foram realizadas pesquisas bibliográficas na área de mapeamento, localização, planejamento de trajetórias de robôs e desvio de obstáculos, linguagem de programação C++, Python e SQL, criação e gerenciamento de banco de dados, desenvolvimento de nós e configuração de redes de comunicação no ROS, configuração e utilização do framework Qt para trabalhar em conjunto com o ROS, desenvolvimento de interface gráfica na linguagem C++.

Após o término das pesquisas bibliográficas foram definidos os algoritmos a serem empregados no sistema robótico desenvolvido neste trabalho, dentre estes estão os responsáveis pelo mapeamento, localização, navegação, detecção de ameaças, controle dos movimentos do robô, controle de base de dados, etc. Nesta mesma etapa foi decidido qual seria a estrutura do programa de computador e sua linguagem, e consequentemente, todas as suas classes com seus atributos e métodos mais relevantes.

A terceira etapa foi marcada pelo desenvolvimento do programa de computador responsável pelo controle do sistema robótico. Este programa possui dois componentes principais:

- um nó do ROS responsável por controlar o robô Turtlebot 2, por comunicá-lo com a interface gráfica, por detectar ameaças, por gerenciar os nós responsáveis pelo mapeamento, e pela localização e navegação do robô em um ambiente conhecido;
- uma interface gráfica para realizar a gestão do sistema robótico, contendo as ferramentas necessárias para o funcionamento do sistema de vigilância.

A última etapa é a dos testes do sistema. Foram realizados cinco tipos de testes, que serão descritos na seção 5.6, com o objetivo de garantir que o programa execute de forma estável, com todas suas funcionalidades operando corretamente e com uma interface aceita por usuários. Para garantir que os testes fossem bem sucedidos, utilizou-se um ambiente de teste e contou com a ajuda de nove voluntários para simularem usuários do sistema.

1.5 Organização do trabalho

Este trabalho está dividido em seis capítulos. Inicialmente, são introduzidos os conceitos básicos de sistemas de vigilância empregados na sociedade atual, bem como a motivação, o objetivo e a justificativa de realizar este trabalho. Por fim, é feito uma breve apresentação da metodologia empregada.

No segundo capítulo, são apresentados três projetos semelhantes ao desenvolvido neste trabalho, onde é implementado um robô móvel para realizar a patrulhamento. Dentre estes projetos está um produto final, um projeto de uma universidade brasileira e um projeto de uma universidade estrangeira.

Logo a seguir, no terceiro capítulo, é apresentada uma revisão da teoria envolvida com a robótica móvel necessária para o desenvolvimento do trabalho. São apresentados, de forma mais detalhada, a teoria a ser aplicada neste trabalho por meio da visão de autores das obras consultadas e de outros trabalhos voltados para teorias e tecnologias aplicadas em sistemas robóticos móveis, o que resulta em um referencial teórico direto e aplicado.

O capítulo quatro explana quais os materiais e métodos empregados para desenvolver um sistema robótico que realiza vigilância de ambientes internos, como qual robô utilizado e a os algoritmos que compõe o sistema robótico.

A seguir, o capítulo cinco apresenta os resultados obtidos, apresentando a estrutura final do aplicativo desenvolvido e realizando diferentes tipos de testes para avaliar a performance do sistema.

Por fim, nas considerações finais são apresentas as conclusões feitas deste trabalho, assim como as dificuldades encontradas e as propostas para trabalhos futuros.

2 TRABALHOS RELACIONADOS

Nesta secção serão apresentados três projetos de sistemas robóticos diretamente relacionados ao que foi desenvolvido neste trabalho, o Horus Patrol, bem como as semelhanças e diferenças do Horus Patrol em relação a cada projeto.

2.1 Knightscope K5 autonomous data machine

O K5 é um robô móvel autônomo desenvolvido pela Knightscope para realizar a vigilância de ambientes abertos por meio de patrulhas. Ele está hoje em fases de validação sendo empregado apenas na área do Vale de Silício nos Estados Unidos. De acordo com a empresa o objetivo principal deste robô é o de predizer e prevenir que um crime aconteça. A Knightscope trabalha apenas com aluguel do K5, com o preço de $6,25 por hora (KNIGHTSCOPE, 2015).

Este robô faz parte de um sistema robótico formado por ele e uma central de comando. Durante seu funcionamento, o robô reúne dados importantes em tempo real por meio seus inúmeros sensores, que é então processado por meio de um mecanismo de análise preditiva. Lá, ele é combinado com uma base de dados de informações sociais para determinar se há uma preocupação ou ameaça na área. Se assim, um problema é criado com um nível de alerta apropriado e uma notificação será enviada para a comunidade e autoridades por meio do Centro de Operações de Segurança da Knightscope, a KSOC, uma interface de usuário baseada em navegador.

Este robô possui 1,5 metros de altura, 0,9 metros de largura e tem uma massa de 136 quilogramas. Quanto as seus sensores, o robô tem 2 sensores para medições de distâncias, 4 vídeo câmeras, 1 GPS, 1 câmera específica para detecção de placas de carros, bem como sensores de temperatura, umidade, som, unidade de controle inercial, odometria, etc (KNIGHTSCOPE, 2015). A figura 1 ilustra a aparência do robô K5:

As principais diferenças do sistema da Kightscope em comparação com o Horus Patrol estão relacionados ao preço, ao local empregado e às *features*. O sistema da Knightscope é mais barato que um sistema convencional de vigilância, porém ele é mais caro que o sistema proposto neste trabalho. O valor mensal gasto com o K5 é de aproximadamente $4.500,00, enquanto que o valor total estimado para implementar este trabalho no mundo real é de $1.184,00. O sistema da Knightscope foi desenvolvido para ambientes externos, enquanto que o Horus Patrol foi desenvolvido para os internos. O K5 tem uma quantidade superior de *features* em relação ao robô utilizado neste trabalho, como a detecção de placas de carros e análise preditiva de comportamentos.

Figura 1 – Kinghtscope K5

Fonte: (KNIGHTSCOPE, 2015)

2.2 Robôs táticos em ambientes indoor

Este projeto foi desenvolvido pelo Instituto de Ciências Matemáticas e de Computação (ICMC) da Universidade de São Paulo (USP) com o objetivo de criar um sistema robótico para monitoramento e segurança de ambientes internos (*indoor*) com o uso de robôs móveis. Os trabalhos desenvolvidos criaram robôs móveis com uma arquitetura de software que implementa um sistema de controle inteligente. O robôs podem ser tele-operados (operação remota semiautônoma) ou podem agir em modo completamente autônomo. Este sistema é capaz de detectar intrusos e situações anômalas, navegar no ambiente desviando dos obstáculos, garantindo assim a integridade do robô e dos elementos presentes no ambiente (OSÓRIO et al., 2011).

Neste projeto foram testados diversos métodos de navegação e localização, porém o método de navegação baseada no sensor Microsoft Kinect® e o método de localização baseada em partículas (MCL) foram os métodos que obtiveram melhores resultados.

As principais diferenças do sistema robótico desenvolvido por este grupo de pesquisadores em comparação com o Horus Patrol estão relacionados ao preço, e às *features*. O sistema deste grupo utilizada uma câmera térmica modelo FLIR PathFindIR que custo aproximadamente $2.495,00 e um robô Pioneer que custa cerca de $4.000,00 totalizando em um valor de $6.495,00 o que é maior que cinco vezes o valor estimado para implementar o projeto desenvolvido neste trabalho. O projeto do grupo de pesquisadores se limitou ao

estudo dos métodos para realização da localização e navegação e, por fim, integrá-lo a um sistema de detecção de intrusos, porém neste trabalho o foco foi no desenvolvimento do sistema de segurança como um todo, visando um sistema viável a um ambiente real possuindo todas as funções necessária para um bom funcionamento da atividade de vigilância. Sendo assim, o Horus Patrol possui mais ferramentas, como gerenciamento de detecções e de rotas de patrulha (OSÓRIO et al., 2011).

Figura 2 – Sistema robótico em tele-operação

Fonte: (OSÓRIO et al., 2011)

2.3 Robô móvel autônomo de segurança em navios

Este trabalho apresenta uma abordagem diferente de um sistema de patrulha, onde o ambiente a ser patrulhado é um navio ao invés de uma área urbana. Neste cenário o sistema robótico não tem como objetivo detectar intrusos e sim possíveis princípios de incêndio que surjam em um navio. O sistema robótico foi desenvolvido pelo pesquisador Long-Yeu Chung da universidade de Chian Nan. Ele é constituído por uma interface gráfica e robô móvel com acionamento diferencial (CHUNG, 2013).

O robô móvel, ilustrado na figura 3, é o responsável por detectar um possível incêndio no navio e, para isto, ele conta com sensores que detectam fumaça, chamas e temperatura. Além destes sensores, o robô possui um sensor laser e um ultrassônico para realizar sua navegação, e uma câmera para captura de imagens. A interface gráfica se comunica com o robô via rede *wireless* e visualiza as imagens capturadas pela câmera e as informações coletadas pelos sensores de detecção de incêndio do robô.

As principais diferenças do sistema robótico desenvolvido por este pesquisador em comparação com o Horus Patrol estão relacionados ao preço, à variável detectada pelo sistema, e às *features* da interface gráfica. O sistema deste pesquisador utilizada uma câmera SONY-EVI CCD e um sensor laser SICK LMS-100. Apenas estes sensores somados custam aproximadamente $4.750,00, o que é maior que quatro vezes o valor estimado

para implementar o projeto desenvolvido neste trabalho. O Horus Patrol possui mais ferramentas, como gerenciamento de detecções e de rotas de patrulha, o que não existe no sistema desenvolvido por Long-Yeu Chung. A maior vantagem do sistema deste pesquisador está na qualidade das imagens coletadas pela câmera, a possibilidade de mover a câmera sem mover o robô e o alcance do seu sensor laser, bem como seu ângulo de visão.

Figura 3 – Robô móvel para detecção de incêndios

Fonte: (CHUNG, 2013)

3 REFERENCIAL TEÓRICO

Neste capítulo será descrita a definição de robôs e seus tipos; a história dos robôs autônomos móveis e suas aplicações; um resumo sobre odometria; uma breve explanação das técnicas empregadas para realizar o mapeamento, a localização e a navegação de robôs móveis; as técnicas empregadas para realizar a detecção de seres humanos; os conceitos do ROS; um resumo da linguagem de programação C++, Python e SQL; e por fim, uma breve explanação acerca da tecnologia TCP/IP e das redes sem fio.

3.1 Robôs e suas classificações

A *Robotic Industries Association* (RIA) considera como sendo robô um dispositivo mecânico programável para execução de algumas tarefas de manipulação ou locomoção sob controle automático. Define ainda que um robô industrial é um manipulador multifuncional e reprogramável projetado para movimentar materiais, peças e ferramentas ou dispositivos especiais, conforme programação prévia, de modo a executar uma variedade de tarefas (NEHMZOW, 2003).

Para a *Japan Industrial Robot Association* (JIRA), o robô é definido como um sistema mecânico que possui movimentos flexíveis análogos aos movimentos orgânicos, e combina esses movimentos com funções inteligentes e ações semelhantes às do humano. Neste contexto, função inteligente significa o seguinte: decisão, reconhecimento, adaptação ou aprendizagem.

Segundo Secchi (2008), existem diversos tipos de robôs e, por consequência disso, existem diversas classificações, onde os robôs podem ser classificados quanto a sua:

- Aplicação: Para o uso doméstico, industrial, médico, bélico, espacial, no entretenimento ou em serviços;
- Meios de atuação: Podem ser terrestres, aéreos, aquáticos, espaciais;
- Processamento: Desprovidos de processamento, processamento mecânico, por meio de eletroválvulas, eletrônicos por meio de transistores ou por meio de circuitos integrados;
- Autonomia: Controlados, programados, autônomos e inteligentes;
- Forma de Locomoção (Cinemática): Robôs estacionários, móveis com rodas, móveis com pernas, rastejantes, voadores, nadadores.

3.1.1 Robôs móveis autônomos com rodas

Os robôs autônomos são máquinas eletromecânicas controladas por um programa de computador ou até mesmo um circuito elétrico capazes de realizar trabalhos de maneira autônoma ou pré-programada.

Segundo Marchi et al. (2001), um robô móvel é um dispositivo mecânico montado sobre uma base não fixa que age sob o controle de um sistema computacional, equipado com sensores e atuadores que o permitem interagir com o ambiente.

De modo semelhante, define-se como robô móvel com rodas uma máquina que seja capaz de se locomover em um ambiente de maneira manual ou autônoma, equipado com motores e sensores ambos controlados por uma unidade central de processamento. O primeiro robô móvel com rodas autônomo que se tem registro foi o Machina Speculatrix, também chamado de "Tortoise", criado por Grey Walter em meados da década de 1940 (SABBATINI, 1999). Este robô possuía duas funções: se afastar ou se aproximar da fonte luminosa presente no ambiente, desviando de possíveis obstáculos.

No ano de 1954, a Barrett Electronics criou o primeiro tipo de robô capaz transportar materiais de um ponto a outro de forma autônoma utilizando um fio emissor de sinais de alta frequência como meio orientação, tal tipo de robô foi denominado Automated Guided Vehicles (AGV). Atualmente existem AGV que se orientam utilizando outros métodos além de fios, como por meio de lasers infravermelhos, GPS, fitas, sistemas de navegação inercial entre outros (SAVANTY, 2014).

Pieri (2002) descreve outros tipos de robôs móveis foram desenvolvidos nos anos seguintes:

- Os robôs de serviço, que são imersos em ambientes estruturados (residências, pátios, etc), e reconhecidos por meio de modelos internos executando tarefas de limpeza, transporte, vigilância e manipulação de objetos;

- Os robôs de campo, que executam tarefas em ambientes desestruturados, pouco conhecidos, muitas vezes perigosos. Podem ser usados em exploração espacial, mineração, busca e resgate, limpeza de acidentes nucleares, entre outras aplicações.

hspace0,6cmInúmeros modelos de robôs móveis com rodas foram desenvolvidos na última década, porém com um uso mais ostensivo e restrito, predominantemente em pesquisa de universidades, indústrias e órgãos de pesquisa. O exemplo mais notável dos últimos anos é o robô Curiosity desenvolvido no Jet Propulsion Laboratory (JPL) em parceria com a National Aeronautics and Space Administration (NASA). Este robô tem como objetivo explorar o terreno do planeta Marte e investigar a capacidade deste planeta sustentar vida microbiana (NASA, 2013).

Figura 4 – Exemplos de Robôs móveis com rodas

Fonte: (SABBATINI, 1999; AUTOMATION, s.d.; NASA, 2013)

3.2 Odometria

Segundo Bezerra (2004), a odometria é um método utilizado para estimar a posição e orientação de um robô por meio da integração dos deslocamentos incrementais de suas rodas, medidos a partir de um referencial fixo. Normalmente, os encoders rotativos são os dispositivos utilizados para tal tarefa em conjunto com uma unidade de processamento, sendo esta responsável pelo cálculo das estimativas.

O encoder rotativo é acoplado ao eixo do motor e tem como função retornar pulsos elétricos a medida que o eixo rotaciona. Com esta técnica é possível determinar, por meio da contagem de pulsos gerados pelo encoder, a posição angular do eixo do motor, o número de voltas dada pela roda do robô e, assim, estimar o deslocamento do robô e sua orientação.

Ainda segundo o autor, a odometria é um método simples e comum de ser empregado em robôs para estimar a localização de um robô móvel. Porém, este método está sujeito a falhas, como, quando as rodas de um robô perdem aderência ao chão. Neste caso, a roda irá rotacionar no vazio gerando erros nas medidas. Tais erros são acumulativos, pois a odometria determina a localização do robô com base no acúmulo de informações coletadas.

Para melhorar a estimativa da posição de um robô, podem ser empregados sensores inerciais como um giroscópio. Este dispositivo é composto de um rotor que pode rotacionar livremente em torno de seus eixos geométricos perpendiculares entre si que se interceptam no seu centro de gravidade. Em suspensão tipo Cardan, um giroscópio pode possuir qualquer orientação, entretanto seu centro de massa permanece fixo no espaço (JÚNIOR, 2014).

3.3 Mapeamento, localização e navegação robótica

No desenvolvimento de um sistema robótico é necessário realizar o projeto do *hardware*, definindo quais sensores, atuadores, interfaces e processadores serão utilizados, e realizar o

projeto de *software*, desenvolvendo o sistema de controle do robô, analisando as informações de sensores e computando as respostas dos atuadores. Está seção irá expor técnicas de mapeamento, localização e navegação de robôs empregadas em projetos de *software*.

3.3.1 Mapeamento

O processo de mapear na robótica consiste na exploração e construção de um mapa digital de um ambiente, identificando onde existem paredes e obstáculos gerando, por exemplo, um mapa de ocupação(WOLF et al., 2009). Para construir um mapa, utiliza-se as informações coletada dos sensores (*laser*, sonares).

Os mapas são utilizados na navegação de robôs, auxiliando com as informações necessárias para se identificar o caminho mais eficiente a ser percorrido. Outra utilização de mapas é na localização de robôs, pois é necessário que o robô saiba com precisão sua localização dentro do ambiente para assim partir para a navegação.

Ainda segundo Wolf et al. (2009), há dois tipos de mapas criados por robôs móveis: topológicos e métricos.

Os mapas topológicos (*roadmap*) representam o ambiente por grafos, no qual os vértices representam regiões de interesse no ambiente e os caminhos representam as vias que interligam estes vértices. Este tipo de mapa é excelente para realizar o planejamento de trajetórias, mas apresentam uma representação muito pobre do ambiente. Vários métodos baseados nesta abordagem foram propostos. Entre eles: grafos de visibilidade, diagramas de Voronoi e decomposição celular.

Os mapas métricos representam o ambiente físico com detalhe e também podem ser utilizados para estimar trajetórias. Entre os mapas métricos, destaca-se a representação utilizando grades de ocupação.

Atualmente, as técnicas mais empregadas para realizar o mapeamento de um ambiente é por meio de algoritmos que resolvam o problema de *Simultaneous Localization And Mapping* (SLAM). Este problema descreve a situação em que um robô precisa mapear um ambiente ao mesmo tempo em que ele se localiza no mesmo. Este problema é complexo e pode ser resumido pela pergunta: “Quem vem primeiro, o mapa ou a localização?”. Um mapa não pode ser gerado sem se conhecer a localização, e a localização precisa de um ponto de referência (um mapa) para ser conhecida. Para solucionar o problema do SLAM, vários algoritmos foram desenvolvidos, dentre eles destaca-se o GMapping.

O GMapping é um algoritmo *open-source* desenvolvido pelo Prof. Dr. Wolfram Burgard e seu time que utiliza um filtro de partícula Rao-Blackwellized para gerar um mapa métrico (do tipo grades de ocupação) ao longo do deslocamento do robô em um ambiente. Suas duas maiores vantagens em relação a outros métodos de mapeamento utilizando partículas é a redução de partículas necessárias e o baixo risco de o algoritmo descartar partículas corretas, portanto é um algoritmo que necesssita de menos processamento para atingir seu objetivo.

Antes de iniciar o algoritmo do GMapping, é necessário entender o que é uma partícula. Uma partícula representa um mapa hipotético contendo uma pose do robô. No algoritmo, estas particulas são inicialmente distribuídas de maneira aleatória, porém em futuras reamostragens, estas partículas são geradas próximas dos pontos que o algoritmo acredita serem os corretos (GRISETTI; STACHNISS; BURGARD, 2007).

Este algoritmo pode ser resumido em duas etapas que se repetem diversas vezes. Na primeira, o robô coleta as informações de *landmarks* (características importantes do ambiente como o canto de uma parede) e gera sua estimativa. O mesmo ocorre em cada partícula. Na segunda etapa o robô se movimenta, estima sua nova posição e verifica quais partículas se aproximam mais do correto (real). Após está verificação, é realizada uma reamostragem das partículas. E assim, o processo retorna a sua primeira etapa, repetindo assim o ciclo.

3.3.2 Localização

Localização consiste em estimar a posição de um robô em um ambiente, sendo necessária para qualquer atividade que envolva a navegação. É fundamental que um robô consiga determinar exatamente sua posição em um ambiente para que o mesmo possa planejar uma trajetória até seu destino.

O maior problema da localização é a incerteza nas medições dos sensores, pois, como já foi explicado, os erros são cumulativos. O grande desafio na solução do problema de localização está no fato de que tanto as informações sobre o ambiente como os dados fornecidos pelos sensores são normalmente limitados e imprecisos. Nesse contexto, técnicas probabilísticas tem sido amplamente utilizadas na solução desse problema (WOLF et al., 2009).

O problema de localização pode ser dividido em: local e global. No problema de localização local, a posição inicial do robô é conhecida. A solução para esse problema consiste em lidar com a imprecisão dos sensores e manter uma estimativa consistente da posição do robô no ambiente. No caso da localização global, o robô não tem informações prévias de sua posição no mapa, o que torna o processo de localização consideravelmente mais complexo. Dentre os algoritmos de localização para robôs móveis, destaca-se o método de Markov. O algoritmo de localização de Markov é baseado no filtro de Bayes, uma poderosa técnica de estimação estatística, aplicado ao contexto da localização robótica (THRUN; BURGARD; FOX, 2005).

Existem diversas formas de implementar esse algoritmo e de representar os dados do ambiente e da posição dos robôs. Dentre as quais destacam-se o filtro de Kalman, o método de *grid* e o *Monte Carlo Localization* (MCL). O método MCL utiliza o conceito de partículas (semelhante ao utilizado pelo GMapping) para descobrir a localização de um robô em um determinado ambiente. Uma partícula representa um possível pose do robô no mapa. Quando iniciado, o MCL distribui partículas em posições aleatórias no

ambiente. É definido uma pontuação para cada partícula comparando os dados obtidos dos sensores com os *landmarks* do mapa. Esta pontuação representa a chance dessa partícula representar a real localização do robô, onde ela é atualiza conforme o robô se desloca no ambiente. Partículas com a pontuação baixa são eliminadas enquanto as partículas com peso alto são replicadas. Com o tempo todas as partículas tendem a se concentrar no local do mapa que representa a estimativa da localização correta do robô (WOLF et al., 2009).

Figura 5 – Exemplo do avanço das partículas utilizando o MCL

(a) Partículas Iniciais

(b) Algoritmo em processamento

(c) Partículas finais

Fonte: (WOLF et al., 2009)

3.3.3 Navegação

O objetivo da navegação na robótica é controlar os movimentos de um robô móvel, garantindo um deslocamento seguro, de um ponto inicial até seu destino. Para realizar essa tarefa de uma maneira ótima, o software de navegação deve possuir dois algoritmos básicos: um para planejar trajetórias e outro para evitar obstáculos (RIBEIRO, 2005).

Planejamento de trajetórias

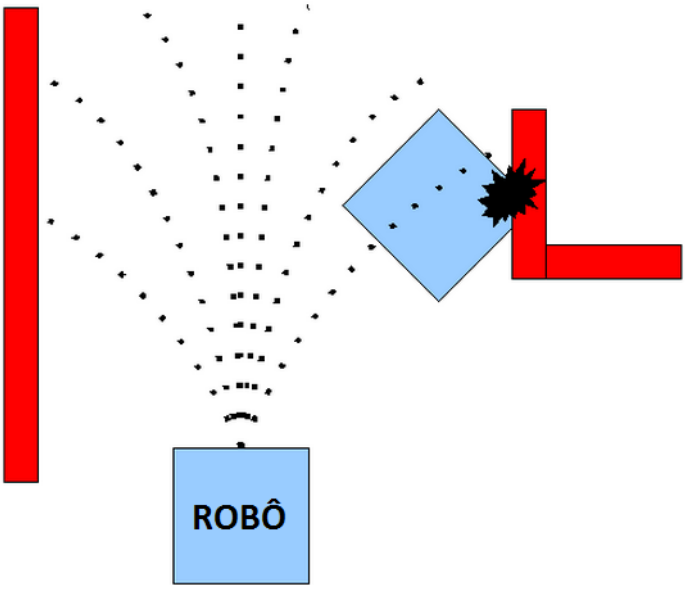
Segundo Chatila (1995), planejar um caminho é a competência que um robô possui de selecionar, dentre várias possibilidades, a rota a ser percorrida utilizando um mapa preestabelecido e as informações coletadas por seus sensores. Existem duas vertentes, o planejamento global, que permite ao robô traçar um caminho por um local fora do alcance de seus sensores, e o planejamento local que é responsável pela navegação em curta distância, baseada nos valores coletados pelos seus sensores. Alguns exemplos de algoritmos para planejamento de trajetórias são o Djikstra e o A*.

Desvio de obstáculos

Desvio de obstáculos refere-se às metodologias de moldar o caminho do robô para superar obstáculos inesperados. O movimento resultante do robô depende de sua localização real e das leituras dos sensores. Há uma rica variedade de algoritmos para evitar obstáculos desde replanejamentos até alterações reativas na estratégia de controle. Técnicas propostas se diferenciam na utilização dos dados sensoriais e nas estratégias de controle de movimento para superar os obstáculos (RIBEIRO, 2005).

Alguns exemplos de algoritmos para desvio de obstáculos são Bug, campos potenciais, *Vector Field Histogram* (VHF) e o *Dynamic Window Approach* (DWA). Neste trabalho foi utilizado o DWA como o algoritmo de desvio de obstáculo e realizar a planejamento de trajetória local. Este algoritmo gera um janela dinâmica e nela realiza simulações de “n” possíveis trajetórias (baseadas na velocidade atual do robô) que o robô pode escolher, procurando a trajetória que não colida com um obstáculo e possua a melhor pontuação. Esta pontuação é baseáda nos seguintes critérios: proximidade de um obstáculo, proximidade do objetivo e velocidade. Após a pontuação, é escolhido a trajetória e se reinicia o algoritmo com uma nova janela. A figura 6 ilustra um exemplo de uma janela dinamíca e as possíveis trajetórias (FOX; BURGARD; THRUN, 1997).

Figura 6 – Exemplo de uma janela dinâmica do DWA com suas possíveis trajetórias.



Fonte: ROS Wiki (http://wiki.ros.org/dwa_local_planner)

3.4 Detecção de presença humana por meio de sensores

A detecção da presença humana utilizando sensores em ambientes é geralmente dividida nas categorias: detecção de movimento, ou detecção de emissão de energia eletromagnética, sendo ambas estudadas e testadas por meio de sensores ativos e passivos, como: sensores infravermelhos, ultrassônicos e de micro-ondas (HARMON, 1982).

Detecção de movimento se concentra em mudanças em um ambiente estável, sendo realizada com sucesso utilizando radares Doppler de micro-ondas, examinando o efeito Doppler do sinal refletido em uma pessoa em movimento. Este tipo de detecção por meio de movimentos é empregada em aplicações estacionárias, como um ambiente interno, porém é péssima em ambientes com mudanças constantes como um sensor acoplado à um robô móvel (EVERETT, 1995).

A detecção por meio de movimentos é o método mais utilizado pela sociedade, por ser mais simples e barato, porém ele possui uma limitação na qual ele não consegue detectar a presença de humanos que estão parados. Neste cenário deve ser empregada a técnica de detecção de humanos pela emissão de energia, que se concentra em sinais intrínsecos produzidos por um ser humano, como a radiação, calor (temperatura) ou sinais acústicos. Um exemplo deste método foi aplicado ao robô ROBART II, que utilizou arrays de transdutores acústicos para triangular a posição de algo que esteja emitindo algum som (NANZER; ROGERS, 2007).

3.4.1 Sensores infravermelhos

Existem sensores de infravermelho (IR) ativos e passivos. Um sensor de infravermelho ativo é composto por um emissor de luz infravermelha e um receptor, que reage a essa luz. Por sua vez, um sensor de infravermelho passivo não emite luz infravermelha, mas apenas capta esse tipo de luz no ambiente. O princípio de detecção deste tipo de sensor se baseia na teoria da emissão de radiação eletromagnética de qualquer objeto cuja temperatura seja superior ao zero absoluto (MAZZAROPPI, 2007). Um exemplo de um sensor ativo é o Microsoft Kinect®, já exemplos de sensores passivos são:

- Sensores IR passivos piroelétricos (PIR), que são sensores de movimento muito utilizados como chaves inteligentes para controle de lâmpadas em ambientes comerciais ou domiciliares. O modelo HC-SR501 é composto de uma lente Fresnel um sensor piroelétrico e uma placa de controle com potenciômetros para o ajuste de sensibilidade e time delay (OLIVEIRA; PETREK, 2014);

- Sensores de pilhas de termopares para detecção de temperatura, utilizados para detecção de movimento e presença. O modelo Omron D6T é composto de uma lente de silicone, uma pilha de termopares *Micro-Electro-Mechanical Systems* (MEMS) e um microcontrolador (KUKI et al., 2012).

hspace0,6cmEstes dispositivos funcionam bem em determinadas situações, tais como ambientes internos devido forte contraste térmico, bem como a falta de luz solar. Em um ambiente interno a assinatura térmica de uma pessoa na banda IR é dominante sobre a maioria das outras fontes de radiação e, portanto, uma pessoa pode ser detectado por meio de seu sinal térmico quente em contraste com um fundo mais frio (NANZER; ROGERS, 2007).

3.5 Robot Operating System (ROS)

Segundo (QUIGLEY et al., 2009), ROS é um *software framework open-source* contendo um conjunto de bibliotecas, pacotes e ferramentas para desenvolvimento de aplicações voltadas para a área da robótica, com o intuito de facilitar a pesquisa e o estudo de sistemas robóticos. Atualmente, de forma estável, o ROS suporta apenas o sistema operacional Linux Ubuntu como sua base. ROS provê recursos presentes em um sistema operacional comum, como abstração de *harware*, controle de dispositivos de baixo nível, implementação de funcionalidades de uso comum, transmissão de mensagens entre processos e gerenciamento de pacotes e arquivos.

Sua primeira versão foi publica em Março de 2010, sendo chamada de ROS Box Turtle. Está versão foi desenvolvida para o sistema operacional Linux Ubuntu 8.04.4 LTS (Hardy

Heron), e a maioria dos principais recursos do ROS tiveram sua versão inicial 1.0 com esta versão, o que exigiu teste intensivos por parte dos usuários (CONLEY, 2011).

Dentre as vantagens de se utilizar o ROS estão: ambiente pronto para programação, independência da linguagem de programação, bibliotecas e ferramentas com interface amigável, ambiente propício para testes com a utilização de simuladores, reutilização de códigos para a pesquisa e desenvolvimento na área da robótica e compatibilidade dos códigos criados no ROS com outros frameworks de robótica e, por fim, ROS é apropriado para grande sistemas.

Segundo (FOOTE, 2013), os conceitos do ROS podem ser divididos em três níveis: sistema de arquivos, computação gráfica e comunidade.

3.5.1 Nível de sistema de arquivos (Filesystem Level)

Este nível possui todos os recursos do ROS presente em disco:

- Pacotes: São as unidades principais para a organização de *software* no ROS. Um pacote pode conter processos *runtime* (nós), uma biblioteca, *datasheets*, arquivos de configuração;
- Manifestos: Provê a *metadata* e a descrição de um pacote, incluindo informação de licença, *flags* do compilador e as dependências do pacote;
- *Stacks*: São coleções de pacotes que oferecem funcionalidade agregada, como a "*navigation stack*";
- Manifestos de *Stacks*: Provê a *metadata* e a descrição de um *stack*, incluindo informação de licença e as dependências com outros *stacks*;
- Tipos de Mensagem (msg): Descrição de mensagens, definindo a estrutura de dados para mensagens enviadas no ROS;
- Tipos de Serviço (srv): Descrição de serviços, definindo a estrutura de dados de solicitações e respostas para os serviços no ROS.

3.5.2 Nível de computação gráfica (Computation Graph Level)

Nível de computação gráfica é a rede *peer-to-peer* dos processos ROS que processam dados em conjunto. Os conceitos básicos são:

- Nós: São processos que executam a computação. Um sistema robótico normalmente possuirá vários nós. Por exemplo, um nó controla um laser scan, outro controla os motores, um nó executa um algoritmo de localização, outro um algoritmo para planejamento de rotas. Um nó é escrito com o uso de bibliotecas, como "roscpp";

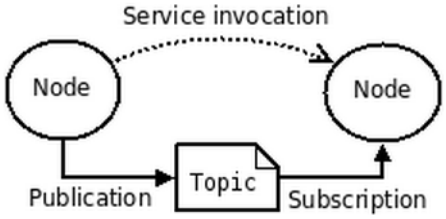
- Mestre: Responsável por prover serviços de nomeação e registro para resto dos nós, além de permitir que nós localizem outros nós;
- Servidor de parâmetro: Permite que dados sejam armazenados em um local central. Atualmente este conceito faz parte do mestre;
- Mensagens: Nós se comunicam com outros nós por meio de mensagens. Uma mensagem é simplesmente uma estrutura de dados, compreendendo os campos digitados. Mensagens podem ser de tipos primitivos (inteiro, ponto flutuante, booleano, etc.) ou estruturas ou matrizes;
- Tópicos: Mensagens são encaminhadas por meio de um sistema de transporte com as funções publicar/inscrever, os tópicos. Um nó envia uma mensagem publicando para um tópico específico. O tópico é um nome utilizado para identificar o conteúdo da mensagem. Um nó interessado em certo tipo de dado irá se inscrever no tópico apropriado. Podem haver vários nós publicando ou inscritos no mesmo tópico e um nó pode publicar ou se inscrever em múltiplos nós;
- Serviços: São definidos por um par de mensagens: um para o pedido (*request*) e uma para a resposta (*reply*). Um nó oferece um serviço com um nome *string*, e um cliente chama o serviço por meio do envio da mensagem de pedido e aguarda a resposta.

O Mestre do ROS funciona como um serviço de nomes. Ele armazena as informações de registro dos tópicos e serviços para os nós, e estes, se comunicam com o Mestre informar suas informações de registro. Caso alguma informação de registro mude, o Mestre será responsável por fazer um *Callback* com os nós.

Os nós se conectam diretamente, sendo que o mestre apenas funciona de maneira semelhante a um servidor de domínio (DNS), provendo informações de *lookup*. Nós inscritos em um tópico solicitam conexões dos nós que publicam neste tópico e estabelecerão esta conexão por meio de um protocolo. O protocolo de comunicação mais comum empregado no ROS é chamado de TCPROS, que é baseado no TCP/IP.

Uma maneira de exemplificar o funcionamento deste nível é por meio da figura 7. Os nós podem se comunicar por meio de tópicos ou serviços, sendo a comunicação por meio de tópicos a mais comum de ser utilizada. Nesta, um nó publica uma mensagem por meio de um tópico, e os nós que estão inscritos neste tópico recebem a mensagem enviada

Figura 7 – Conceito da comunicação do ROS



Fonte: (FOOTE, 2013)

3.5.3 Nível de comunidade (Community Level)

Definido como sendo os recursos do ROS que possibilitam uma troca de *softwares*, conhecimentos e informações entre comunidades. Os recursos disponíveis são:

- Distribuições: São coleções de versões de *stacks* que podem ser instalados. Possuem a função semelhante das distribuições de Linux: facilitam a instalação de coleções de *softwares*, e eles também mantêm versões consistentes em um conjunto de *software*;
- Repositórios: ROS se baseia em uma rede federada de repositórios de código, onde diferentes instituições podem desenvolver e publicar seus próprios componentes de *software* de robôs;
- O ROS Wiki: É o fórum principal para documentar informações sobre ROS. Qualquer pessoa pode se inscrever e contribuir com a sua própria documentação, fornecer correções ou atualizações e escrever tutoriais;
- ROS *Answers*: Um site de qualidade assegurada para responder as dúvidas de usuários;
- Blog: É um *blog* que prover informações referente a atualizações incluindo fotos e vídeos.

3.6 Linguagem de programação C++, Python e SQL

A linguagem de programação tem como objetivo informar instruções a serem executadas por um computador, possuindo um conjunto de regras sintáticas e semânticas. Ao longo dos anos várias linguagens foram desenvolvidas e, por causa da predominância no ROS, esta pesquisa se limitará ao uso de duas delas: Python e C++.

A linguagem de programação C++ é uma linguagem de médio/alto nível que provê um modelo de memória e computação que se ajusta em quase qualquer computador. Ele também fornece mecanismos para abstração, que permitem ao programador introduzir, modificar e utilizar tipos de objetos que se encaixam a determinada aplicação. Além disso, C++ suporta estilos de programação que dependem de manipulação de recursos

do *hardware* para entregar um alto grau de eficiência (STROUSTRUP, 1999). Esta linguagem foi inicialmente denominada de linguagem C com classes, pois introduziu o uso da programação orientada objeto nesta linguagem. Uma classe é uma estrutura que agrupa um conjunto de objetos com características similares, como a classe "professor", que agrupa o conjunto de pessoas denominadas professores. Cada professor possui certas propriedades próprias, como nome, idade, disciplina ministrada. Além destes atributos, cada professor possui certos métodos, como lecionar aula ou pesquisar. Esta mesma analogia pode ser aplicada às classes da linguagem C++, cada objeto dentro de uma classe possui seus atributos, que definem suas propriedades, e seus métodos, que definem suas ações.

A linguagem de programação Python é uma linguagem de alto nível que combina seu poder com uma sintaxe simples e clara. Ela possui módulos, classes, exceções, tipos de dados dinâmicos de alto nível e tipagem dinâmica. Há interfaces para várias chamadas de sistema e bibliotecas, bem como interface para sistemas de janela em uma interface gráfica do usuário (GUI). Novos módulos embutidos, que utilizam outras linguagens de programação, são facilmente escritos. Python também pode ser usado como uma linguagem de extensão para aplicações escritas em outras linguagens que necessitam de *scripts* simples e fáceis ou ainda interfaces de automação (ROSSUM; JR, 1995).

Python e C++ são linguagens de alto nível que suportam a maioria das técnicas da programação orientada a objeto. Ambas as linguagens possuem outras semelhanças, porém Python possui uma vantagem: o desenvolvimento de um código feito nesta linguagem necessita de menos tempo em comparação com um código em C++, consequentemente, são mais curtos. Esta diferença deve-se ao fato de que Python possui tipos de dados de alto nível e uma tipagem dinâmica, conforme dito anteriormente. Em contra partida, códigos feitos em C++ executam mais rapidamente. Desta maneira, Python é comumente empregado como uma linguagem *glue*, gerenciando códigos e conectando diferentes componentes de um *software*, enquanto que o C++ é empregado em uma linguagem de implementação de baixo nível (ROSSUM, 2002).

A *Structured Query Language* (SQL) é uma linguagem de programação voltada para o uso em gerenciamento de banco de dados. Esta linguagem foi desenvolvida como uma linguagem de manipulação de dados para um protótipo de sistema de gerenciamento de dados da IBM, chamado de System R, sob o nome de SEQUEL. Somente nos anos oitenta a linguagem começou a ser chamada de SQL. Atualmente esta linguagem é a mais utilizada para a manipulação de sistemas de gerenciamento de banco de dados (SGBD) (PRATT; LAST, 2008).

Para se obter um melhor entendimento acerca da necessidade desta linguagem de programação, faz-se necessário explicar os conceitos de banco de dados. Ainda segundo Pratt e Last (2008), um banco de dados é uma estrutura que contém diferente categorias de informação e a suas relações. Quanto ao desenvolvimento de um banco de dados, três conceitos são importantes conhecer: entidade, atributo e relacionamento. Entidade é um

objeto, um lugar, um nome, como “homem” ou “rota”. Um atributo é uma propriedade de uma entidade, como o atributo “cor dos olhos” ou “altura” pertencente a entidade “homem”. Por fim, relacionamento é a associação entre duas entidades como no caso das entidades “homem” e “mulher”.

3.7 Wireless Local Area Network (WLAN)

Comunicação *wireless* é uma aplicação da ciência e tecnologia que se tornou vital na atualidade, pois é por meio desta que a maioria dos dispositivos se conectam à internet. Este tipo de comunicação está em constante evolução contando com inúmeros padrões em operação, e com vários outros em desenvolvimento. O padrão que define as especificações para a implementação de uma comunicação *wireless* é o IEEE 802.11, sendo este subdividido em outros padrões (BHOYAR; GHONGE; GUPTA, 2013). Um exemplo de tecnologia que utiliza as especificações contidas no padrão IEEE 802.11 é o Wi-Fi, que é a tecnologia wireless dominante da atualidade.

Como dito anteriormente, o padrão IEEE 802.11 estabelece especificações para a criação e uso de redee decisão de quais algoritmos de mapeamento, localização e navegação, s wireless. A transmissão deste tipo de rede é feita por sinais de radiofrequência, que se propagam pelo ar e podem cobrir áreas na casa das centenas de metros. Como existem inúmeros serviços que podem utilizar sinais de rádio, é necessário que cada um opere de acordo com as exigências estabelecidas pelo governo de cada país. Esta é uma maneira de evitar problemas, especialmente interferências (ALECRIM, 2008). Há, no entanto, alguns segmentos de frequência que podem ser usados sem necessidade de aprovação direta de entidades apropriadas de cada governo: as faixas *Industrial, Scientific and Medical* (ISM), que podem operar, entre outros, com os seguintes intervalos: 902 MHz - 928 MHz; 2,4 GHz - 2,485 GHz e 5,15 GHz - 5,825 GHz (dependendo do país, esses limites podem sofrer variações).

A estrutura básica de uma rede WLAN segundo Crow et al. (1997) conta com o *basic service set* (BSS) definido como o conjunto de estações que estão sob o controle direto de uma função de coordenação tendo sua estrutura de rede estabelecida utilizando um ponto de acesso (AP). Uma estação (STA) deve poder se comunicar diretamente com qualquer outra STA dentro de uma BSS.

Segundo Bhoyar, Ghonge e Gupta (2013), os padrões do 802.11 mais utilizados são:

- IEEE 802.11a (1999): Este padrão utiliza o espectro do 5 GHz e possui uma transferência de dados teórica máxima de 54Mbps. Por causa de sua alta frequência, este padrão possui uma atenuação maior em comparação com os outros padrões (maior a perda de sinal com o distanciamento da fonte), seu alcance *indoor* é de aproximadamente 35m e o alcance *outdoor* é de 120m. São encontrados em redes corporativas ou provedores de rede *wireless* em áreas abertas;

- IEEE 802.11b (1999): Este padrão utiliza o espectro do 2.4 GHz e possui uma transferência de dados teórica máxima de 11Mbps. Seu alcance *indoor* é de aproximadamente 38m e o alcance *outdoor* é de 140m. Apesar de possuir uma taxa de transferência inferior ao padrão 802.11a, este padrão foi muito mais usado;

- IEEE 802.11g (2003): Este padrão utiliza o espectro do 2.4 GHz e possui uma transferência de dados teórica máxima de 54Mbps. Seu alcance *indoor* é de aproximadamente 38m e o alcance *outdoor* é de 140m. É o padrão mais utilizado atualmente e é compatível com o 802.11b;

- IEEE 802.11n (2009): Este padrão utiliza o espectro do 2.4 e 5 GHz e possui uma transferência de dados teórica máxima de 72Mbps (utilizando uma antena e um canal de 20Mhz). Porém este padrão pode chegar a 600Mbps teóricos, com seu sistema de controle *multiple-input multiple-output* (MIMO), que possibilita a utilização de até quatro antenas para captação de dados. Seu alcance *indoor* é de aproximadamente 70m e o alcance *outdoor* é de 250m (PERAHIA, 2008).

3.8 Tecnologia TCP/IP

A pilha de protocolos TCP/IP é organizada em quatro níveis ou camadas conceituais, construídas sobre uma quinta camada correspondente ao nível físico ou de *hardware* (DOUGLAS, 2000). As funções de cada camada da pilha de protocolos são:

1. Camada Física: corresponde ao nível de hardware, que trata dos sinais eletrônicos. Esta recebe os *frames* da camada de enlace, convertidos em sinais eletrônicos compatíveis com o meio físico, e os conduz até a próxima *interface* de rede, que pode ser a do *host* destino ou a do *gateway* da rede, caso esta não pertença a rede local;

2. Camada de Enlace de Dados: responsável por aceitar os datagramas IP, encapsulá-los em *frames*, preencher o cabeçalho de cada *frame* com os endereços físicos de origem e destino e transmiti-los para uma rede específica. Possui o *device-driver* da *interface* de rede, por onde é feita a comunicação com a camada física;

3. Camada de Rede: Trata da comunicação entre *hosts*. Esta aceita uma requisição de envio de pacote vinda da camada de transporte, com a identificação do *host* para onde o pacote deve ser transmitido. Encapsula o pacote em um datagrama IP e preenche o cabeçalho com os endereços lógicos de origem e destino, dentre outros dados. Utiliza o algoritmo de roteamento para determinar se o datagrama deve ser entregue diretamente, ou enviado para um *gateway;*

4. Camada de Transporte: Provê comunicação entre aplicações (comunicação fim-a-fim). Responsável pelo estabelecimento e controle do fluxo de dados entre dois *hosts*.

Provê transporte confiável, garantindo que as informações sejam entregues sem erros. Divide os dados em pacotes que são repassados a camada de rede;

5. Camada de Aplicação: Onde são executadas as aplicações. Uma aplicação interage com os protocolos da camada de transporte para enviar ou receber dados. Cada aplicação escolhe o tipo de transporte.

4 MATERIAIS E MÉTODOS

Neste capítulo são abordados, de maneira detalhada, quais os materiais utilizados para realização do trabalho, bem com como uma explanação detalhada da metodologia adotada e suas etapas. Os materiais utilizados neste trabalho foram o robô móvel Turtelbot 2 e um notebook para executar a interface gráfica de usuário.

O trabalho foi desenvolvido baseando-se em estudos bibliográficos, publicações *online* e em experimentos práticos sobre os melhores métodos para permitir que um robô navegue em um ambiente interno de forma autônoma, para se criar um mapa métrico de algum ambiente, para um robô se localizar em um lugar mapeado, para realizar a detecção humana em um ambiente interno, e por fim, métodos para desenvolvimento de aplicativo de computador utilizando uma programação orienta a objetos.

4.1 Turtlebot

Turtlebot é um robô móvel com acionamento diferencial e com *software open-source* de baixo custo projetado para o uso em pesquisas e educação. Eles foi desenvolvido pela Willow Garage, um laboratório de pesquisas aplicadas à robótica, tendo sua primeira versão publicada em abril de 2011 (YIRKA, 2011). Após o início de uma parceria entre a Willow Garage e a Yujin Robot, empresa que criou a base móvel Kobuki, foi publicado a segunda versão deste robô, chamado de Turtlebot 2, no final do ano 2012. Está nova versão foi criada para resolver três problemas principais, que a primeira versão não resolvia: a possibilidade legal de exportação, capacidade de se utilizar tanto a voltagem de 110V quanto a de 220V, e a sua odometria de baixa precisão. a figura 8 ilustra a versão do Turtlebot 2.

Figura 8 – Turtlebot 2

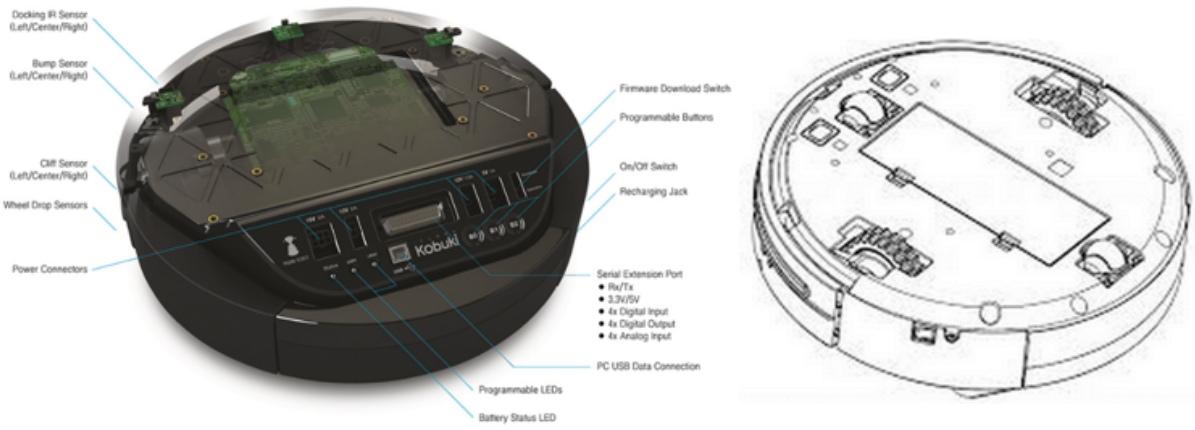
Fonte: (GARAGE, 2014)

Este trabalho utilizou a segunda geração deste robô, o Turtlebot 2, pois o mesmo é um robô móvel com acionamento diferencial e *software open-source*, com o melhor custo benefício identificado, ideal para aplicações de mapeamento e localização, que são os temas mais importantes para a execução deste projeto, além de utilizar o sistema ROS para controlar o robô, que possui uma grande comunidade de pesquisadores além de ser um *software* gratuito, ajudando assim para a criação de um sistema robótico de baixo custo(INTERNATIONAL, 2014).

O Turtlebot 2 é composto de três dispositivos principais:

- iClebo Kobuki: base móvel aplicada à robótica, baseada em um modelo de aspirador de pó autônomo, composta de 2 motores DC, sistema de odometria, uma giroscópio, uma bateria de Li-ion com autonomia de até três horas, botões e LEDs programáveis, e sensores para detecção de toque e desnível. Esta base é responsável por movimentar o Turtlebot, detectar o contato da base com algo, bem como, prover alimentação para a unidade central de processamento, para o sensor de movimento Microsoft Kinect® ou outros sensores acoplados ao Kobuki (ROBOT, s.d.). A figura 9 ilustra a vista superior e inferior da base Kobuki e todos os seus componentes;

Figura 9 – Vista superior (foto à esquerda) e inferior (foto à direita) da base móvel Kobuki.

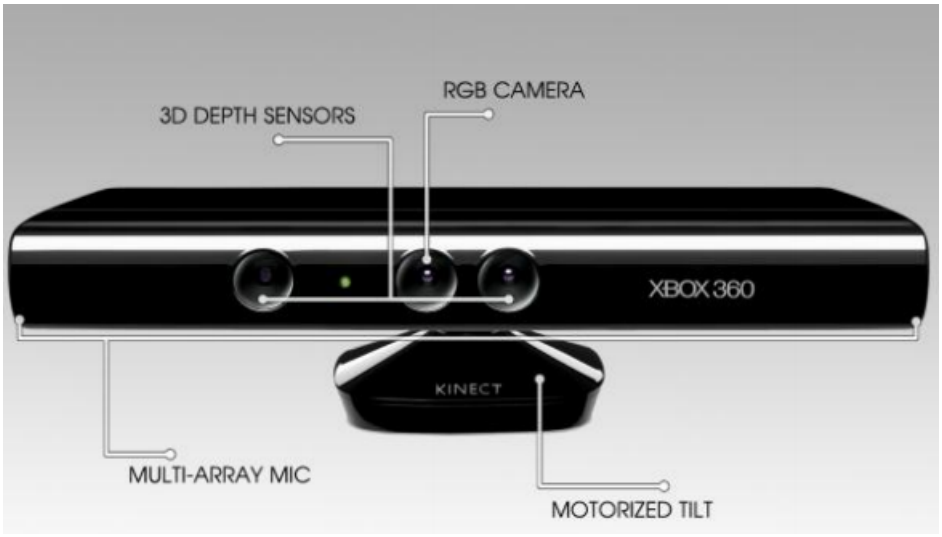


Fonte: (ROBOT, s.d.)

- Microsoft Kinect®: sensor de movimento composto por uma câmera de vídeo, sensores de profundidade 3D e um microfone. No Turtlebot 2, este sensor de movimento é responsável por toda a coleta de dados para o processamento digital de imagens, como imagens RGB e de profundidade podendo formar, a partir da combinação dos dois sistemas, o RGB-D. Com seu sensor de profundidade, o Microsoft Kinect® é capaz de fazer medições precisas em tempo real da distância entre o sensor e os objetos presentes no ambiente. Segundo Tölgyessy e Hubinský (2011), Este recurso é a maior vantagem de se usar o Kinect em aplicações voltadas a visão computacional, pois simplifica o processo de segmentação de imagens uma vez que ele utiliza informações

de distância para distinguir dois objetos com cores semelhantes. É o Microsoft Kinect® que permite ao Turtlebot 2 trabalhar com navegação, mapeamento e localização A figura 10 apresenta os componentes que estão presente neste sensor;

Figura 10 – Disposição dos sensores do Microsoft Kinect®



Fonte: (TÖLGYESSY; HUBINSKÝ, 2011)

- Computador Portátil: responsável por todo o processamento dados e controle computacional do Turtlebot 2. Este computador deve possuir o sistema operacional ROS instalado, que permite a possibilidade de controle dos motores do robô, bem como a coleta de dados dos seus sensores. Qualquer computador móvel que possua compatibilidade com o ROS pode ser utilizado, porém por razões de peso e consumo de bateria, o Turtlebot 2 utiliza por padrão um Netbook como esta unidade de controle computacional, além disto, nada impede que modificações sejam feitas a fim de que o Turtlebot 2 seja controlado por qualquer outro tipo de computador, como um Raspberry PI, que é um computador com tamanho, consumo de energia e capacidade de processamento reduzidos, mas que atende as necessidades básicas do Turtlebot 2.

4.2 Controle do Turtlebot 2

Conforme explicado na seção 4.1, o controle do robô é realizado por meio do sistema ROS. Tal controle pode ser dividido em duas partes, o de movimentos e o da aquisição de dados dos sensores.

4.2.1 Publicação de comandos

Neste trabalho, as mensagens de comando aos atuadores do robô são enviados ao nó responsável por meio de tópicos (conceito chamado de *Publish* no ROS).

Os movimentos do Turtlebot 2 podem ser controlados de duas maneiras: por meio da teleoperação (onde um ser humano controla o Turtlebot 2 remotamente) ou por meio de sua navegação autonoma que é utilizada durante o processo de patrulhamento. Para movimentar o robô de maneira manual pelo sistema robótico, mensagens do tipo "geometry_msgs::Twist" são publicadas no tópico "/cmd_vel_mux/input/teleop". Já no modo de navegação autônoma, estas mensagens são publicadas no tópico "/cmd_vel_mux /input/navi". Uma mensagem do tipo "geometry_msgs::Twist" contém as infomações de velocidade angular e linear que devem ser executadas pelo Turtlebot.

O "cmd_vel_mux" é um pacote que funciona como um multiplexador, selecionando qual mensagem deve comandar os movimentos do robô. Esta seleção é feita pela prioridade do tópico, onde mensagens recebidas no tópico "/cmd_vel_mux/input/safety_controller" são as com a maior prioridade, seguidas pelo tópico "/cmd_vel_mux/input/teleop" e, por fim, mensagens enviadas pelo tópico "/cmd_vel_mux/input/navi". Portanto, durante a execução do sistema robótico, os comandos de movimento oriundos da teleoperação possuem prioridade em relação aos comandos enviados pela navegação autônoma.

No modo de patrulhamento, as mensagens de poses de objetivos (pose de um lugar a ser patruhado) são publicadas no tópico "/move_base_simple/goal", e são então utilizadas pelo nó de controle de navegação "/amcl". Este por sua vez publica mensagens de velocidades no tópico "/cmd_vel_mux/input/navi" a fim de guiar o robô à pose solicitada.

4.2.2 Aquisição de dados

A aquisão de dados é feita por meio da inscrição em tópicos (conceito chamado de *Subscribe* no ROS). Neste método o nó do sistema robótico está inscrito em diferentes tópicos, onde são transmitidas mensagens publicadas por outro nó (ou nós). As infomações do Turtlebot 2 coletadas neste trabalho são: imagem representado no espaço de cor RGB, imagem de profundidade, pseudo laser scan (gerado a partir das imagens de profundidade do Microsoft Kinect®), sensores de toque, sensor de desnível e pose. Diferentes informações de poses também são coletadas, desde informações da pose atual do robô até poses do robô quando ele alcança algum objetivo específico.

Cada tipo de dado é controlado por um nó do ROS, por exemplo, as imagens que são coletadas pela câmera do robô são gerenciadas pelo nó "/camera/camera_nodelet_manager" ou, no caso de uma simulação, pelo nó "/gazebo". Cada nó é responsável por enviar as mensagens no tópico correspondente. Neste último exemplo, o nó da câmera publica mensagens das imagens RGB no tópico "/camera/rgb/image_raw", e publica as mensagens de profundidade no tópico "/camera/depth/image_raw". Então, nós inscritos nestes tópicos

irão receber as mensagens. Uma publicação de uma mensagem pode ocorrer de maneira periódica, como no caso das imagens da câmera, ou de maneira eventual, como ocorre com os sensores de toque.

O sistema robótico coleta a estimativa da pose do robô no tópico "robot_pose_ekf/odom_combined". Esta estimativa é gerada combinando informações coletadas de diferentes sensores em um filtro Kalman extendido. Neste trabalho estão sendo utilizadas as informações da odometria e da *Inertial Measurement Uni*t (IMU), que contém as informações do giroscópio.

Em uma mensagem deste tópico, o valor das posições "x" e "y" são um número racional. O valor da orientação também é um número racional porém está em um *range* de zero até duas vezes o PI (representando uma volta completa), onde o sinal desta variável é positivo caso o robô rotacione no sentido horário e negativo caso rotacione no sentido anti-horário. Se o robô ultrapassar um valor extremo do *range*, ele é alterado para seu outro extremo (de zero para "2*PI" e vice-versa).

O sistema de controle do Turtlebot 2 utiliza as estimativas de pose recebidas para calcular o distância percorrida e o ângulo rotacionado pelo robô. Isto é necessário pois as poses são apenas pontos no espaço, e não percursos ou valores acumulados. Para cada mensagem recebida o sistema calcula a distância entre os pontos gerados pelas coordenadas "x" e "y" e armazena o resultado no acumulador de distância percorrida. Para se calcular a rotação acumulada, é necessário converter o valor de orientação recebida pelo tópico para que o *range*, originalmente de 0 até "2*PI", seja ilimitado. Após a conversão, o valor convertido é armazenado no acumulador de ângulo rotacionado.

4.3 Mapeamento

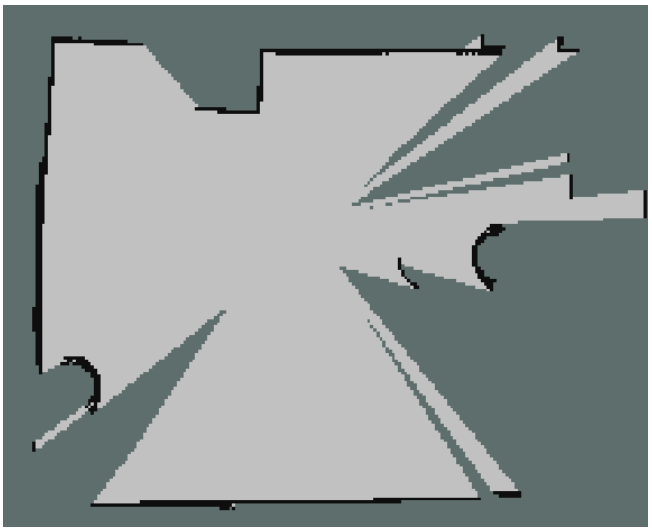
Neste trabalho, os mapas criados devem ser do tipo métrico, pois a interface gráfica deve mostrar a posição do robô no ambiente em tempo real e este tipo de mapa representa de forma mais realista o ambiente a ser monitorado.

O processo de mapeamento do sistema robótico deve solucionar o seguinte problema: mapear um ambiente interno desconhecido enquanto, simultaneamente, mantém o Turtlebot 2 localizado no mapa métrico que está sendo criado. Este problema computacional é chamado de SLAM, conforme foi explanado na seção 3.3.1 do capítulo anterior.

Para solucionar este problema foi empregado um método de mapeamento manual (o usuário deverá guiar o robô pelo ambiente) que utiliza o algoritmo GMapping. Ele foi escolhido por ser o método mais empregado pela comunidade do ROS para realizar o mapeamento 2D de ambientes e por possuir um pacote estável no ROS. Este algoritmo também atende a um dos pré-requisitos do sistema robótico deste trabalho, que é o de possuir um algoritmo "leve", ou seja, que não consuma muito processamento do computador.

No ROS o GMapping é executado pelo pacote de “gmapping”, um pacote nativo do sistema que implementa o algoritmo GMapping para gerar mapas métricos 2D de um ambiente. Seu nó de controle é chamado de “/slam_gmapping”, que é o responsável por gerar um mapa a partir de mensagens recebidas dos sensores: “pseudo laser” e da odometria (publicadas nos tópicos “/scan” e “/tf” respectivamente). Este nó publica periodicamente mensagens com o estado parcial do mapa do ambiente no tópico “/map”. Tais mensagens são do tipo “nav_msgs/OccupancyGrid” e contém os dados de um mapa parcial do tipo grades de ocupação. A figura 11 exibe um mapa parcial gerado pelo ROS gmapping.

Figura 11 – Mapa parcial criado pelo pacote gmapping



Fonte: Autor

4.4 Localização

Para localizar o Turtlebot 2 em um mapa, este trabalho utiliza um algoritmo MCL modificado chamado de *Adaptive Monte Carlo Localization* (AMCL). A diferença deste algoritmo é que ele adapta dinamicamente a quantidade de partículas necessárias para localizar o robô no ambiente (quanto maior a incerteza da posição do robô, maior o número de partículas). No ROS, o pacote responsável por localizar o robô utilizando o AMCL é o “/amcl”.

O “/amcl” utiliza as informações do “pseudo laser” do Turtlebot 2 (publicadas no tópico “/scan”), as estimativas de pose geradas pela odometria e o mapa do ambiente para estimar a localização do robô. Suas estimativas são publicadas no tópico “amcl_pose” para então serem utilizadas pelo sistema.

Ao iniciar, o “/amcl” necessita saber qual a posição do robô, e a partir desta informação ele mantém o robô localizado ao longo de seu deslocamento. No sistema robótico existem duas maneiras de informar a este nó qual a localização do robô: selecionando uma rota de patrulha ou informando manualmente esta informação por meio da interface gráfica. O

objetivo deste método é garantir que o robô sempre saiba qual sua pose inicial em uma rota de patrulha.

4.5 Navegação

Durante o patrulhamento de um ambiente, o robô se desloca, de forma autônoma, de um ponto a ser patrulhado a outro. A navegação robótica é a responsável por realizar esta atividade, guiando o robô do seu ponto de partida até seu destino. Para alcançar este objetivo o método de navegação empregado neste trabalho utiliza dois algoritmos: o A* para planejar a trajetória global a ser seguida pelo robô no mapa (deslocando este de um ponto de partida até o destino), e o DWA para garantir que o robô não colida com objetos inesperados que podem haver no meio de uma trajetória global.

No ROS, o pacote "move_base" é o responsável pela navegação autonoma de robôs móveis. Ele combina informações de outros pacotes a fim de garantir um deslocamento pelo menor percusor e de forma segura (sem colidir com o obstáculos). Dentre estes pacotes se destacam o "global_planner" (responsável pela execução do algoritmo A*) e o "base_local_planner" (responsável pela execução do algoritmo DWA).

O seu nó recebe um objetivo de deslocamento pelo tópico "move_base_simple/goal", calcula a melhor trajetória entre sua posição atual e este objetivo recebido, e controla os movimentos do robô publicando velocidades no tópico "cmd_vel_mux /input/navi" até alcaçar a pose de destino. Ao chegar, ele publica uma mensagem de sucesso no tópico "move_base/result". Entretando, se por algum motivo (como caminho bloqueado) o robô não conseguir chegar ao seu objetivo, este nó publica uma mensagem de erro. A navegação pode ser cancelada durante sua operação publicando-se um mensagem ao tópico "move_base/cancel".

4.6 Detecção de ameaças

Para realizar a detecção da presença de ameaças dentro de um ambiente interno, foi necessário identificar uma característica do ser humano que o diferencie do ambiente onde ele se encontra. A característica selecionada foi a temperatura corporal, uma vez que esta é nativa do ser humano e difícil de ser camuflada, além de ser uma variável que não demanda uma alta carga de processamento do robô.

Durante as pesquisas bibliográficas esta característica do ser humano se mostrou menos suscetível a ruídos ou falhas de leitura em comparação com outras técnicas de detecção, como a por movimento ou as que utilizam visão computacional. A maioria dos sensores de movimento comerciais hoje possuem pontos cegos o que prejudica a detecção, além disto não seria simples implementar detecção por movimento utilizando o sensor acoplado a um robô móvel, pois o robô fará com que o sensor entre em movimento. De maneira

semelhante, detecção por meio da visão computacional não é uma opção viável, pois a mesma requer utilização de uma carga relativamente elevada de processamento do robô que é limitado.

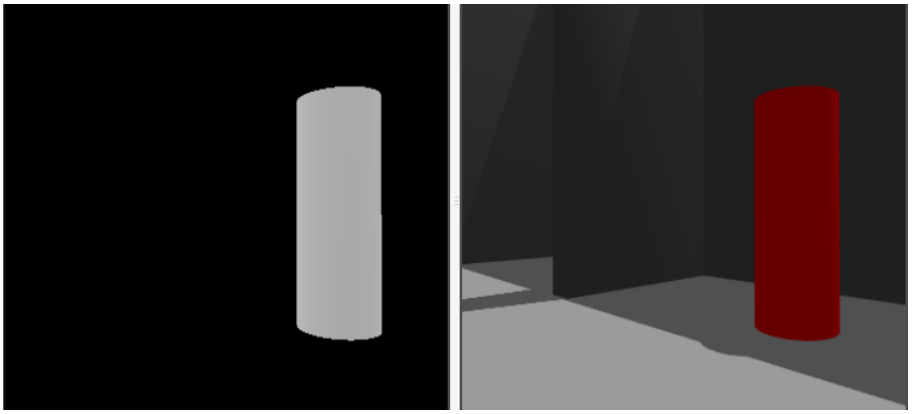
O princípio de funcionamento do método de detecção por temperatura é simples: o sensor de temperatura empregado envia para o computador do robô uma matriz com valores de temperatura capturados, onde cada posição da matriz consiste em uma área de campo de visão do sensor. Uma ameaça é detectada caso algum valor de temperatura da matriz for superior à temperatura do ambiente (definida pelo sistema).

As desvantagens da detecção por meio da temperatura estão em situações as quais a temperatura ambiente é superior a de um ser humano ou em situações quando a ameaça consegue camuflar sua temperatura por meio de roupas especiais, porém, vale lembrar que estas roupas ainda são muito caras (uma roupa que camufla somente a parte superior do corpo custa aproximadamente $3.500,00).

Para a realização de simulações do sistema robótico, foi empregado outro tipo de detecção de ameaças, pois não é possível simular valores de temperatura no simulador utilizado. Então, foi definido que as ameaças nas simulações seriam objetos cilíndricos da cor vermelha.

O método de detecção empregado neste cenário foi por meio de uma técnica de visão computacional onde são aplicadas máscaras nas imagens para selecionar os dados que serão analisados posteriormente. Neste trabalho é aplicado uma máscara inicial para remover as cores que não possuam tons de vermelho, nas imagens coletadas pelo robô. A imagem gerada por esta máscara é binária e possui valor alto nos pixels que possuam tons de vermelho. É aplicada uma operação morfológica de abertura na imagem binária a fim de remover ruídos que podem estar presentes no ambiente. Então, esta nova imagem é aplicada como uma nova máscara na imagem de profundidade capturada pela câmera do robô. Assim, a imagem resultante possui apenas os valores de distância de objetos com tons de vermelho. A figura 12 apresenta um exemplo com a imagem original ao lado da modificada.

Figura 12 – Imagem original a direita e imagem modificada a esquerda



Fonte: Autor

É feito a seguinte análise na imagem resultante: gera-se um valor em porcentagem a partir da média da soma dos valores que representam distância pela quantidade de pontos de tons vermelho na imagem. Esta porcentagem é então utilizada como um *threshold* em um comparador, sendo este o responsável por determinar uma possível ameaça. O comparador verifica se a porcentagem de pontos de tons vermelho na imagem é maior que o valor do *threshold*, caso seja, um sinal de ameça é emitido. É necessário realizar esta análise para reduzir o efeito causado pela distância de uma ameaça, pois um objeto longe do robô é percebido com um tamanho menor do seu real, impactando assim na sua análise.

4.7 Base de dados

Três base de dados são utilizadas pelo sistema robótico para armazenar informações necessárias para seu funcionamento:

1. *Users*: contém a lista dos usuários cadastrados e que possuem acesso ao sistema. Cada usuário cadastrado no sistema robótico possui nome, sobrenome, privilégio do usuário, seu *username* e *password* de acesso ao sistema;

2. *Routes*: possui uma lista de todas as rotas criadas no sistema sendo cada uma armazenada em uma tabela. Cada tabela contém todos os lugares a serem patrulhados em uma rota, e cada lugar deve conter seu nome, sua posição “x”, “y” e sua orientação;

3. *Threats*: contém a lista de todas as ameaças detectadas pelo sistema robótico. Cada rota possui uma tabela contendo a lista das ameaças detectadas durante sua execução. As informações anotadas quando uma ameaça é detectada são o horário em que ela ocorreu, o seu lugar aproximado, sua posição “x” e “y” bem como a orientação do robô no momento da detecção;

Ao longo do desenvolvimento deste trabalho, foram desenvolvidos dois métodos para gerenciar estas bases de dados:

- Armazenamento utilizando arquivos de texto: neste cenário, todos os dados são armazenados em arquivos de textos. A leitura de informações é feita por meio de uma varredura no arquivo, procurando palavras-chave, como “*Place*”, ou caracteres delimitadores de variáveis como a vírgula. Já a escrita é feita da forma serial não possuindo qualquer forma de edição posterior da informação pelo sistema robótico, apenas por meio de um editor de texto. Este método foi substituído por um que utilize um banco de dados, justamente pela dificuldade de edição das informações e também pela falta de segurança dos dados armazenados. A figura 13 ilustra um exemplo de armazenamento de uma rota por este método:

Figura 13 – Exemplo de uma rota armazenada em um arquivo de texto

```
.Use '.' to comment a line
.Please fill with the necessary information as the following template:
.Initial Pose: Place Name (pose x,pose y,orientation angle)
.Place: Place Name (pose x,pose y,orientation angle)


.HORUS PATROL - Route 0

.Description: HORUS

Initial Pose: Base (-0.00501347,-1.47459,0.56923)
Place: Main Hall (1.19904,1.52511,1.52428)
Place: Office 01 (-4.80477,9.66002,-1.79489)
Place: Closet 01 (-3.84305,-1.99505,2.27963)
Place: Office 02 (-12.2207,4.58895,-1.92306)
Place: Meeting Room 01 (-19.9668,-3.44002,1.06242)
Place: Workstation 01 (-5.06726,-5.51227,2.25918)
Place: Workstation 02 (-8.7799,-8.65442,2.20659)
Place: Corridor (-8.87267,-3.89758,-2.93693)
Place: Meeting Room 02 (-12.2104,-8.77358,2.95355)
Place: Reception 01 (-16.3894,-11.5562,-1.6649)
Place: Storeroom (-26.2378,-13.1952,0.874406)
Place: Reception 02 (-21.4005,-15.2546,3.1091)
Place: Exposition Room 01 <p1> (-23.1905,-22.9088,-1.47986)
Place: Exposition Room 01 <p2> (-18.4446,-26.3426,0.552656)
Place: Exposition Room 01 <p3> (-18.7031,-19.9991,-0.682444)
Place: Corridor 02 (-15.0464,-19.9141,2.17435)
Place: Meeting Room 02 (-6.60935,-14.592,-2.46878)
Place: Study Room 01 (4.68951,-9.46999,0.945499)
Place: Workstation 03 (6.89321,-0.605631,-0.935)
Place: Workstation 04 (3.57729,-2.65078,-1.9987)
Place: Closet 02 (-0.418946,-6.90698,1.19911)
```

Fonte: Autor

- Armazenamento utilizando banco de dados: este método foi empregado para resolver as limitações do método anterior. Neste cenário, os dados são armazenados em arquivos gerenciados pelo MySQL®, que é um sistema de gerenciamento de banco de dados (SGBD) *open-source*. Este gerenciador possibilita que um usuário com privilégios de administrador gerencie com facilidade as informações contidas no banco de dados pela interface do sistema robótico, sendo possível, por exemplo, criar, remover e editar informações contidas em uma rota. Além desta vantagem, com este SGBD é possível criptografar informações contidas na base de dados, como as senhas de cada usuário. A Figura 14 apresenta o formato das base de dados utilizadas.

Figura 14 – Exemplo de base de dados utilizadas no sistema robótico

Index	Name	Position X	Position Y	Orientation
0	Initial Pose	-0.4	-1.7	0.54855
1	Main Hall	1.4	1.7	1.68145
2	Office 01	-4.6	9.9	-1.79551
3	Closet 01	-3.8	-2.2	2.2143
4	Office 02	-12	5	-2.56522
5	Meeting Room 01	-19.5	-2.7	0.974597
6	Reception 01	-8.9	-3.8	-2.83708
7	Workstation 01	-5.4	-5.6	1.82598
8	Workstation 02	-8.4	-8.1	2.52134
9	Reception 02	-16.7	-11.7	1.13246
10	Meeting Room 02	-12.1	-9.4	2.64225

Fonte: Autor

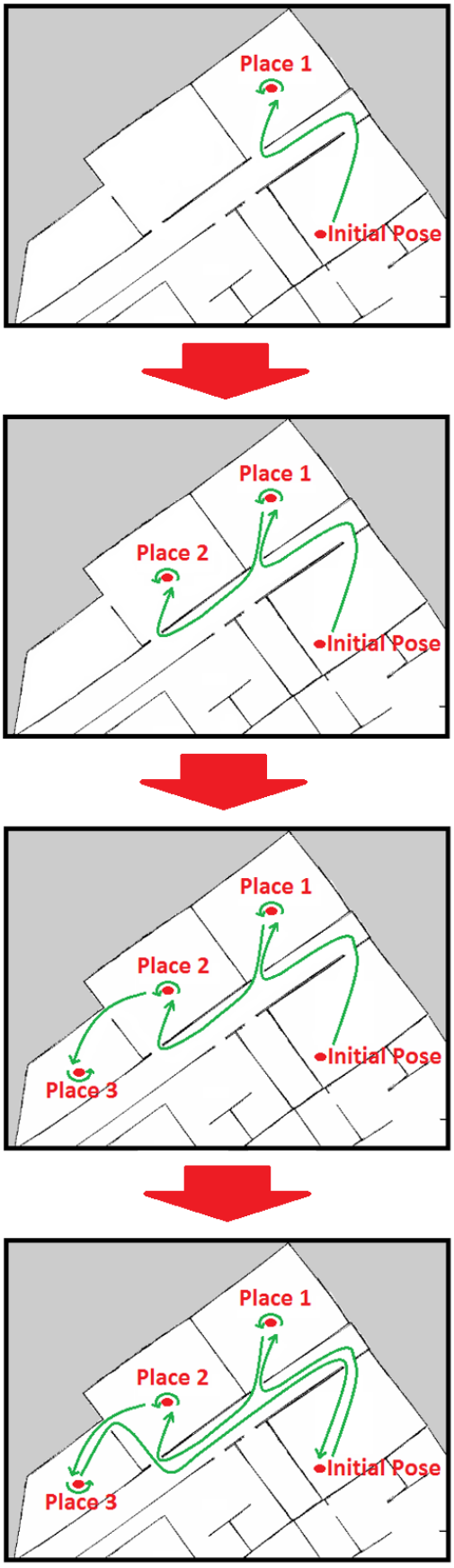
4.8 Sistema robótico

O sistema robótico é o responsável por realizar a vigilância de um ambiente interno por meio de patrulhas. Neste sistema, um robô móvel executa uma patrulha pré-definida pelo usuário em busca de ameaças. Para realizar esta tarefa, o robô possui um mapa do ambiente a ser monitorado e utiliza um método de navegação autônoma.

Antes de explicar como funciona o método de patrulhamento definido, é necessário que se entenda o que é uma rota de patrulha. Uma rota é definida por uma sequência de pontos a serem patrulhados (chamados de "*Places*"), e um ponto inicial (chamado de "*Initial Pose*").

O processo de patrulha começa com o robô posicionado na "*Initial Pose*", ele se desloca para cada ponto definido pelo usuário como um lugar a ser patrulhado (*Place* 1,2,3). Quando o robô chega em um "*Place*", ele rotaciona em torno do seu eixo até completar uma volta a fim de procurar por uma ameaça no lugar. Após isto, o robô se desloca para o próximo "*Place*", repetindo o processo até acabar os lugares definidos na rota. Quando o robô finaliza uma rota ou quando recebe uma ordem para cancelar a mesma, ele retorna para o ponto inicial e aguarda por novas instruções da central de comando. A figura 15 ilustra este método descrito em um exemplo de uma sequência de patrulha de uma rota composta por três lugares.

Figura 15 – Exemplo de uma sequência de patrulha



Fonte: Autor

Somente um usuário com privilégios de administrador tem permissão de gerenciar as rotas do sistema. Um administrador pode criar, apagar ou modificar uma rota, alterando a ordem da execução dos pontos de patrulha ou editando os dados contidos nestes pontos. Um usuário sem este tipo de privilégio pode apenas selecionar uma rota da lista, e executar sua sequência de patrulhamento.

Conforme explanado na secção 4.7, as rotas são armazenadas em uma base de dados controlada pelo sistema robótico, chamada de "*Routes*". Durante a execução de uma sequência de patrulha no modo de patrulhamento do sistema, uma das tabelas da base de dados é selecionada. Uma tabela contém todos os atributos de uma única rota, onde cada linha contém as informações de um "*Place*", onde estes, por sua vez, são lidos linha a linha. A base de dados "*Routes*" pode conter várias tabelas, ou seja, várias rotas, porém apenas uma pode ser selecionada e executada por vez.

Ao longo de uma rota, quaisquer ameaças detectadas são capturadas (pela câmera de vídeo) e enviadas em forma de imagem para uma central de comando, junto com a localização atual do robô, onde um ser humano tomará as devidas ações. Após uma detecção, o robô continua a execução da sequência contida na rota sem qualquer alteração no seu comportamento a fim de não chamar atenção, pois o Turtlebot 2 é um robô que pode facilmente ser confundido com um robô aspirador de pó. Além das informações das detecções, o robô também envia imagens capturadas nos pontos de interesse para registro e futuras análises.

De modo semelhante às rotas, as detecções também são armazenadas em uma base de dados que é gerenciada por um usuário, onde este pode apagar um falso positivo ou todo o histórico. As informações capturadas quando uma ameaça é detectada são a imagem da câmera no momento da detecção, o horário em que ela ocorreu, o seu lugar aproximado, sua posição "x" e "y" bem como a orientação do robô no momento da detecção. A determinação do lugar aproximado é definido como sendo o "*Place*" mais próximo da posição atual do robô. Este método não é preciso, uma vez que a trajetória ou o ambiente podem assumir qualquer forma, mas ele serve para dar ao vigilante um entendimento da área que ocorreu a detecção.

O sistema robótico conta outras funções auxiliares à de patrulhamento:

- Mapeamento manual: função para criar, de forma manual, um mapa métrico de um ambiente interno a ser vigiado;
- Transmissão em tempo real de vídeo: transmite em tempo real o que a câmera do Turtlebot 2 está vendo;
- teleoperação do robô: possibilita que um vigilante na sala de comando controle os movimentos do robô remotamente. Esta função em conjunto com a transmissão de imagens em tempo real se torna uma importante ferramenta, permitindo que um vigilante investigue ameaças sem se expor à riscos;

O sistema é constituído fisicamente de três componentes principais: a sala de comando, o robô Turtlebot 2 e a interface gráfica do usuário.

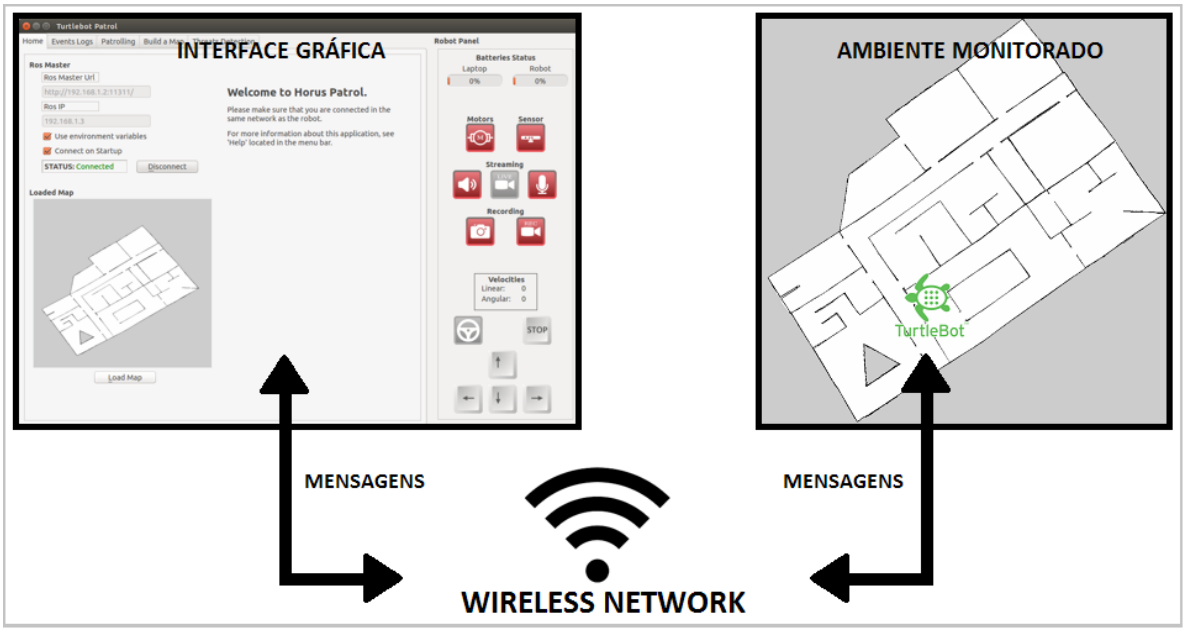
A sala de comando é definida como sendo um ambiente seguro que não necessita ser patrulhado. Neste local está localizado a estação de trabalho (computador) que executa a interface gráfica, sendo esta por sua vez, monitorada por um vigilante humano. Este local pode até estar presente no mapa da área a ser patrulhada, porém não pode ser representada por um "*Place*" de uma rota.

O Turtlebot 2 é o robô responsável por executar o papel de um vigia no sistema. Ele realiza as patrulhas de forma autônoma procurando por ameaças. Seus principais objetivos são o de executar o trabalho com uma performace próxima da que é alcançada por um ser humano, e de servir como uma extensão das capacidades físicas do mesmo, pois por meio da teleoperação em conjunto com a transmissão de imagens em tempo real permite que um vigilante esteja virtualmente presente no ambiente sendo patrulhado.

A interface gráfica é um programa desenvolvido na linguagem C++ utilizando uma programação orientada a objetos, com o objetivo de permitir que um vigilante na sala de comando interaja com o Turtlebot 2. Ela é responsável por realizar a gestão do sistema robótico possibilitando a seleção dos modos de patrulhamento, mapeamento manual, teleoperação, e gerenciamento: de rotas, de usuários e das ameaças encontrada. Além desses modos de operação, esta interface gráfica possui diversas ferramentas que auxiliam um vigilante a interagir livremente com o ambiente onde o Turtlebot 2 se encontra, como transmissões em tempo real de vídeo ou áudio, e capturas de imagens ou vídeos do ambiente.

Para realizar a comunicação entre a interface gráfica e o robô móvel, é utilizado um método que funciona da seguinte maneira: a interface gráfica, executada em um computador localizado na sala de comando, se comunica com o Turtlebot 2 por meio de uma rede *wireless* utilizando a tecnologia TCP/IP, onde cada componente possui um endereço IP. A interface coleta as informações capturadas pelo Turtlebot 2 e envia ações ao mesmo, o Turtlebot 2 envia o status das suas variáveis mais importantes além das anomalias detectadas, tudo ocorrendo em tempo real. Exemplos deste cenário seriam quando o robô envia uma detecção de ameaça à interface gráfica ou quando a interface gráfica coleta imagens da câmera publicadas pelo robô. A figura 16 ilustra a arquitetura de comunicação que foi descrita deste sistema.

Figura 16 – Arquitetura de comunicação do sistema robótico



Fonte: Autor

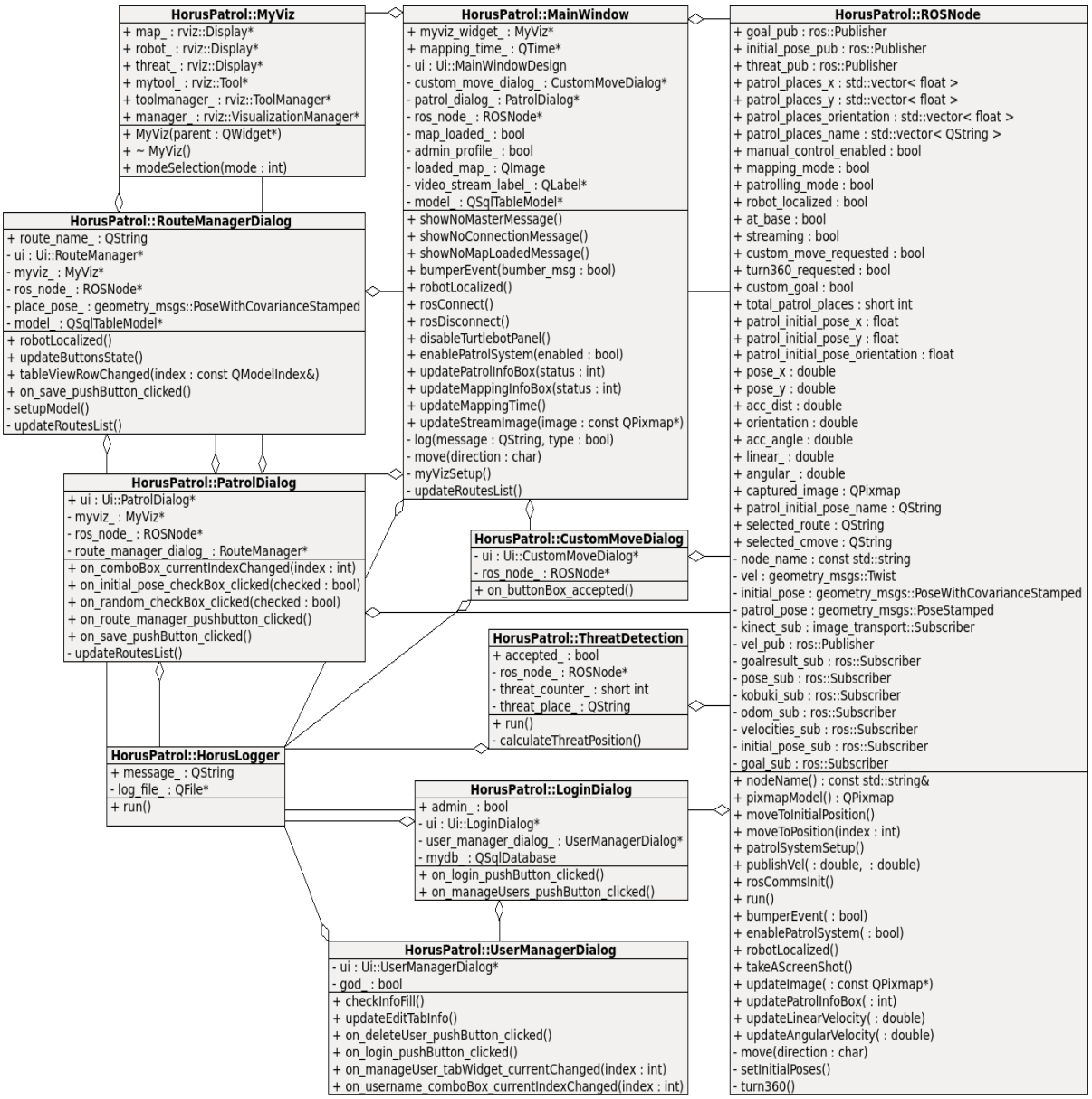
4.8.1 Classes

O sistema robótico foi projetado a partir de uma representação de diagrama de classes, uma vez que o sistema utiliza uma programação orientada a objeto. Sendo assim, esta representação foi utilizada como base para o desenvolvimento do programa de controle do sistema robótico. As classes foram criadas de acordo com os modos de operação do sistema robótico, com as janelas de diálogo ou função dentro do sistema. Para cada classe foram definidos seus atributos e métodos básicos para o seu funcionamento, bem como sua visibilidade.

O sistema final possui no total dez classes, onde sete fazem parte da interface gráfica (responsáveis por controlar o comportamento das janelas e caixas de diálogo), e as outras três constituem nas *threads* do sistema (responsáveis pelo controle geral das ações do robô). Uma *thread* é uma rotina executada concorrencialmente, ou seja, é um processo executado em paralelo com outras *threads* e/ou com a interface gráfica do sistema robótico. Utiliza-se uma *thread* quanto se é necessário executar uma rotina sem que haja interrupções da mesma, como no caso de uma detecção de uma ameaça.

A figura 17 ilustra uma representação resumida do diagrama de classes do sistema robótico. Este diagrama resumido apresenta os principais atributos e métodos de cada classe, bem como as relações entre as classes.

Figura 17 – Diagrama de classes resumido



Fonte: Autor

4.8.1.1 ROSNode

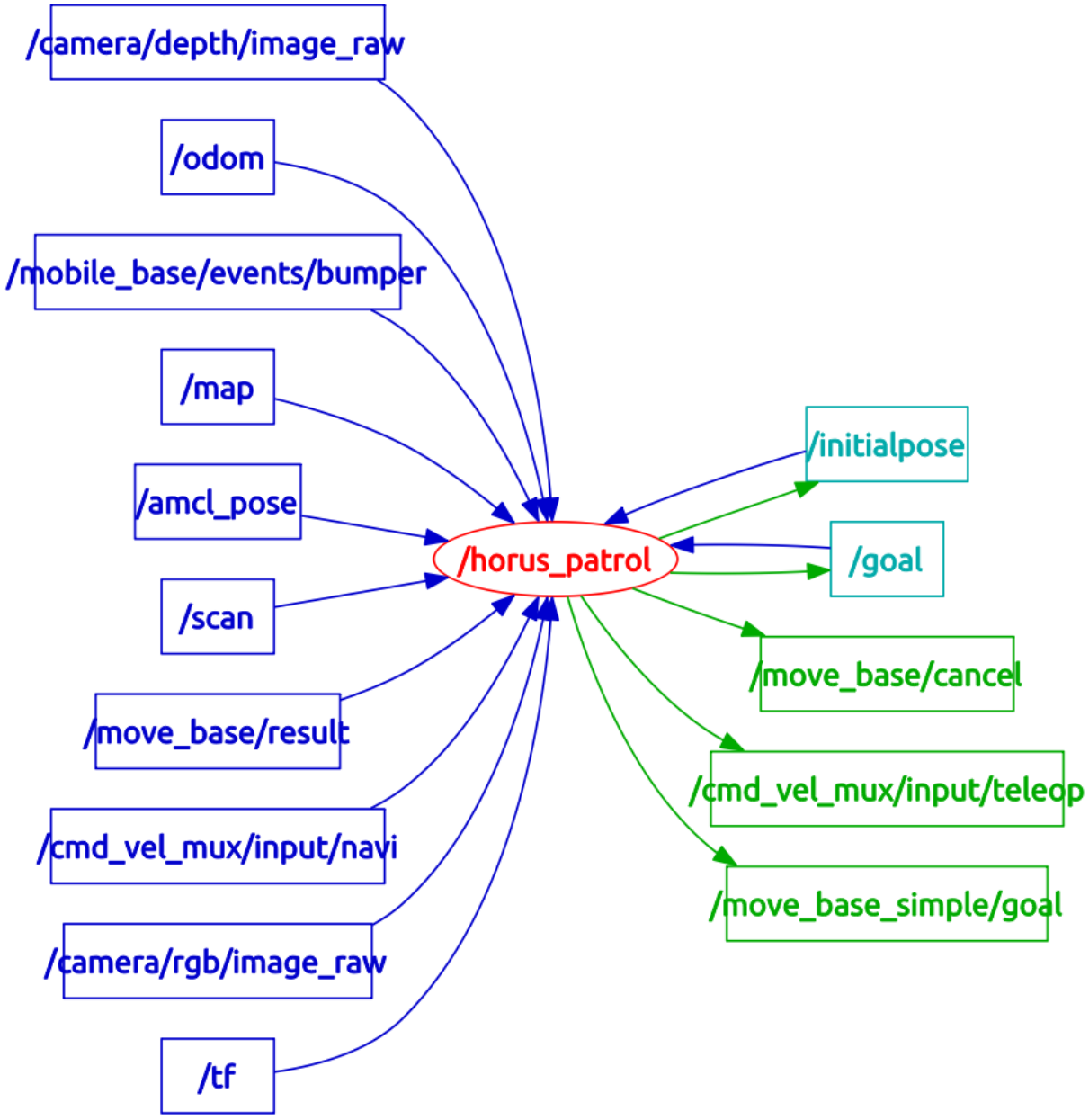
Esta é a classe principal do sistema robótico, atuando como o nó do ROS do sistema robótico. Ela é executada no ROS instalado no computador do Turtlebot 2 e é responsável por controlar o robô (conforme foi explanado na seção 4.2), por permitir que a interface interaja com o Turtlebot 2, por detectar as ameaças no modo simulado e por gerenciar os nós responsáveis pelo mapeamento, localização e navegação.

O nó da classe ROSNode é chamado de "horus_patrol". Este nó está inscrito em nove tópicos e publica mensagens em outros cinco, totalizando uma interação com catorze tópicos. Dentre os tópicos que o nó está inscrito, destacam-se os que transmitem mensagens: da câmera (RGB, profundidade), do pseudo-laser (gerado a partir das imagens de

profundidade), do nó responsável por estimar a pose do robô, dos sensores de toque do robô. Quanto aos tópicos mais relevantes onde o "horus_patrol" publica mensagens são nos: de movimento do Turtlebot 2, de definição de objetivo de deslocamento, de definição da posição inicial.

A figura 18 ilustra a interação do nó "horus_patrol" com os tópicos do ROS em execução no sistema robótico. Este tipo de representação é chamada de ROS *graph* onde é possível visualizar, em um gráfico, como está configurada as trocas de informações entre os nó em execução no ROS, e quais tópicos eles estão utilizando para transmitir suas mensagens. No apêndice A é apresentada o ROS *graph* do sistema robótico completo com todos os nós e tópicos utilizados.

Figura 18 – ROS *graph* somente do nó horus_patrol



Fonte: Autor

Neste gráfico, a elipse vermelha no centro da figura é a representação do nó horus_patrol, os retângulos na cor azul são os tópicos que o horus_patrol está recebendo mensagens, ou seja, está inscrito. Os retângulos em verde são os tópicos que o horus_patrol publica mensagens. Por fim, os retângulos na cor teal são tópicos em que o horus_patrol publica informações e está inscrito. As flechas indicam a direção do fluxo das mensagens enviadas.

A "ROSNode" além de controlar o Turtlebot 2, é responsável por inicializar e finalizar os nós responsáveis pelo mapeamento, e pela localização e navegação. Esta classe também troca mensagens com estes nós, onde "ROSNode" recebe, via tópico, a pose do robô em relação ao mapa, a posição inicial quando modificada e uma mensagem informando quando o robô chega ao seu objetivo.

No modo simulado, é esta classe a responsável por fazer a detecção de ameças por meio da cor vermelha, conforme foi descrito na secção 4.6. A metodologia adotada foi que para cada mensagem de uma imagem recebida pela câmera do robô, a "ROSNode" execute o processo explicado na secção 4.6.

Por fim, a classe "ROSNode" é responsável por atuar como uma ponte de ligação entre a interface gráfica e o robô. Neste cenário, a interface gráfica acessa as informações do robô por meio dos atributos da ROSNode", de modo semelhante, a interface pode modificar informações contidas nesta classe. Portanto, a interface jamais irá coletar informações diretamente do robô ou comandá-lo.

4.8.1.2 ThreatDetection

O comportamento do robô mediante a uma detecção de uma possível ameaça é controlado por meio desta classe, que possui uma rotina específica a ser seguida em casos de detecção. Ela é executada no ROS instalado no computador do Turtlebot 2.

Durante a execução desta rotina, as informações da pose do robô, da área aproximada onde o robô se encontra no mapa, e a hora do acontecimento são capturadas e estes atributos são armazenados em uma base de dados para serem utilizados para futura análises. Além disso, é feito a coleta da imagem capturada pela câmera no momento do acontecimento.

Esta *thread* pode ser chamada de duas maneiras diferentes, dependendo do modo com que o sistema robótico esteja sendo executado. Na primeira maneira, a "ThreatDetection" é chamada por uma função *callback* no instante que o nó "horus_patrol" recebe uma mensagem informando nível alto da porta de entrada do Turtlebot 2 onde está conectado o sensor de temperatura. Este método é empregado quando o sistema estiver sendo executado em um ambiente real. Já na outra maneira esta *thread* é chamada quando a classe "ROSNode" detecta uma ameaça pela sua cor. Este segundo método é utilizado quando o sistema está sendo executado em um ambiente virtual,

4.8.1.3 HorusLogger

Esta classe é uma *thread* que atua como um *logger* de eventos do sistema, sendo responsável por armazenar determinados acontecimentos, identificados previamente como importantes. Ela é executada no ROS instalado no computador do Turtlebot 2.

Alguns exemplos destes acontecimentos a serem registrados: criação, remoção e acesso de usuários, rotas criadas e deletadas, ameaças detectadas, sequências de rota iniciadas e finalizadas, dentre outros eventos. Junto das informações a serem registradas, deve ser incluído a hora e a gravidade do evento.

O objetivo de se realizar este armazenamento de eventos é o de se obter um histórico dos acontecimentos mais importantes que ocorreram no sistema robótico e utilizar tais informações coletadas para futuras análises, como o tempo médio de execução de uma rota ou os usuários que mais utilizam o sistema.

4.8.1.4 MainWindow

A "MainWindow" é a principal classe da interface gráfica do sistema robótico, sendo responsável por realizar a gestão da janela central desta interface gráfica. Ela é executada no ROS instalado no computador da estação de trabalho que fica na sala de comando.

A janela que esta classe controla é organizada de modo que seja possível acessar qualquer modo de operação do sistema robótico com praticidade. Ela possui uma área específica que agrupa todos os botões de comando direto do robô, como o botão de acionamento de sua câmera, bem como uma área para cada modo de operação do sistema, como o de patrulhamento ou construção de mapas. Portanto, esta janela funciona como um ponto de acesso aos modos de operação e às funções do sistema.

Desse modo, a função da "MainWindow" é controlar o comportamento desta janela, definindo as ações de seus objetos como botões, abas, ou opções da barra de menu. Esta classe também é responsável por chamar outras classes que controlam as caixas de diálogo.

Além de definir as ações dos objetos, está classe é responsável por definir a configuração de rede de comunicação do sistema (que pode ser uma rede local virtual ou via Ethernet TCP/IP), por controlar qual mapa de ambiente esta selecionado (para ser utilizado no modo de patrulhamento), por realizar o gerenciamento dos *logs* de eventos do sistema e das ameças detectadas pelo mesmo.

4.8.1.5 PatrolDialog

Esta classe é chamada pela "MainWindow" e é responsável por controlar o comportamento da caixa de diálogo de seleção de rotas. Ela é executada no ROS instalado no computador da estação de trabalho que fica na sala de comando.

Dentre suas funções está a de: permitir que um usuário selecione uma rota contendo uma sequência de patrulha a ser percorrida pelo robô; permitir que um usuário com

privilégios de administrador acesse a janela de gerenciamento de rotas, onde esta por sua vez é controlada pela classe "RouteManagerDialog"; permitir que um usuário substitua a pose inicial do robô definida na rota, pela pose atual do robô; possibilitar que uma rota seja selecionada de forma aleatória; exibir um resumo das informações contidas na rota que esteja selecionada.

4.8.1.6 RouteManagerDialog

Esta classe é chamada pela "PatrolDialog" e é responsável por controlar o comportamento da caixa de diálogo de gerenciamento de rotas. Ela é executada no ROS instalado no computador da estação de trabalho que fica na sala de comando.

Conforme foi descrito no início desta secção, um usuário (com privilégios de administrador) pode modificar uma rota, alterando a ordem da execução dos pontos de patrulha, criando ou apagando uma rota, editando os dados contidos em uma rota e, por fim, pode até renomeá-la. Um usuário com um tipo de privilégio inferior ao de administrador pode apenas selecionar uma rota da lista, e executar sua sequência de patrulhamento.

Esta classe acessa a base de dados e adiciona, remove ou altera rotas (ou "*Places*") após um administrador finalizar as alterações e requisitar que elas sejam salvas.

4.8.1.7 LoginDialog

A "LoginDialog" é a classe responsável por realizar o controle de acesso ao sistema robótico e controlar o comportamento da janela de *login*. Ela é executada no ROS instalado no computador da estação de trabalho que fica na sala de comando.

Ao iniciar o aplicativo do sistema, uma janela de *login* aparecerá, e por meio desta um usuário deve informar um *username* e senha válidos, ou seja, um *username* que esteja cadastrado na base de dados do sistema. O papel desta classe é validar se as informações de *username* e senha inseridas pelo usuário estão cadastradas em algum usuário da base de dados. Caso as informações sejam válidas, a "LoginDialog" permite que o usuário acesse o sistema robótico com seu respectivo privilégio.

Está classe também permite que um usuário acesse o gerenciador de usuários.

4.8.1.8 UserManagerDialog

Para realizar o controle da caixa de diálogo de gerenciamento de usuários, utiliza-se a classe "UserManagerDialog". Esta classe é chamada pela "LoginDialog" e também é executada no ROS instalado no computador da estação de trabalho que fica na sala de comando.

O gerenciamento de usuários funciona de maneira semelhante ao processo que ocorre com o gerenciamento de rotas. Um usuário (com privilégios de administrador) pode criar ou remover um usuário, e modificar suas informações como sua senha. Um usuário com

um tipo de privilégio inferior ao de administrador não tem acesso a esta funcionalidade do sistema.

Esta classe acessa a base de dados e adiciona, remove ou altera as informações de usuários após um administrador finalizar as alterações e requisitar que elas sejam salvas.

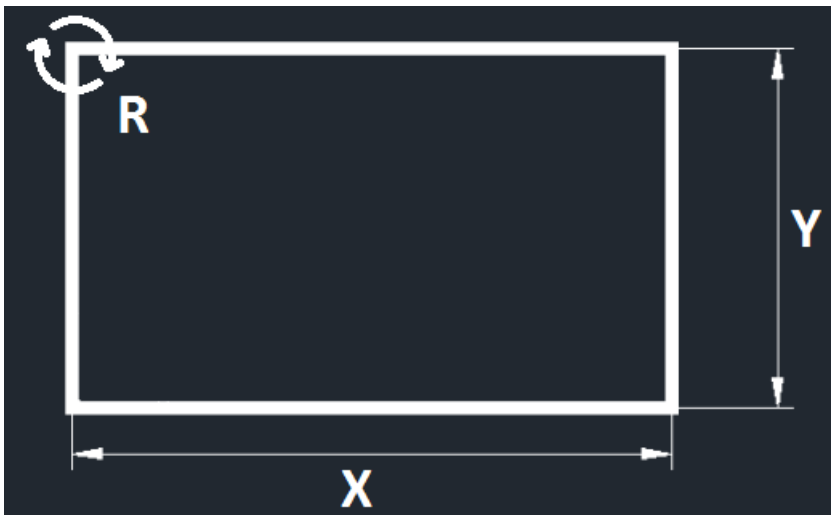
4.8.1.9 CustomMoveDialog

Esta classe tem como objetivo controlar a caixa de diálogo que permite a execução de movimentos retangulares pré-programados de forma automática. Ela é executada no ROS instalado no computador da estação de trabalho que fica na sala de comando.

Esta execução de movimentos de forma automática é uma função auxiliar no processo de mapeamento e tem como objetivo o de reduzir o tempo gasto por um usuário controlando o robô manualmente para gerar um mapa do ambiente, pois o tempo total necessário para criar um mapa é muito alto.

As informações coletas pela classe são as dimensões do retângulo, o sentido de rotação do movimento (horário/anti-horário), o número de vezes que este retângulo deverá ser percorrido, e se será necessário realizar rotações extras em cada vértices. Após coletá-las, a "CustomMoveDialog" envia um comando para a "ROSNode", que controlará a execução do movimento do Turtlebot 2. A figura 19 ilustra os parâmetros básicos para a execução deste método de movimentação, onde "X" e "Y" são as dimensões do retângulo e "R" é o sentido de rotação do movimento.

Figura 19 – Atributos básicos necessários para a execução do retângulo.



Fonte: Autor

4.8.1.10 MyViz

A "MyViz" é a classe que possibilita a visualização, em um ambiente simulado 3D, da pose do robô em relação ao mapa selecionado durante a execução de uma sequência

de patrulha ou durante a construção de um mapa. Ela é executada no ROS instalado no computador da estação de trabalho que fica na sala de comando.

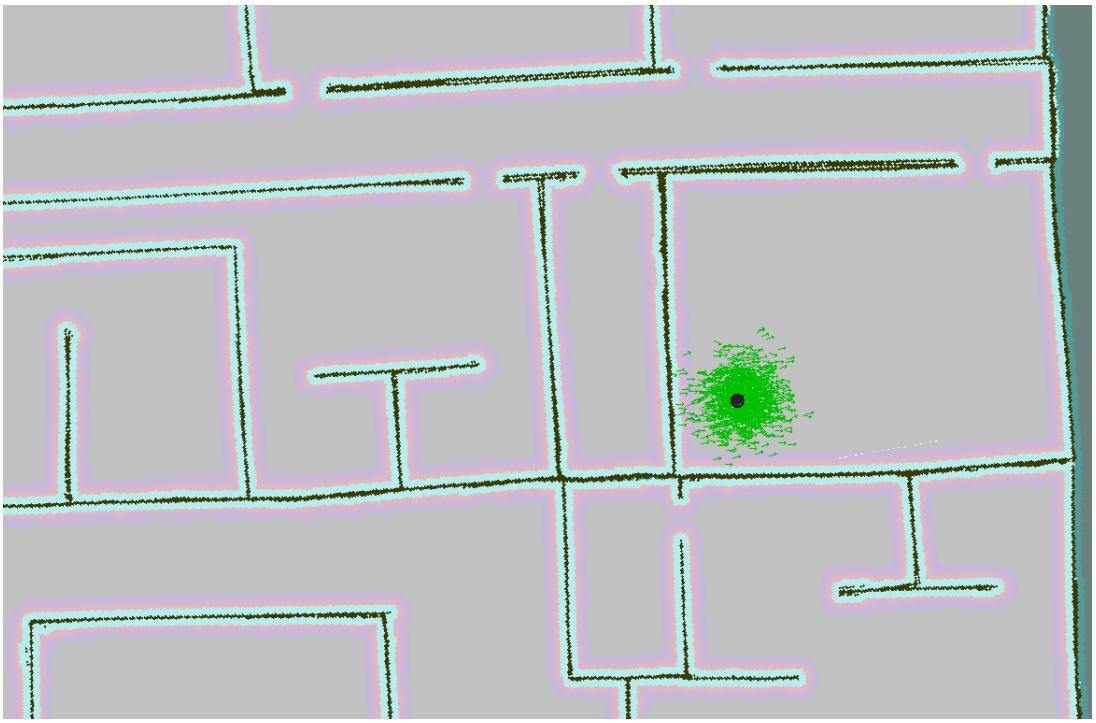
Esta classe não controla uma caixa de diálogo ou é uma *thread*, ela é uma classe do tipo "RenderPanel" do pacote Rviz. Este pacote do ROS gera um ambiente tridimensional para visualizações do comportamento de robôs (móveis ou manipuladores). A interface gráfica do sistema robótico utiliza parte deste pacote para gerar um ambiente com o Turtlebot 2 dentro do mapa selecionado, porém ela não consegue mostrar objetos não mapeados que possam surgir em um ambiente real. Com este método é possível exibir a posição em tempo real do Turtlebot 2 no mapa selecionado.

A "MyViz" também é empregada no gerenciamento de rotas, porém não com o objetivo de mostrar a pose atual do robô, e sim mostrar a pose de um "*Place*" pertencente a uma rota. Este método foi selecionado para melhorar a visualização de um "*Place*", pois é uma tarefa difícil para um ser humano imaginar uma pose em relação a um mapa apenas com os valores numéricos das posições x, y e a orientação do robô.

O Rviz é executado como um aplicativo e ele permite que seja montado um ambiente a ser visualizado por meio da seleção de modelos e de tópicos para serem lidos. Entretando, neste projeto é utilizado apenas parte das classes contidas no Rviz, como o "RenderPanel" (gera o ambiente) e "VisualizationManager", que gerencia o "RenderPanel".

A figura 20 apresenta um exemplo de ambiente, utilizado no sistema robótico, gerado pelo pacote Rviz, exibindo os tópicos selecionados, o mapa do ambiente a ser patrulhado e o modelo do robô Turtlebot 2.

Figura 20 – Pacote Rviz em execução no sistema robótico



Fonte: Autor

5 RESULTADOS OBTIDOS

Neste capítulo serão apresentados, de maneira detalhada, os resultados obtidos após o desenvolvimento do sistema robótico proposto neste trabalho. Tais resultados são analisados por meio de diferentes tipos de testes a fim de avaliar as funcionalidades e viabilidade deste sistema robótico denominado de Horus Patrol.

O nome Horus Patrol traduzido para o português significa "Patrulha de Horus", onde Horus é o nome romano dado a um deus da mitologia egípcia. Segundo a mitologia, após uma batalha contra Seth, Horus perdeu seu olho esquerdo que foi remontado pelos deuses e devolvido a ele. Este novo olho, chamado de "Udyat" dava a Horus a habilidade de ver além de um olho comum. Atualmente, o olho de Horus é usado como amuleto de proteção contra perigos e é assimilado ao "olho que tudo vê". Por esta razão o sistema foi nomeado de Horus Patrol.

Este sistema robótico foi implementado utilizando a linguagem C++ em uma programação orientada a objetos. A sua interface gráfica foi utiliza o *framework* de código livre Qt e, conforme explanado no capítulo anterior, o controle do robô é realizado por meio do sistema de código livre ROS.

Cada classe é formada por um arquivo de cabeçalho (formato .hpp) e um arquivo de código fonte (formato .cpp). Esta separação é feita para se diferenciar a declaração da implementação de uma classe. No cabeçalho são declarados os atributos e métodos de uma classe, enquanto que no código fonte são definidos o funcionamento de cada método.

A seguir, são apresentados os resultados obtidos com o desenvolvimento da interface gráfica, do nó de controle e dos métodos de: detecção de ameaças, mapeamento, localização e navegação.

5.1 Interface gráfica

A interface gráfica é composta de uma janela principal e outras cinco janelas auxiliares, as caixas de diálogo, onde cada uma é representada por uma classe e possui uma função específica. A janela principal apresenta os principais dados do Horus Patrol e possibilita que um usuário acesse todas as funcionalidades do sistema, como o modo de patrulhamento. As caixas de diálogo auxiliam no processo de configuração da interface, exibindo e coletando, do usuário, informações específicas de uma funcionalidade. Umas das caixas de diálogo presente nesta interface gráfica é a responsável por coletar os dados de uma rota criada pelo usuário.

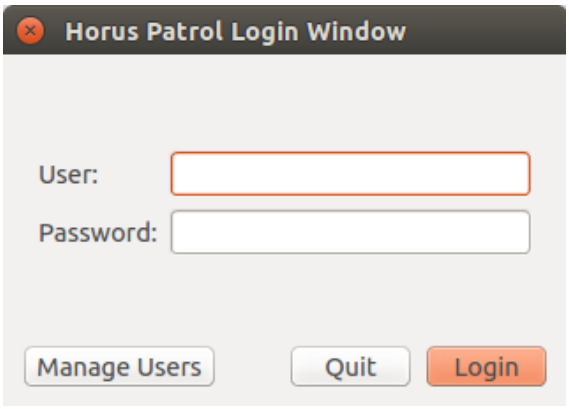
É importante ressaltar que cada janela ou caixa de diálogo possui uma classe que a controla. Uma janela serve apenas como o meio de um usuário transmitir dados para uma classe de controle. É a classe que processa o comportamento do sistema robótico quando

um usuário aperta um botão ou insere um dado em uma caixa de texto. A interface gráfica pode ser dividida em quatro setores: controle de acesso, a janela principal, gestão e seleção de rotas, e janelas auxiliares.

5.1.1 Controle de acesso

Ao iniciar a interface do Horus Patrol, uma caixa de diálogo de *login* é chamada. Sua função é a de capturar os dados de *username* e senha para que a classe de controle de acesso "LoginDialog" valide o *login*. A figura 21 ilustra esta caixa de diálogo.

Figura 21 – Janela de *login*



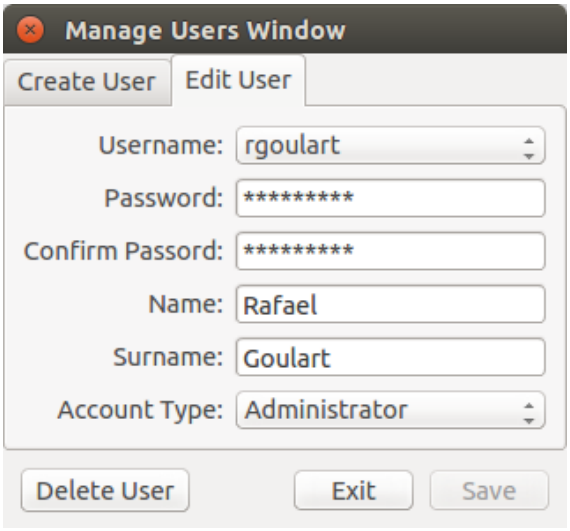
Fonte: Autor

Esta janela possui um botão, no canto inferior esquerdo chamado de "*Manage Users*", que da acesso à janela de gerenciamento de usuários. Ao apertar este botão uma nova janela de login aparecerá, porém nesta apenas usuários cadastrado na base de dados com privilégio de administrador serão aceitos.

Na janela de gerenciamento de usuários, ilustrada na figura 22, um administrador pode criar, apagar ou editar um usuário. Esta janela está dividida em duas abas, uma para criar um novo usuário e outra para editar e apagar um usuário selecionado. Cada usuário deve possuir os seguintes atributos: um nome, um sobrenome, um tipo de privilégio (usuário ou administrador) um *username* e uma senha. Destes atributos, o único que não pode se repetir é o *username*, ou seja, não podem haver dois usuários com o mesmo nome de *username*.

Um usuário administrador tem permissão de apagar ou alterar as informações de outro administrador e de modificar seus próprios dados. O objetivo disto é o de dar o máximo de liberdade aos administradores e evitar problemas, como esquecimento. Vale ressaltar que qualquer alteração nos usuários é salva em um arquivo de *log*.

Figura 22 – Janela de gerenciamento de usuários



Fonte: Autor

5.1.2 Janela principal

Após um usuário efetuar o *login* com sucesso, a janela principal do Horus Patrol é iniciada junto com sua classe "MainWindow". Esta janela é composta de duas partes: um *dashboard* do Turtlebot 2 e uma área de abas.

O *dashboard* é um painel que agrupa todos os objetos referentes ao controle manual dos dispositivos do robô, como os botões de acionamento dos motores e câmera. Este painel também exibe o status de algumas variáveis do robô, como os indicadores de velocidade angular e linear. É por meio deste painel que um usuário pode teleoperar o Turtlebot 2.

A área de abas contém os *widgets* que interligam as funcionalidades do Horus Patrol à janela principal. Esta janela está dividida em quatro abas, a "*Home*", a "*Events Logs*", a "*Patrolling*", a "*Build a Map*" e a "*Threads Detection*".

A "*Home*" é a aba padrão da interface, ou seja, é a janela mostrada assim que um usuário efetua um login com sucesso. Ela disponibiliza as informações básicas para o funcionamento do sistema, como as configurações de rede e o mapa carregado para execução.

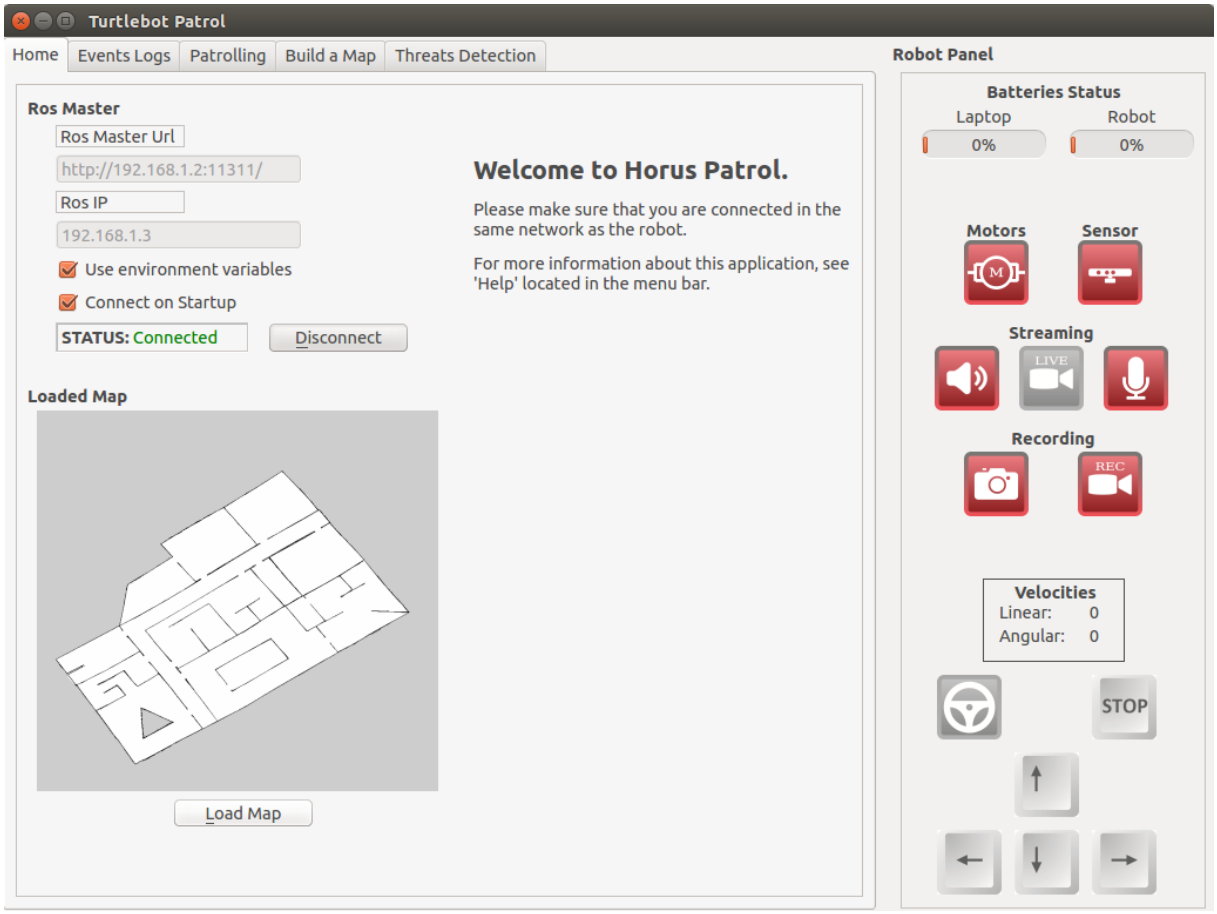
Conforme já foi explanado no capítulo anterior, existem dois métodos de comunicação entre a interface gráfica e o Turtlebot 2. O primeiro método utiliza uma rede *wireless* Ethernet TCP/IP, já o segundo utiliza uma rede local para realizar esta comunicação com o robô.

A área de configuração de rede está localizada no canto superior esquerdo da aba *Home*, ela possui um campo para inserir o IP do Mestre do ROS e outro para inserir o IP do robô. Estes são os parâmetros necessários para realizar a comunicação via rede *wireless*. Para utilizar uma comunicação via rede local, basta marcar a caixa da opção "*Use environment variables*". Com esta opção marcada, o computador da interface gráfica não se comunicará com outros computadores e executará a função de Mestre do ROS.

Por fim, se o usuário preferir, ele pode marcar a caixa *"Connect on Startup"* para que a interface tente se conectar automaticamente ao iniciar.

Nesta aba há uma outra área que controla a seleção de mapas. Ela é composta por um botão chamado de "*Loaded Map*" e uma *label* que informa para o usuário, por meio de imagens, qual o mapa está selecionado pela interface. Quando apertado o botão, é aberto um janela de seleção de mapas. A figura 23 ilustra a aba "*Home*" e seus componentes, a direita está o "*Robot Panel*" que é o *dasboard* do Turtlebot 2.

Figura 23 – Aba *Home* da janela principal (a esquerda) e o *dashboard* (a direita)



Fonte: Autor

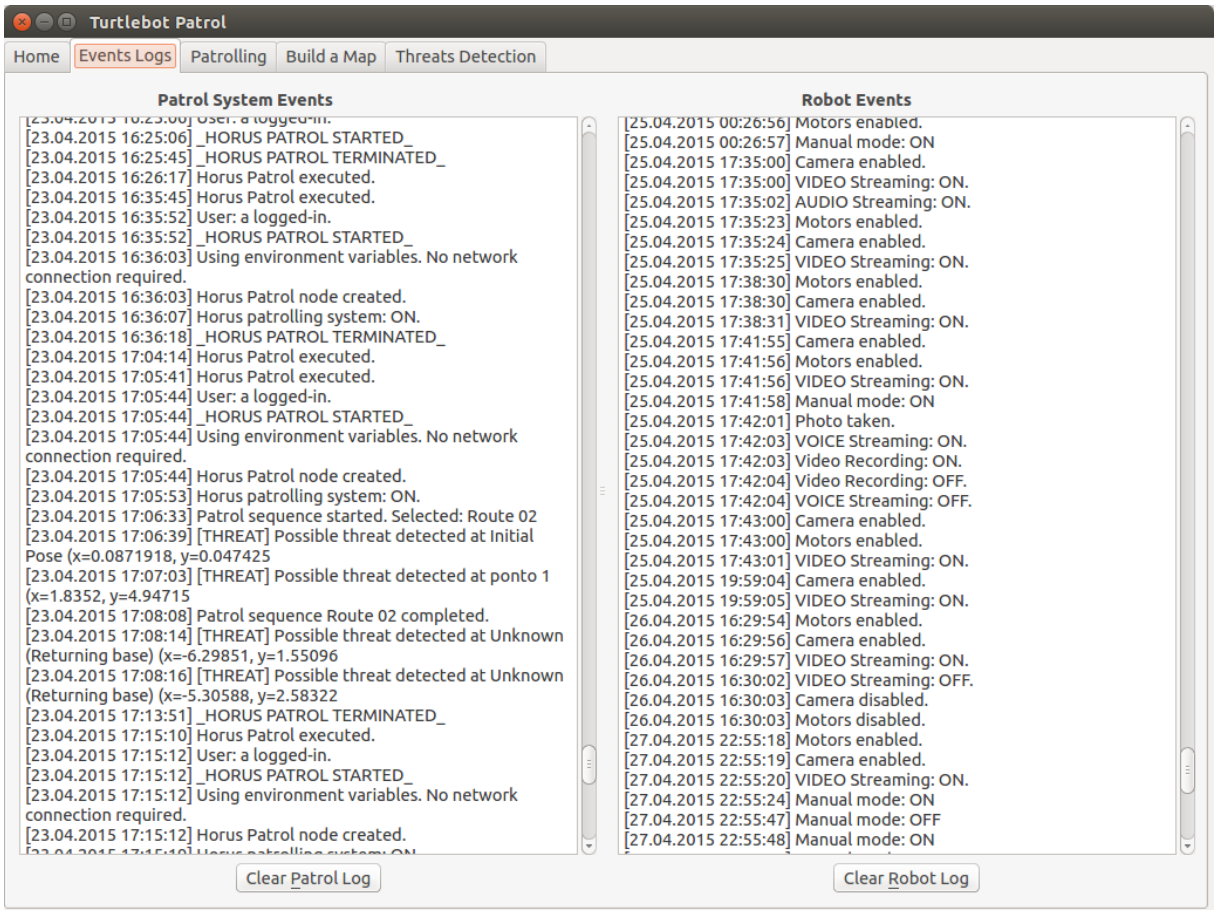
A aba "*Events Logs*", ilustrada na figura 24, informa para o usuário o histórico de acontecimentos considerados importantes que já ocorreram no Horus Patrol. Tais acontecimentos são divididos em dois grupos: os eventos que ocorreram unicamente com o Turtlebot 2 e os que ocorreram com o restante do sistema robótico.

O histórico de eventos são armazenados em arquivos de textos, que são acessados pela interface gráfica. Existem dois botões na parte inferior da aba, e estes tem como função limpar o registro de eventos, onde cada um limpa seu respectivo arquivo. Porém apenas um usuário com privilégios de administrador pode utilizar este recurso.

Cada evento armazenado deve conter três itens: o horário que ocorreu o evento, sua criticidade e sua descrição. Existem três níveis de criticidade de um evento:

- Informação (*INFO*): são eventos comuns no sistema e servem apenas para registro e futuras análises, como quando um usuário inicia uma sequência de patrulha;
- Aviso (*WARNING*): são eventos internos do sistema que necessitam de atenção e ação por parte do usuário, como quando o Turtlebot 2 atinge algo;
- Ameaça (*THREAT*): são eventos que identificam a detecção de uma ameaça externa, como um intruso.

Figura 24 – Aba *Events Logs* da janela principal



Fonte: Autor

A "*Patrolling*", ilustrada na figura 25, é a aba que unifica todos os objetos responsáveis pela execução do processo de patrulhamento do Horus Patrol. Esta aba é composta por quatro componentes: uma caixa de grupo de botões, um painel que exibe informações do processo de patrulhamento, uma *label* que transmite as imagens da câmera do robô em tempo real, e um *widget* que mostra o ambiente simulado criado pelo Rviz mostrando em tempo real a pose do robô em relação ao mapa. A sua caixa de grupo possui seis botões:

1. *Start*: inicia uma sequência de patrulha;
2. *Stop*: interrompe uma sequência de patrulha;

3. *Go to..*: solicita que o robô se desloque até um ponto de patrulha (*Place*);

4. *Route Selection*: abre a janela de seleção de rotas;

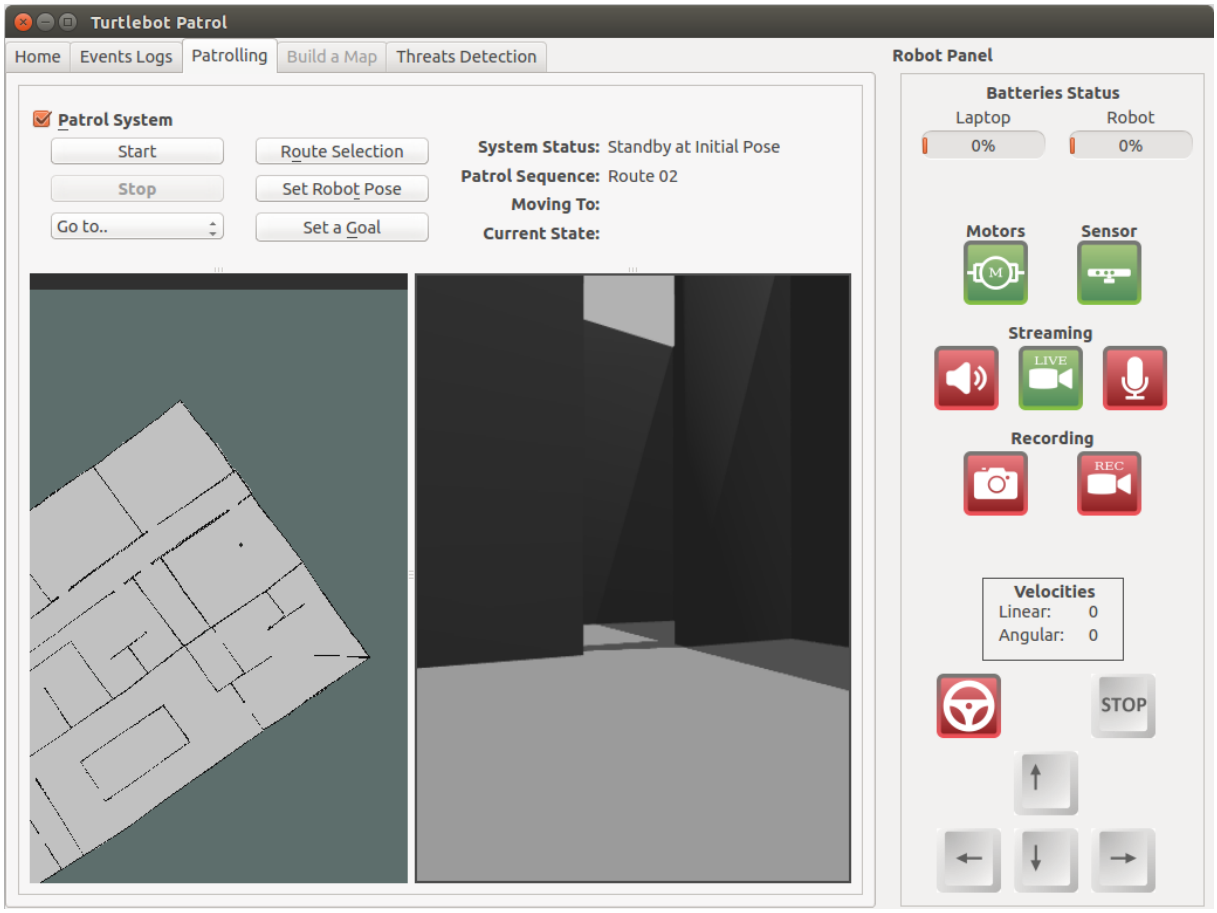
5. *Set Robot Pose*: informa ao robô sua pose no mapa. Ao apertar este botão o usuário deve escolher no mapa do Rviz (localizado no canto inferior esquerdo) a localização da pose;

6. *Set a Goal*: comanda que o robô se desloque até uma posição do mapa. Ao apertar este botão o usuário deve escolher no mapa do Rviz a localização do destino.

A caixa "*Patrol System*" é a responsável por habilitar o modo de patrulha e ela, por sua vez, só é habilitada quando houver um mapa carregado no Horus Patrol. Quando este modo é iniciado, apenas o botão de seleção de rota e o *Set Robot Pose* são habilitado, o funcionamento dos outros botões só é liberado após a escolha de uma rota, com exceção do botão *Set a Goal* que é habilitado após o Turtlebot 2 estiver localizado no mapa.

O painel de informações da patrulha exibe as seguintes informações: o estado do sistema, a rota que está selecionada, para onde o robô está se deslocando e em que ponto da sequência de patrulha ele está.

Figura 25 – Aba *Patrolling* da janela principal



Fonte: Autor

O modo de operação de mapeamento do Horus Patrol está localizado na aba "*Build a Map*". Ela está organizada em uma estrutura semelhante a aba de patrulhamento, pois também é composta por quatro componentes: uma caixa de grupo de botões, um painel que exibe informações do processo de mapeamento, uma *label* que transmite as imagens da câmera do robô em tempo real, e um *widget* que mostra o ambiente simulado criado pelo Rviz mostrando em tempo real a evolução do mapa sendo criar e a pose do robô em relação a este mapa.

A sua caixa de grupo possui quatro botões:

1. *Start*: inicia o processo de mapeamento;
2. *Stop*: cancela o processo de mapeamento. Não é possível pausar a construção de um mapa;
3. *Custom Movement*: abre a janela de movimentos customizado;
4. *Save Map*: Salva o estado atual do mapa com o nome e local escolhido pelo usuário.

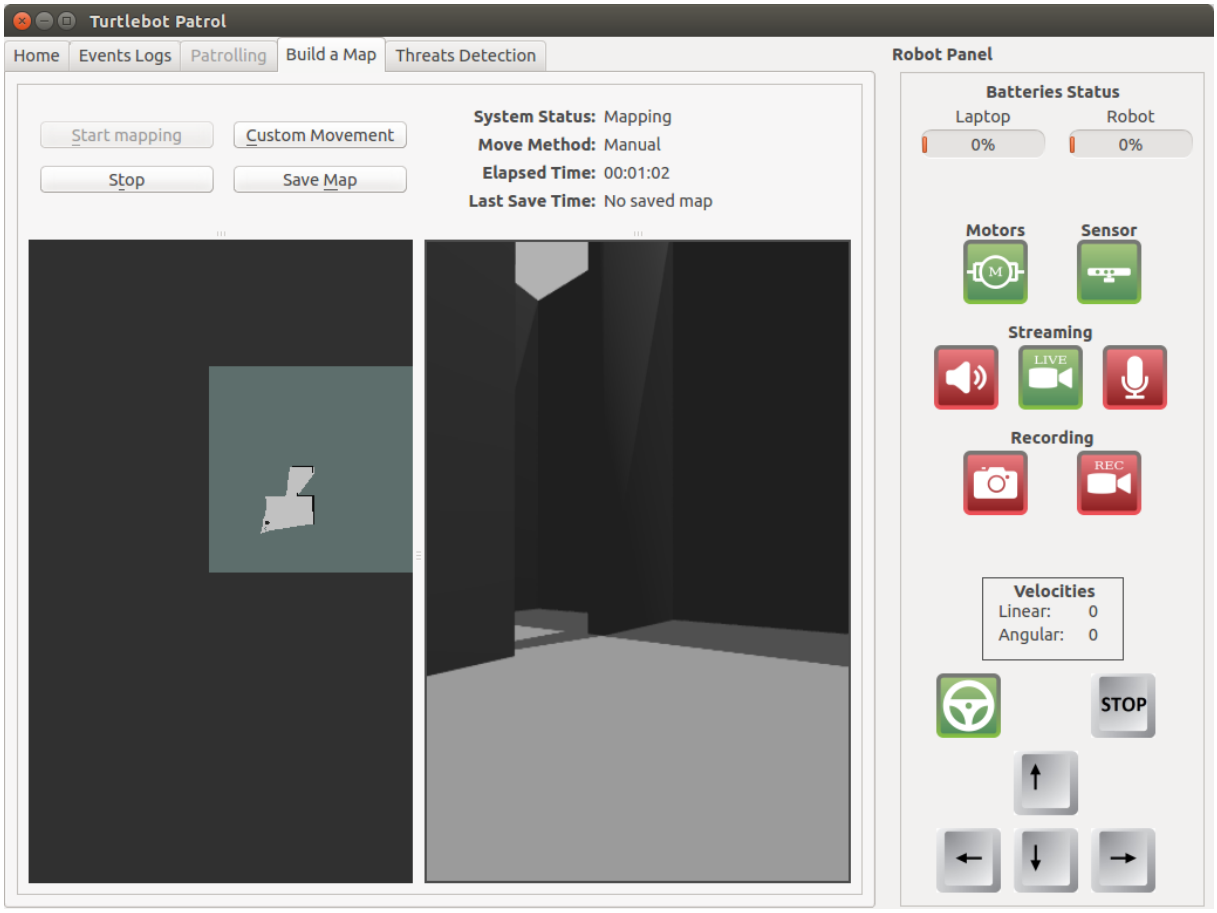
O painel de informações do processo de mapeamento exibe as seguintes informações: o estado do sistema, o método de movimento utilizado (manual ou automático), o tempo de mapeamento decorrido, e o momento da última vez que um mapa foi salvo durante o processo de mapeamento.

É necessário exibir o tempo decorrido do processo de mapeamento mais o tempo da última vez que um mapa foi salvo, para auxiliar o usuário a saber quanto tempo se passou desde a última vez que ele salvou um mapa.

O processo de mapeamento é uma tarefa que demanda tempo de execução, onde este é proporcional ao tamanho da área a ser mapeada. Mapear áreas muito grandes (acima de 1000m^2) requer horas de trabalho, e como não é possível recuperar ou pausar o processo, é útil saber quanto tempo se passou desde a última vez que foi salvo o estado do mapa sendo criado a fim de evitar possíveis problemas.

A figura 26 ilustra a aba de mapeamento em execução no modo manual.

Figura 26 – Aba *Build a Map* da janela principal



Fonte: Autor

A gestão das detecções de possíveis ameaças do Horus Patrol é realizada na aba "*Threats Detection*". Nesta área um usuário pode visualizar todas as ameaças que foram detectadas em cada rota. Para cada detecção, quatro informações são apresentadas pela interface gráfica: O horário local da estação de trabalho no momento da detecção, a área aproximada em que ela ocorreu, o ponto em que o robô estava quando ele detectou a ameaça, e uma captura de imagem da câmera do robô do objeto detectado.

Conforme foi explicado no capítulo anterior, todas estas informações são armazenadas em uma base de dados. A "*Threats Detection*" faz apenas a leitura dos dados armazenados nesta base e os organiza para melhor visualização. A única informação que não é armazenada na base de dados, é a imagem capturada. As imagens são armazenadas em ficheiros do sistema operacional, onde estes também são divididos por rota.

A área aproximada em que ocorreu uma detecção é calculada da seguinte maneira: o deslocamento do robô, conforme explicado no capítulo anterior, é realizado sempre entre o ponto atual do robô para o próximo ponto a ser patrulhado, então para realizar o cálculo é traçada uma linha imaginaria entre os pontos. A área aproximada será o ponto mais próximo da pose do robô no momento da detecção em relação a linha imaginária traçada.

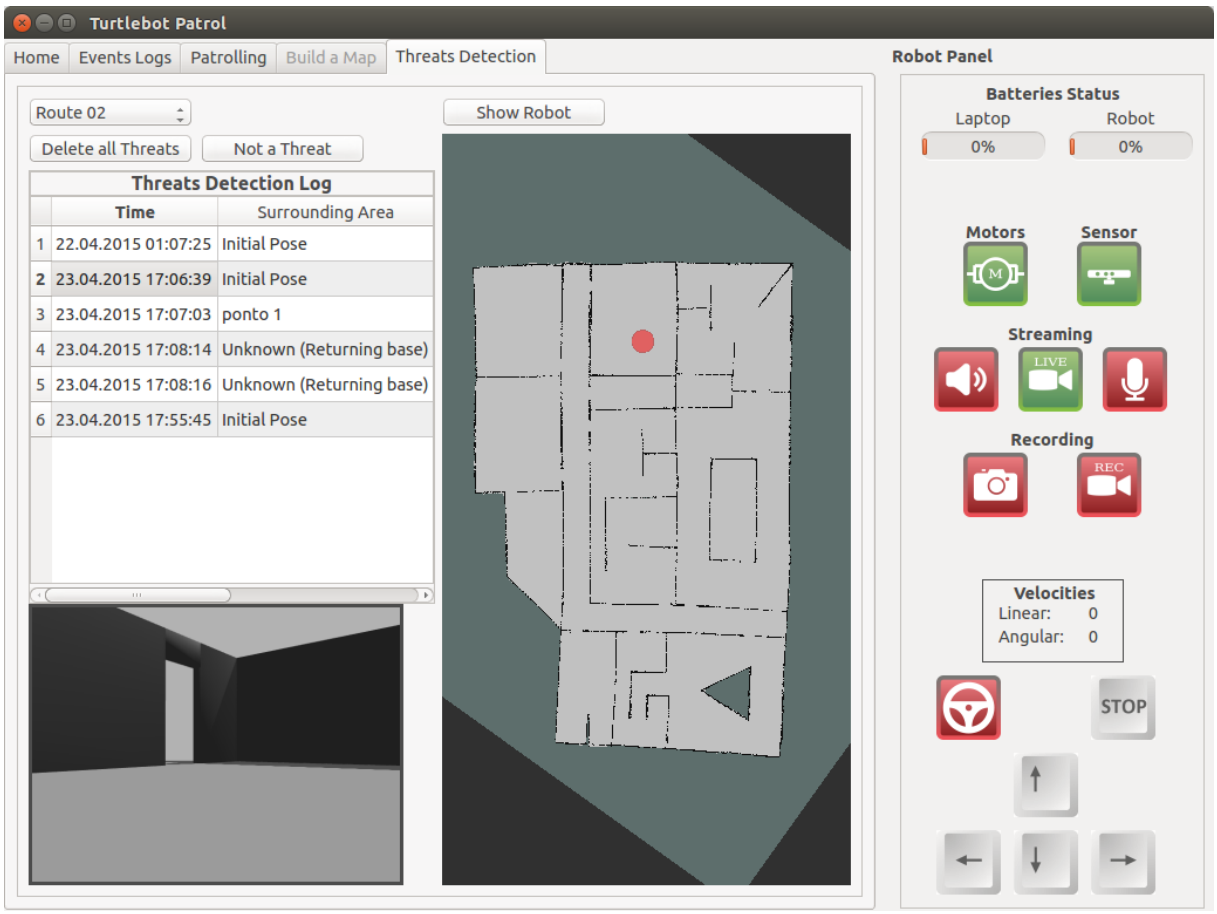
Esta aba é composta por quatro componentes: um grupo de botões, uma tabela

detalhando as detecções do sistema, uma *label* mostrando a imagem capturada pela câmera no momento de uma detecção, e um *widget* que mostra o ambiente simulado criado pelo Rviz mostrando em tempo real a pose do robô em relação ao mapa e a localização de uma ameaça.

Existem quatro botões nesta aba: um para seleção da rota a ser apresentada, um para apagar todas as ameaças detectadas na rota selecionada, um para apagar a detecção que está selecionada, e um botão para mostrar e esconder a pose do robô no mapa.

Quando um usuário seleciona uma ameaça da tabela, a interface ilustra a posição do robô no momento da detecção e mostra a imagem que foi capturada no momento da detecção. Este cenário é exemplificado pela figura 27, onde um usuário selecionou a ameaça de numero dois. O círculo vermelho representa a posição do robô no mapa no momento da detecção e no canto inferior esquerdo é mostrada a imagem capturada pela câmera.

Figura 27 – Aba *Threats Detection* da janela principal



Fonte: Autor

5.1.3 Seleção e gerenciamento de rotas

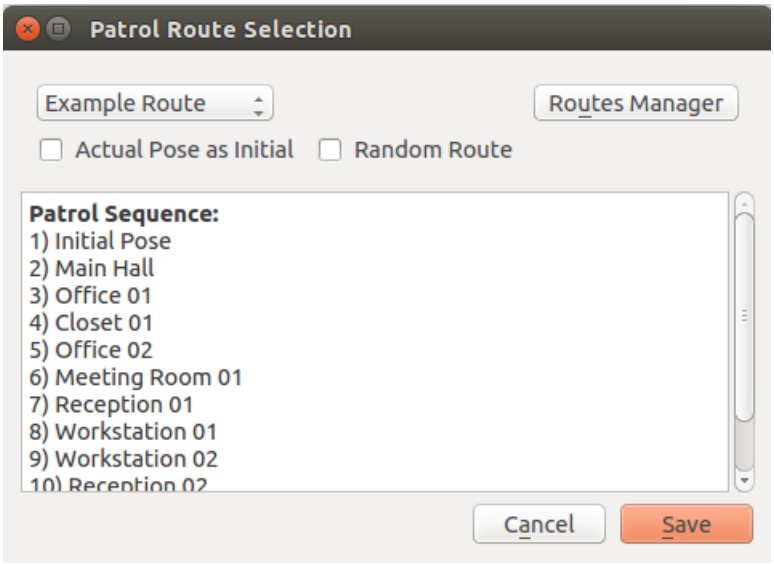
Na aba de patrulhamento, ao apertar o botão de seleção de rota, a caixa de diálogo de seleção de rotas é chamada. Nela um usuário seleciona qual rota o robô deverá patrulhar.

Esta caixa contém um *combobox* que possui a lista de possíveis rotas a serem selecionadas, bem como outros três botões: um para acessar o gerenciador de rotas, um para fechar a caixa de diálogo e ignorar quaisquer mudanças, e um botão para aceitar a seleção feita. Ela também possui uma caixa de texto que exibe os detalhes da rota selecionada e duas outras caixas de seleção: uma para escolher uma rota aleatória, e a outra para substituir a pose inicial da rota pela pose atual do robô.

Ao iniciar está caixa de diálogo, a interface adiciona todas as rotas (que são tabelas da base de dados "*Routes*") ao *combobox* de seleção de rota. Quando um usuário seleciona uma rota, a caixa de texto exibe todos os nomes contidos no atributo "*Place*" desta rota. Se o usuário marcar a caixa "*Actual Pose as Initial*", o Horus Patrol irá substituir a pose inicial da rota pela pose atual do robô, porém esta função só poderá ser utilizada se o robô estiver em uma posição conhecida. Por fim, ao apertar o botão "*Save*", o sistema carrega a rota selecionada e seus atributos em variáveis e retorna para a janela principal.

A figura 28 ilustra uma situação da caixa de diálogo de seleção de rotas, onde foi selecionada um rota.

Figura 28 – Caixa de diálogo para seleção de uma rota



Fonte: Autor

Ao apertar o botão "*Routes Manager*", uma nova caixa de diálogo é chamada. Nesta nova janela um usuário pode gerenciar as rotas cadastradas na base de dados "*Routes*". As funções de gerenciamento de rotas que esta janela contém são: criação, edição, e remoção.

Esta janela é composta por três componentes:

1. Área de abas: localizada no canto inferior direito, é uma área composta de duas abas, uma para criar rotas e outra para apagar ou remover rotas.

2. *Route Summary*: localizada no canto superior direito, ela é uma tabela interativa que apresenta o resumo das informações contidas em uma rota. Quando o usuário

está na aba de edição de rotas, esta tabela apresenta as informações contidas na rota selecionada. Se o usuário estiver na aba de criar uma rota, a tabela mostrará as informações da tabela que está sendo construída.

3. Rviz *widget*: ilustra o ambiente simulado criado pelo Rviz mostrando a pose do robô em relação ao mapa de um "*Place*".

As abas possuem um grupo, chamado de "*Place Information*", que contém cinco editores de linha para que um usuário adicione as informações de um "*Place*" a ser criado ou modificado, como seu índice, ou seja, sua posição na ordem de execução na sequência de patrulha; seu nome; sua posição "x"; sua posição "y"; e sua orientação. Destas, a única informação que não pode se repetir é o índex, pois o sistema patrulha apenas um ponto por vez.

Com exceção do *combobox* presente na aba de edição, que tem a função de selecionar a rota a ser modificada, as duas abas possuem os mesmos botões:

1. *Get Pose*: Captura uma pose do mapa. Ao apertar este botão o usuário deve pressionar no mapa do Rviz o ponto (coordenada "x" e "y") desejado, e arrastar o ponteiro para informar a orientação do robô. As informações capturadas são então inseridas em seus respectivos editores de linha;

2. *Add Place*: Adiciona o "*Place*" a rota. Caso já exista um ponto cadastrado no mesmo índice escolhido, o usuário deverá decidir se irá substituir o "*Place*" ou escolher outro índice para o ponto. Este botão só é habilitado quando o usuário informa todos os dados do ponto;

3. *Clear Place*: Limpa os dados contidos no grupo "*Place Information*";

4. *Delete Route*: Apaga a rota sendo criada ou modificada;

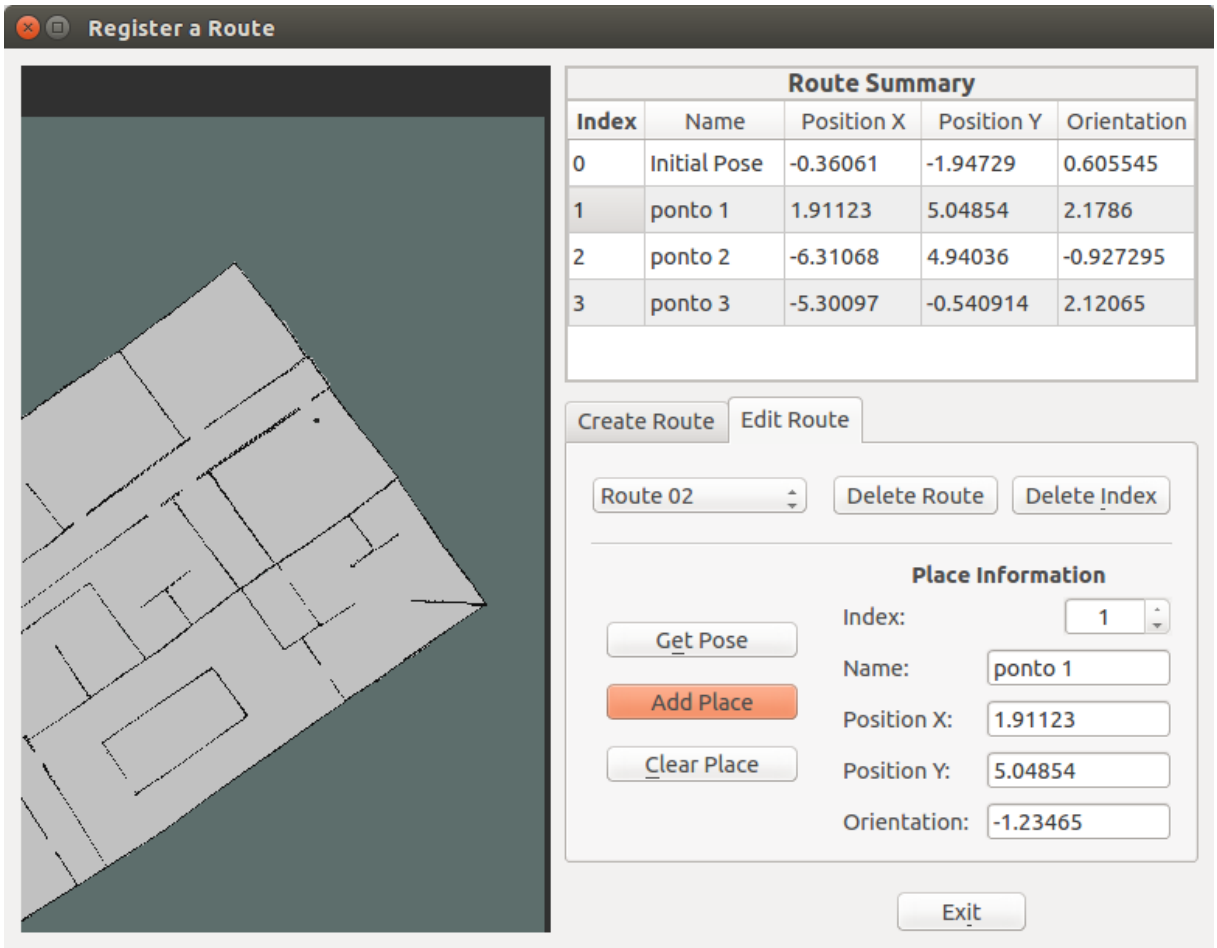
5. *Delete Index*: Apaga apenas um índice selecionado na tabela pelo usuário.

No modo criação de rota, os pontos adicionados pelo usuário são armazenados em uma rota temporária chamada "temp_route". Esta rota é oculta na interface gráfica e não pode ser utilizada como uma sequência de patrulha. Sua função é armazenar os "*Places*" de uma rota que está sendo criada. Quando o usuário aperta o botão de salvar uma rota, a "temp_route" é renomeada para o nome escolhido pelo usuário e uma nova rota temporária é criada. A "temp_route" armazena suas informações mesmo se o usuário sair deste gerenciador. Assim, o usuário pode interromper a criação de uma rota sem perder as informações.

Tanto na aba de edição quanto na criação, quando o usuário seleciona uma linha da tabela a pose do robô é mostrada no ambiente virtual do Rviz. isto é feito pra melhorar a visualização de cada "*Place*" de uma rota.

A figura 29 apresenta um exemplo onde uma rota está sendo editada.

Figura 29 – Janela de gerenciamento de rotas



Fonte: Autor

5.1.4 Caixas de diálogo auxiliares

Nesta interface gráfica existem outras caixas de diálogo com funções auxiliares. Dentre elas, duas são se destacam: a janela para configuração de um movimento automático para ser utilizado no processo de mapeamento e a janela que informa ao usuário que uma ameaça foi detectada.

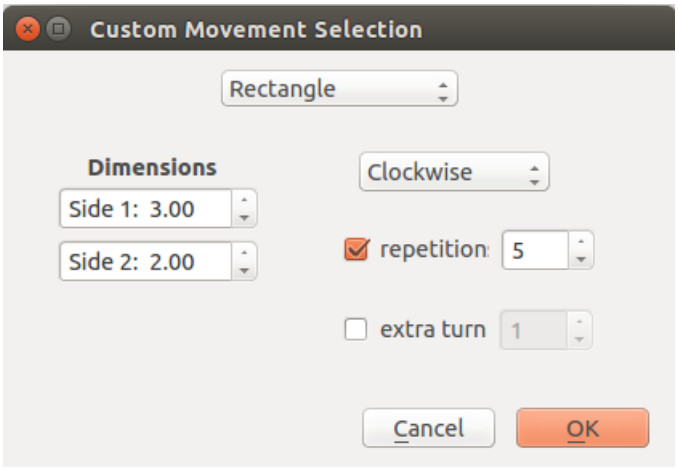
A "*Custom Movement Selection*", ilustrada na figura 30, é o nome dado à esta caixa de diálogo que configura um movimento automático em forma de retângulo para auxiliar no processo de mapeamento. Ela é chamada quando o usuário aperta o botão "*Custom Movement*" na aba de mapeamento da janela principal. Quando iniciada o usuário deve informar se a forma geométrica é um quadrado ou retângulo, após isso ele deve informar as dimensões da forma e o sentido de rotação do robô, ou seja, se é no sentido horário ou não. Por fim, esta função possui duas caixas de seleção opcionais:

1. *repetition*: possibilita que o usuário configure quantas vezes a execução da forma geométrica deve ser repetida. Esta opção, embora opcional, é muito importante

para justificar o uso deste movimento personalizado, pois a ideia é que o usuário programe o robô para executar o movimento selecionado várias vezes, reduzindo assim, o tempo gasto pelo usuário controlando o robô no processo de mapeamento;

2. *extra turn*: quando marcada, adiciona rotações extras em cada vértice da forma geométrica, ou seja, toda vez que o robô chega a um vértice ele rotaciona o número de vezes selecionado e depois continua seu movimento.

Figura 30 – Caixa de diálogo para configuração de um movimento automático

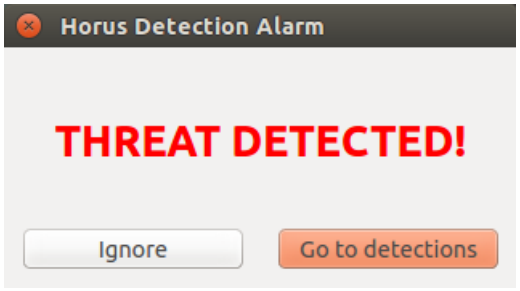


Fonte: Autor

A "*Horus Detection Alarm*", ilustrada na figura 31, é o nome dado à janela que informa ao usuário que uma ameaça foi detectada. Durante o processo de patrulhamento, quando uma ameaça é detectada pelo robô, esta janela é chamada.

Enquanto esta janela estiver aberta, nenhuma outra pode ser aberta ou utilizada, além disso, um sinal sonoro será tocado para chamar atenção do usuário. Nesta janela há dois botões, um que ignora (mas não apaga) a detecção, e outro que direciona o usuário para a aba "*Threats Detection*" da janela principal, a fim de mostrar os detalhes da detecção ao usuário.

Figura 31 – janela de aviso de ameaça



Fonte: Autor

5.2 Nó de controle do sistema

O controle e gestão do robô foi realizado pela classe ROSNode, conforme explanado na seção 4.8.1.1. Primeiramente, foram realizados testes para avaliar se a implementação desta classe foi eficaz em coletar as informações publicadas nos tópicos, dos quais ela é inscrita, e verificar se ela consegue publicar mensagens de comando ao sistema. A ROSNode obteve um comportamento esperado, conseguindo enviar comandos aos nós de controle do Turtlebot 2 e receber as mensagens as publicadas pelos nós de controle dos sensores.

Após garantir que o nó de controle do sistema esteja integrado corretamente ao ROS, foram testados seus algoritmos internos de teleoperação, detecção de ameaça por cor, movimentação automática, cálculo de distâncias acumuladas, e o algoritmo responsável pelo controle da execução de uma sequência de patrulha. Tais testes são explanados com mais detalhes na seção 5.6.

Destes algoritmos, apenas o de movimentação automática apresentou resultados abaixo do esperado pois, após aproximadamente quinze minutos utilizando esta função, o erro acumulado da odometria começa a impactar a precisão do movimento. Este erro gera uma diferença entre a posição inicial do movimento e seu final apenas para grandes repetições, porém esta diferença é baixa (quinze centímetros).

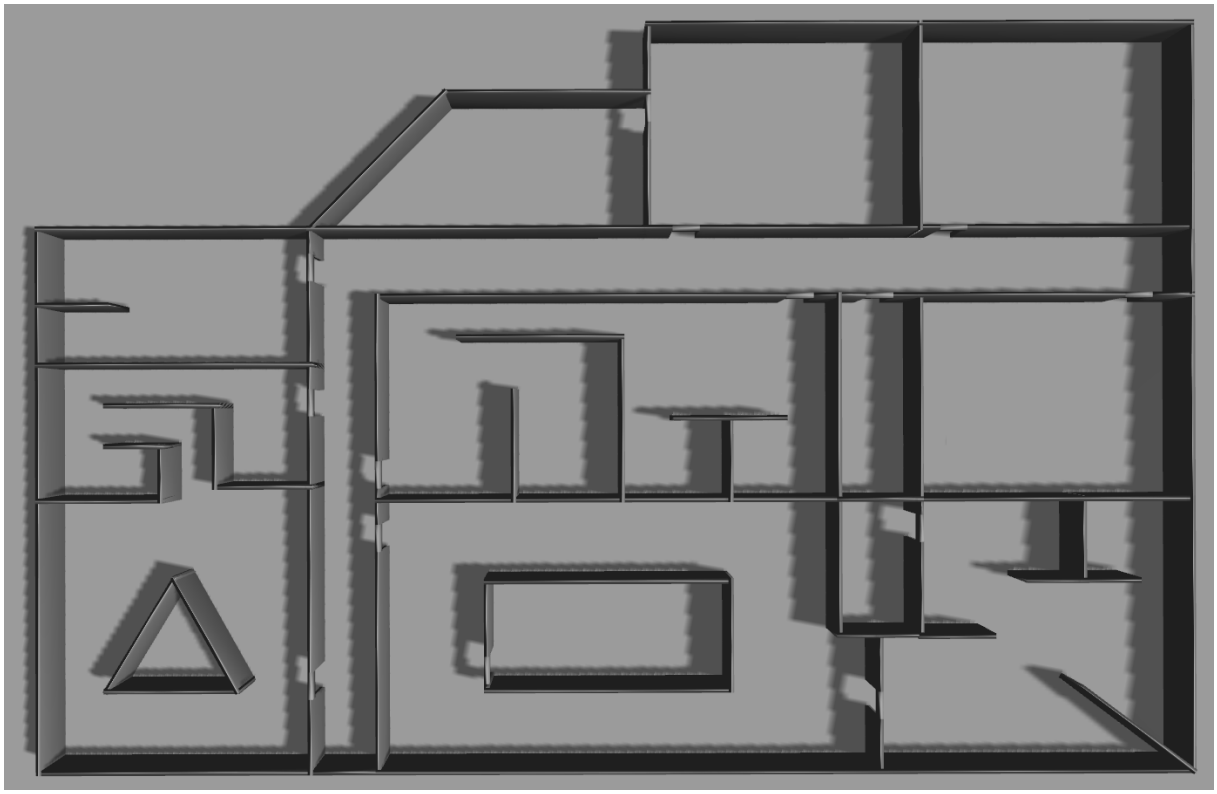
Portanto, os resultados desta classe foram satisfatório, pois todas as suas funções cumpriram seu objetivo. A implementação completa pode ser vista no apêndice C.

5.3 Mapeamento

O processo de mapeamento manual se mostrou uma ferramenta ineficiente apesar de ser eficaz, ou seja, ela atinge seu objetivo de criar um mapa do ambiente, porém o tempo necessário é muito longo. Conforme explanado no capítulo anterior, o Horus Patrol utiliza um algoritmo de SLAM chamado de "gmapping", que consiste em uma técnica que utiliza um filtro de partículas para gerar mapas.

Esta funcionalidade do sistema foi testada em um ambiente virtual, ilustrado na figura 32, extenso com dimensões de aproximadamente 1180m², contendo diversas áreas com características diferentes. Este ambiente foi desenvolvido utilizando o *software* de simulação 3D Gazebo, que é uma ferramenta para criação de ambientes 3D dinâmicos para testes de sistemas robóticos complexos envolvendo um ou mais robôs de forma realista (KOENIG; HOWARD, 2004).

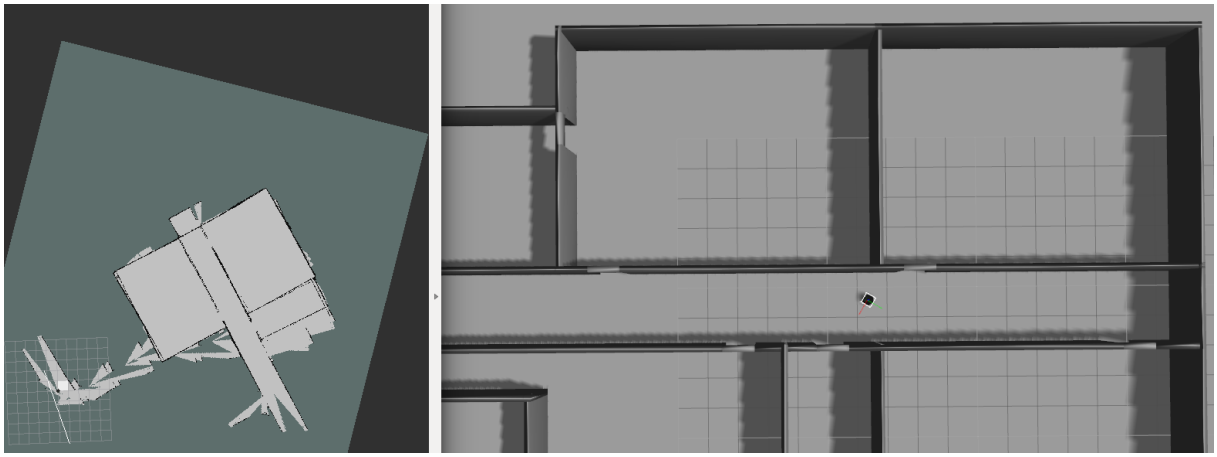
Figura 32 – Ambiente de Testes



Fonte: Autor

Foram executadas diversas tentativas para conseguir gerar um mapa próximo do real, onde em cada tentativa era testado uma abordagem de mapeamento diferente. Em um processo de mapeamento, o problema mais comum é quando o robô se perde no mapa. Neste cenário, o robô continua o mapeamento porém no local errado, gerando assim mapas distorcidos ou até mesmo sobrepostos. A figura 33 ilustra um exemplo que ocorreu durante um dos testes, no qual o robô se perdeu no meio do mapeamento. Só se mostrou possível o robô se relocalilzar se ele voltar, imediatemente após se perder, a uma área já mapeada. Entretanto, as chances são diretamente proporcionais à quantidade do ambiente já mapeado, ou seja, se boa parte do ambiente já está mapeado então o robô tem grandes chances de se relocalizar.

Figura 33 – Robô perdido durante um processo de mapeamento



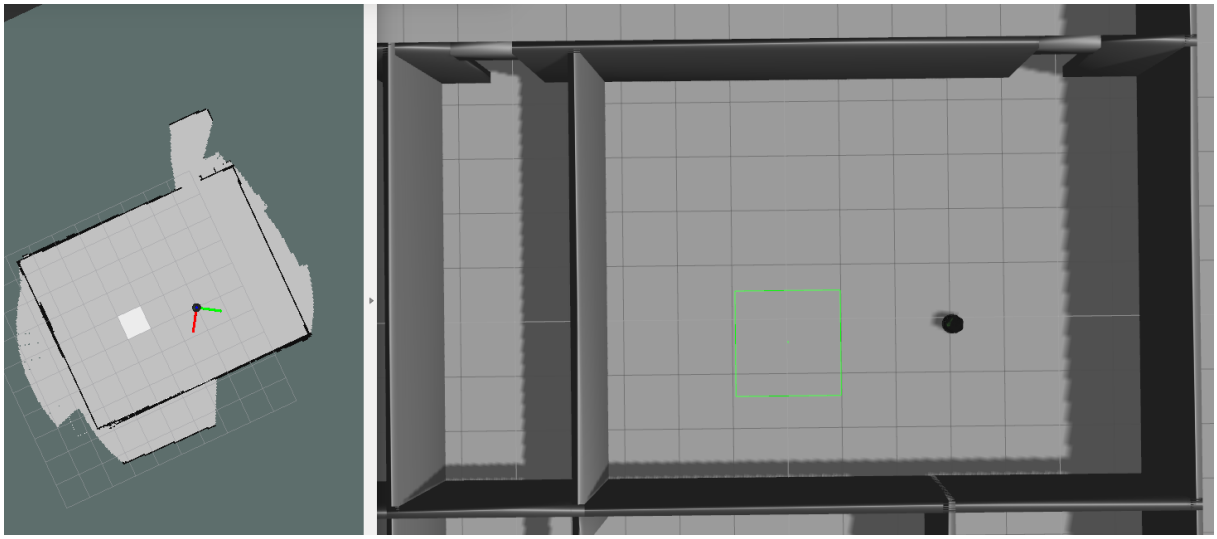
Fonte: Autor

Após os testes, a abordagem de mapeamento mais eficiente encontrada utiliza uma movimentação sempre visando uma *feature* do ambiente (como um canto ou uma porta), e velocidades medianas. O objetivo desta abordagem é movimentar o robô pelo ambiente, mas sempre olhando para uma ou mais *features*. Quanto as velocidades: a odometria não é precisa em velocidades baixas, e o algoritmo não consegue processar todos as medições dos sensores com velocidades altas.

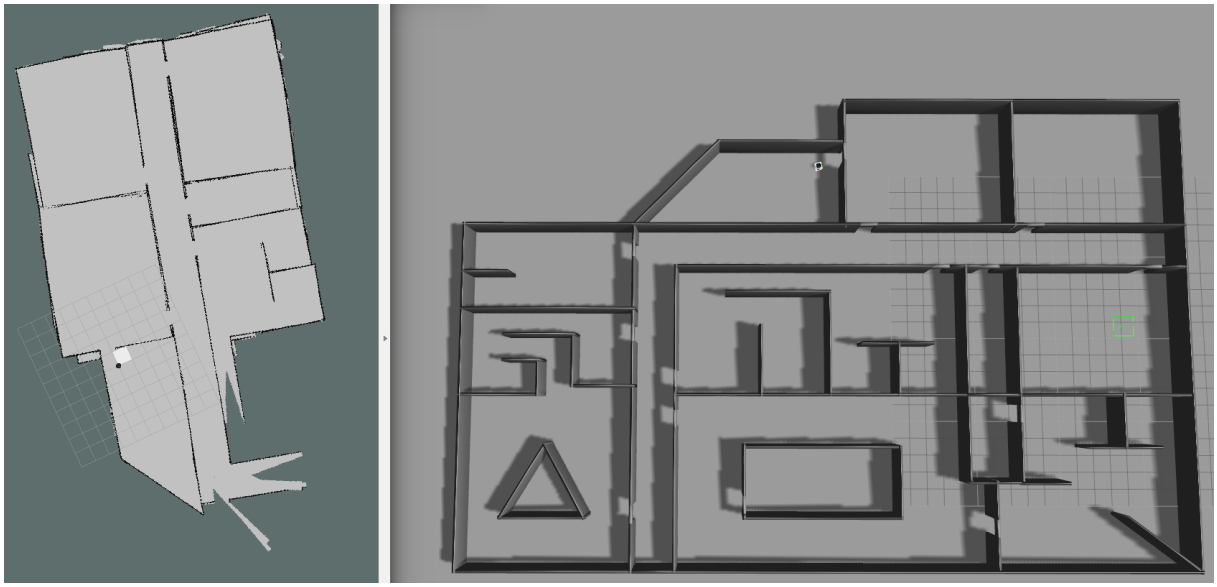
Conforme dito iniciamente, o maior problema deste método de mapeamento utilizando o "gmapping" está no tempo necessário para criar um mapa. Para reduzir este tempo, foi desenvolvido uma função que executa movimentos retangulares pré-programados de forma automática. Esta funcionalidade não tem como objetivo reduzir o tempo total de mapeamento, mas sim o tempo gasto por um usuário controlando o robô manualmente. O melhor tempo gasto para se criar um mapa do ambiente de teste foram de seis horas e meia (utilizando o controle manual auxiliado pela função de movimento automático).

A figura 34 apresenta três imagens capturadas durante o um processo de mapeamento do ambiente de testes.

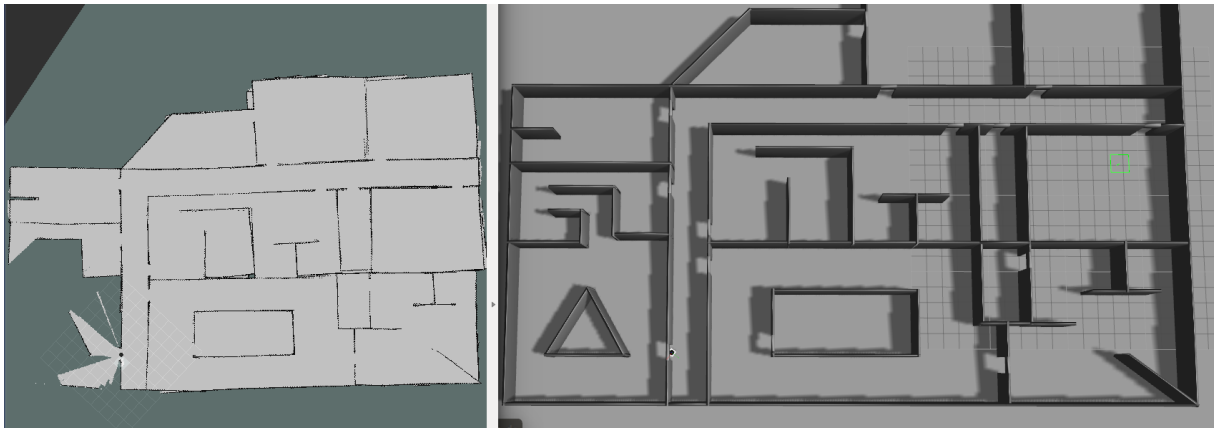
Figura 34 – Evolução do mapa ao longo do processo de mapeamento



(a) Mapa parcial do ambiente após 15 minutos de mapeamento



(b) Mapa parcial do ambiente após 2:30 horas de mapeamento



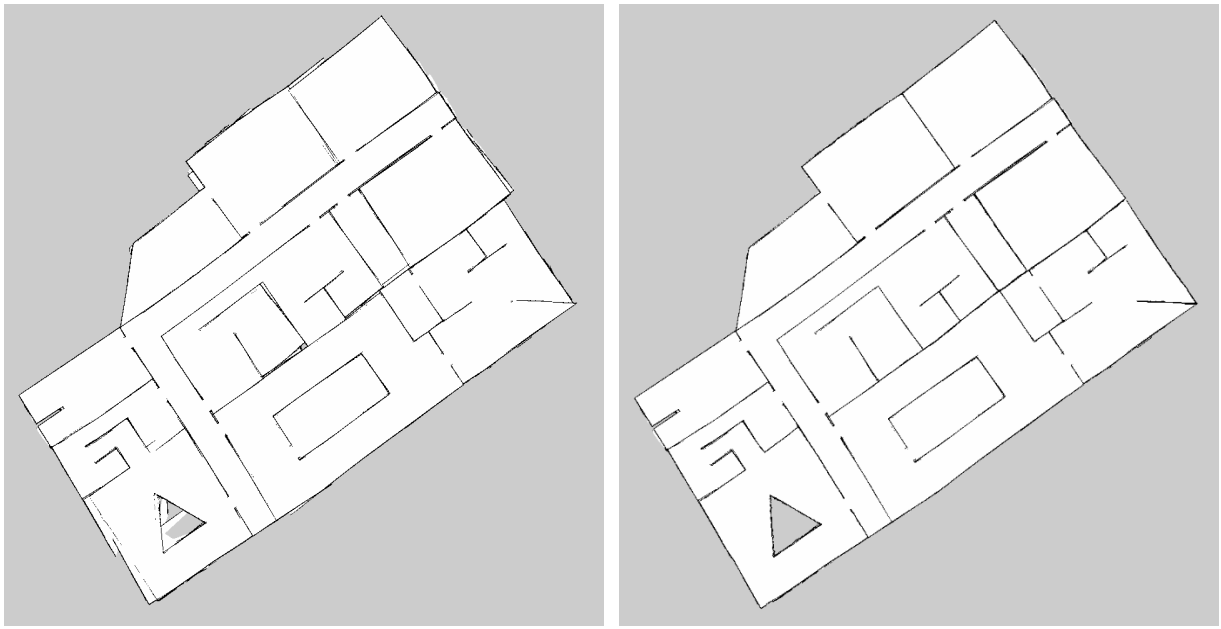
(c) Mapa parcial do ambiente após 5 horas de mapeamento

Fonte: Autor

Não existe um ponto final do mapeamento, isto é definido pelo usuário que manda uma mensagem para o gmapping salvar o mapa criado, bem como seus parâmetros, como origem e resolução (em metros por pixel).

Todos os mapas criados neste trabalho apresentaram falhas, mesmo empregando a abordagem de mapeamento identificada nos testes e o movimento automático. Para melhorar a qualidade do mapa após concluído, utilizou-se um editor de imagem. Por meio dele são removidas rebarbas e são consertadas distorções. Vale ressaltar que um mapa original pode ser utilizado para navegação, pois a grande maioria dos ajustes necessários são apenas estéticos. Até mesmo distorções não impedem o uso do mapa, pois o robô ao navegar se ajusta no mapa. A figura 35 apresenta um mapa antes e depois da edição.

Figura 35 – Comparação entre as versões de mapa concluído



(a) Mapa original (b) Mapa editado

Fonte: Autor

5.4 Localização e navegação autônoma

No Horus Patrol, localizar o Turtlebot 2 pode ser feito de duas maneira: por meio do botão "*Set Robot Pose*" (localizado na aba "*Patrolling*"), ou quando um usuário seleciona uma rota. Conforme já foi explicado anteriormente, cada rota possui uma pose inicial que é a responsável por localizar o robô. Porém espera-se que o usuário posicione o robô aproximadamente na pose correta antes de selecionar a rota pretendida.

O método de localização funcionou da maneira prevista. Foram testados diferentes valores de covariância no momento de informar ao robô sua pose inicial a fim de visualizar o quanto a incerteza (oriunda do posicionamento do usuário) influencia na localização. Após os testes, constatou-se que mesmo sem utilizar um valor de covariância da posição

(ou utilizar um valor pequeno), o robô consegue se manter localizado a longo do tempo. Entretanto, quanto menor a covariância, maior deve ser a precisão da pose informada ao robô. Assim, decidiu-se empregar um valor baixo para esta variável, para garantir que o sistema mantenha localizado o robô mesmo que um usuário informe sua pose (por meio do botão "*Set Robot Pose*") ou o posicione (na pose inicial de uma rota) de maneira imprecisa.

Quanto a navegação, três maneiras de navegar o robô de forma autônoma foram implementados: por meio do botão "*Set a Goa*l", ou por meio do botão "*Go to..*", ou quando o sistema está executando uma sequência de patrulha de uma rota. Foi testado a capacidade do robô se deslocar de um ponto de partida a um de destino seguindo a menor trajetória segura possível, garantindo que o robô não colida com o ambiente. Além disto, foi testado a capacidade de o robô conseguir navegar após uma hora de uso constante. Os resultados foram satisfatórios, pois o sistema conseguiu guiar o robô da maneira almejada. Caso o nó responsável pela navegação não consiga guiar o robô até seu destino (existência de um obstáculo que bloqueie seu caminho), este alerta o sistema e aguarda uma nova operação.

O Apêndice B apresenta um cenário onde há obstáculos no ambiente de teste e o robô deve atravessá-lo. O campo colorido em volta do Turtlebot é o janela dinâmica criada pelo algoritmo de navegação local. Foram reallizadas cinco repetições de deslocamento neste cenário, onde todas as tentativas alcaçaram seu objetivo.

Os testes ajudaram a identificar algumas falhas no sistema de navegação autonoma, como os erros gerados ao utilizar as três métodos de navegação autônoma em conjunto. Portanto, diversas mudanças foram feitas no sistema a fim de permitir que eles trabalhem corretamente em conjunto.

5.5 Detecção de ameaças

Esta função foi testada por meio do algoritmo de detecção da cor vermelha descrito no capítulo anterior. Não foi possível testar o algoritmo de detecção por temperatura, pois não é possível gerar valores de temperatura com o simulador 3D Gazebo. Vale ressaltar que ambos os algoritmos foram implementados na classe ROSNode.

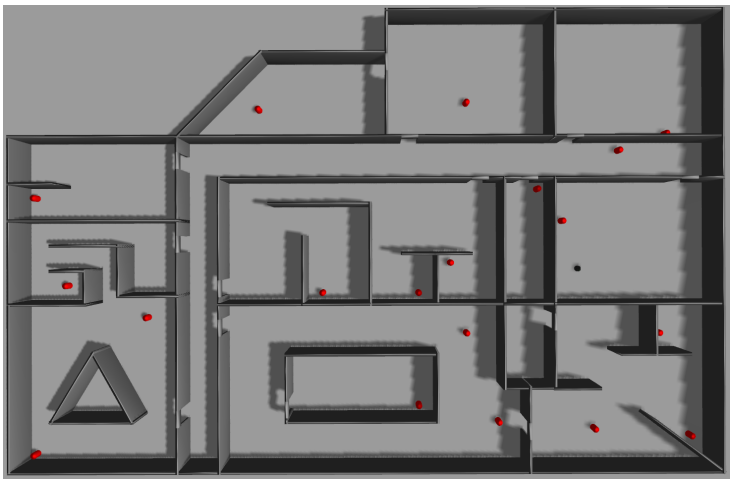
No início da execução dos testes desta função foi detectado uma falha no comportamento do sistema, onde eram gerados diversas detecções por ameaça. Isso se deve ao fato do sistema não ter empregado uma lógica para controlar quando o sistema deve procurar por uma ameaça e, com isso, o sistema detectava continuamente até perder a ameaça de vista. Para solucionar este problema foi adicionada uma lógica que envia apenas um pulso ao detectar uma ameaça.

Após a correção da falha, os testes foram reiniciados e foi possível constatar que esta função estava funcionando corretamente, pois todas as ameaças foram detectadas e suas informações enviadas à base de dados para que seja exibida pela interface gráfica.

5.6 Testes realizados no Horus Patrol

Após todas as funcionalidades (mapeamento, patrulhamento, teleoperação...) ter sido finalizada e testadas separadamente, foram realizados diversos testes descritos por (MYERS; SANDLER; BADGETT, 2011) no sistema Horus Patrol. O seguinte cenário foi adotado para a realização destes testes: um ambiente simulado foi gerado utilizando o simulador 3D Gazebo. Neste ambiente foi adicionado uma área para ser patrulhada e um modelo 3D do Turtlebot que se comunicou com a interface gráfica por meio de uma rede virtual local do ROS. A figura 36 ilustra este cenário descrito.

Figura 36 – Cenário onde foram realizados os testes do Horus Patrol



Fonte: Autor

O primeiro teste realizado no Horus Patrol foi o de sistema (*system testing*), no qual são analisados se os objetivos iniciais descritos na proposta do projeto são cobertos pelo sistema. Foi verificado se todos os objetivos específicos deste trabalho 1.3.1 foram alcançados de maneira satisfatória no sistema.

Em seguida, foi realizado o teste de função (*facility testing*), onde é verificado se o sistema cumpre sua função, ou seja, se todos seus modos de operação, funcionalidades e objetos, se comportam de maneira satisfatória. Os resultados foram satisfatórios, ou seja, o Horus Patrol consegue:

- criar mapas métricos de um ambiente;
- navegar de maneira autônoma;
- detectar ameaças e enviá-la imediatamente à interface gráfica;
- exibir em tempo real a pose do Turtlebot 2 no mapa;
- transmitir as imagens capturadas pela câmera em tempo real;
- permitir que um usuário teleopere o Turtlebot 2;

- gerenciar rotas, usuários e eventos do sistema.

O terceiro teste foi o de desempenho (*performance testing*), que verificou se o tempo de resposta e taxa de transferência das funcionalidades estão em um padrão aceitável. O resultado foi satisfatório para todas as funções. A taxa de transferência total das mensagens que são constantemente transmitidas para a interface podem ser vistas na imagem 37. Nela é possível verificar que durante a execução de uma sequência de patrulha o valor total transmitido pela rede de comunicação é aproximadamente 829,9KB, o que é facilmente suportado por um roteador *wireless*.

Os dados *unknown* indicam que o referido tópico não estava publicando no momento da captura da imagem. No sistema todos estes valores ocorrem em tópicos que publicam poucas mensagens, como informações de pose inicial publicadas no "/initialpose".

Figura 37 – Taxa de transferência de dados por tópicos transmitido à interface gráfica

Topic Monitor

Topic	Type	Bandwidth	Hz
/amcl_pose	geometry_msgs/PoseWithCovarianceStamped	776.93B/s	2.13
/camera/rgb/image_raw/compressed	sensor_msgs/CompressedImage	183.46KB/s	20.02
/cmd_vel_mux/input/navi	geometry_msgs/Twist	962.14B/s	20.00
/initialpose	geometry_msgs/PoseWithCovarianceStamped	unknown	unknown
/move_base/result	move_base_msgs/MoveBaseActionResult	2.11B/s	0.03
/odom	nav_msgs/Odometry	623.53KB/s	1000.00
/scan	sensor_msgs/LaserScan	21.17KB/s	8.02
/mobile_base/events/bumper	kobuki_msgs/BumperEvent	unknown	unknown
/goal	geometry_msgs/PoseStamped	unknown	unknown

Fonte: Autor

O quarto teste foi o de usabilidade (*usability testing*), empregado para validar se a interface gráfica é aceita pelo usuário. A realização deste teste contou com a ajuda de nove voluntários, onde cada um utilizou todas as funcionalidades do Horus Patrol (por cerca de vinte minutos) e apresentou suas opiniões sobre as possíveis melhorias e os pontos positivos do sistema. Este teste foi o que apresentou o maior número de alterações no sistema. A mais significativa destas alterações foi a mudança do método de gerenciamento das bases de dados, que passaram a utilizar a linguagem SQL ao invés de arquivos de texto.

O último teste realizado foi o operacional a fim de avaliar a estabilidade do programa após uma hora realizando patrulhas. O resultado deste teste também foi satisfatório, onde seu único ponto inesperado foi o acréscimo de memória RAM consumida pelo aplicativo ao longo do tempo. O consumo de memória ao iniciar o aplicativo foi de 37MB, após iniciar o nó "horus_patrol" ele sobe para 49MB e com o modo de patrulhamento ou mapeamento ligado, a memória sobe até se estabilizar com 200MB (após quinze minutos). Foi feita uma pesquisa na comunidade do ROS a fim de identificar a causa raiz. Esta situação, na verdade, não é uma falha ou problema, é algo normal que acontece ao utilizar o visualizador Rviz. Este aumento de consumo da memória surge pois o Rviz armazena dez minutos de mensagens das tranformadas do tópico "tf".

6 CONSIDERAÇÕES FINAIS

Este trabalho apresentou uma proposta de um protótipo de um sistema de vigilância por meio de patrulhas empregando a robótica móvel, com o objetivo de substituir o vigilante responsável pela execução das patrulhas, por um Turtlebot 2. Este robô realiza patrulhas pré-definidas no ambiente, buscando a presença de um intruso, reportando suas detecções em tempo real à uma interface gráfica, que contém ferramentas necessárias para gerir o sistema e interagir com o robô.

Foram desenvolvidas as funções do sistema e para cada uma foram realizados testes para avaliar sua eficácia.

A função de mapeamento utilizada gerou mapas corretamente porém se mostrou um processo demorado. Outro ponto negativo é que não se pode deslocar o robô aleatoriamente no mapa, é necessário empregar uma metodologia de movimentação.

A função de patrulhamento utilizou quatro algoritmos, o AMCL, o A*, o DWA e o algoritmo de gestão a fim de garantir que uma rota seja executada da maneira correta. Os testes mostraram que o desempenho desta função atingiu o seu objetivo, conseguindo seguir os pontos de patrulha contidos em uma rota, navegando em um ambiente mesmo contendo diversos objetos não mapeados e sempre encontrando a menor trajetória a ser seguida na transição entre pontos.

A detecção de ameaças foi desenvolvida por meio de dois algoritmos. Um detecta uma anomalia se na matriz enviada pelo sensor de temperatura existir algum valor superior ao preestabelecido. O segundo detecta uma ameaça por meio de cor. Neste trabalho, somente o algoritmo que verifica cores foi avaliado, pois o simulador empregado nos testes não simula valores de temperatura. Entretanto o método de detecção por cores foi o suficiente para avaliar o comportamento do sistema mediante a uma detecção de ameaça. Portanto, esta função alcançou o objetivo específico estabelecido no trabalho, identificando todas as ameaças colocadas em diferentes distâncias.

Após a análise dos resultados das funções do Horus Patrol nos testes, conclui-se que este sistema está apto a ser empregado em um ambiente virtual. Além disso, por sua interface ter obtido resultados satisfatórios em sua interação com o ROS, pode-se afirmar que ela está pronta para ser empregada em um sistema real, visto que a função de coletar e enviar informações por meio de tópicos no ROS independe do tipo de rede utilizada, virtual ou não. O único requerimento para seu funcionamento é que ambos, a interface e o robô, estejam conectados na mesma rede de comunicação.

Por fim, vale ressaltar que o ROS possibilita que o Turtlebot 2 seja substituído por outro robô móvel. Faz-se necessário apenas ajustar os tópicos a serem utilizados pelo sistema.

A tabela 1 apresenta os custos necessários para implementar o sistema em um ambiente

real. O custo total é muito inferior ao dos trabalhos apresentados no capítulo 2 deste trabalho, o que valida a hipótese de que é possível criar um sistema de vigilância de baixo custo.

Tabela 1 – Custos de implementação do Horus Patrol

Item	Preço (USD)
Turtlebot 2 Essentials	$999,00
Microsoft Kinect (v1)	$130,00
Rapsberry PI 2	$35,00
D6T-44L-06 Thermopile Sensor	$45,00
Roteador 300Mbps	$44,99
Computador para a Interface Gráfica	$500,00
TOTAL:	$1753,99

Fonte: Autor

6.1 Principais Dificuldades Encontradas

A maior dificuldade foi identificar quais objetos do pacote Rviz que precisariam ser utilizados pelo Horus Patrol para que fosse possível gerar a visualização do mapa em tempo real na interface. Para isso, foi necessário procurar, em cada arquivo de implementação de classe deste pacote, os métodos que interagem com a sua interface gráfica padrão. Não existe documentação ou tutorias sobre isto na comunidade do ROS, pois não é comum um pacote utilizar fragmentos do Rviz para gerar sua visualização personalizada.

Outra dificuldade foi a aquisição do Turtlebot 2, pois o mesmo não possui um distribuidor no território brasileiro.

6.2 Trabalhos Futuros

O sistema proposto neste trabalho possui as funcionalidades necessárias, porém ainda é preciso testar na vida real o algoritmo de detecção de ameaças por meio de temperatura. Além deste teste, é necessário implementar algumas melhorias no Horus Patrol para que ele possa ser aplicado em um ambiente real. Abaixo são listadas algumas destas melhorias:

- Implementar a interface gráfica em um dispositivo móvel com o sistema operacional Android, pois o ROS é compatível com o mesmo. O objetivo desta implementação é permitir que um vigilante tenha acesso à interface gráfica, mesmo fora da sala de comando. Portanto, caso o vigilante necessite ir até o local onde uma ameaça foi detectada, ele poderá visualizar e controlar o robô;

- Adicionar a função de transmissão de audio capturado pelo robô e o audio enviado por um vigilante para os auto-falantes do robô a fim de melhorar a interação do vigilante na sala de controle com o robô;

- Substituir o Microsoft Kinect® do Turtlebot 2 pela segunda versão deste dispositivo desenvolvido para o Microsoft Xbox One. O objetivo desta alteração é permitir que o Turtlebot 2 seja empregado em ambientes externos. Isso se torna possível pois esta nova versão do kinect é menos sensível a interferência da iluminação do ambiente e consegue medir profundidade em ambientes bem iluminados como áreas externas.

- Adicionar um sensor *Laser Imaging, Detection and Ranging* (LIDAR) com um campo de visão de 360° a fim de melhorar o processo de mapeamento, pois este tipo de sensor consegue capturar mais *features* de uma vez além de possuírem um alcance maior.

- Substituir o Turtlebot 2 por um por um quadricóptero ou outro tipo de veículo aéreo não tripulado (VANT). O uso de um robô autônomo aéreo como um drone permite que o Horus Patrol seja empregado em qualquer terreno ou formatos de ambiente. Este será o última modificação do sistema planejada.

REFERÊNCIAS

ALECRIM, E. *O que é Wi-Fi (IEEE 802.11)*. Disponível, 2008. Disponível em: <http://www.infowester.com/wifi.php>. 33

AUTOMATION. *History of AGVs*. s.d. Disponível em: <http://www.egeminusa.com/pages/agv_education/education_agv_history.html>. 22

BEZERRA, C. G. *Localização de um robô móvel usando odometria e marcos naturais*. Tese (Doutorado) — Universidade Federal do Rio Grande do Norte, 2004. 22

BHOYAR, R.; GHONGE, M.; GUPTA, S. Comparative study on ieee standard of wireless lan/wi-fi 802.11 a/b/g/n. *International Journal of Advanced Research in Electronics and Communication Engineering (IJARECE)*, v. 2, n. 7, p. 687–691, 2013. 33

CHATILA, R. Mobile robot navigation algorithms. In: AK PETERS, LTD. *Proceedings of the workshop on Algorithmic foundations of robotics*. [S.l.], 1995. p. 97–108. 26

CHUNG, L.-Y. Remote teleoperated and autonomous mobile security robot development in ship environment. *Mathematical Problems in Engineering*, Hindawi Publishing Corporation, v. 2013, 2013. 18, 19

CONLEY, K. Ros/introduction - ros wiki. *ROS Wiki*, 2011. 29

CROW, B. P. et al. Ieee 802.11 wireless local area networks. *Communications Magazine, IEEE*, IEEE, v. 35, n. 9, p. 116–126, 1997. 33

DOUGLAS, E. C. *Internetworking with TCP/IP-Volume 1: Principles, Protocols, and Architectures*. [S.l.]: Prentice Hall, 2000. 34

EVERETT, H. *Sensors for mobile robots: theory and application*. [S.l.]: AK Peters Wellesley, MA, 1995. v. 21. 27

FOOTE, T. *ROS/Concepts - ROS Wiki*. 2013. Disponível em: <http://wiki.ros.org/action/recall/ROS/Concepts?action=recall&rev=63>. 29, 31

FOX, D.; BURGARD, W.; THRUN, S. The dynamic window approach to collision avoidance. *IEEE Robotics & Automation Magazine*, v. 4, n. 1, p. 23–33, 1997. 26

GARAGE, W. *TurtleBot*. 2014. Disponível em: <http://www.turtlebot.com/>. 36

GRISETTI, G.; STACHNISS, C.; BURGARD, W. Improved techniques for grid mapping with rao-blackwellized particle filters. *Robotics, IEEE Transactions on*, IEEE, v. 23, n. 1, p. 34–46, 2007. 24

HARMON, L. D. Automated tactile sensing. *The International Journal of Robotics Research*, SAGE Publications, v. 1, n. 2, p. 3–32, 1982. 27

INTERNATIONAL, G. *TurtleBot 2*. 2014. Disponível em: <http://www.gaitech.hk/wp-content/uploads/2014/09/turtlebot.pdf>. 37

JÚNIOR, H. C. de S. *MODELAGEM, SIMULAÇÃO E CONTROLE DE UM GIROSCÓPIO*. Tese (Doutorado) — Universidade Federal do Rio de Janeiro, 2014. 22

KNIGHTSCOPE. *K5 Robot*. 2015. Disponível em: <http://knightscope.com/technology.html>. 16, 17

KOENIG, N.; HOWARD, A. Design and use paradigms for gazebo, an open-source multi-robot simulator. In: IEEE. *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*. [S.l.], 2004. v. 3, p. 2149–2154. 71

KUKI, M. et al. Human movement trajectory recording for home alone by thermopile array sensor. In: IEEE. *Systems, Man, and Cybernetics (SMC), 2012 IEEE International Conference on*. [S.l.], 2012. p. 2042–2047. 28

MARCHI, J. et al. Navegação de robôs móveis autônomos: estudo implementação de abordagens. Florianópolis, SC, 2001. 21

MAZZAROPPI, M. Sensores de movimento e presença. *Monografia para obtenção de grau em Engenharia Életrica. Universidade Federal do Rio, Escola Politécnica, Rio de Janeiro*, 2007. 28

MYERS, G. J.; SANDLER, C.; BADGETT, T. *The art of software testing*. [S.l.]: John Wiley & Sons, 2011. 77

NANZER, J. A.; ROGERS, R. L. Human presence detection using millimeter-wave radiometry. *Microwave Theory and Techniques, IEEE Transactions on*, IEEE, v. 55, n. 12, p. 2727–2733, 2007. 27, 28

NASA. *NASAfacts: Mars Science Laboratory/Curiosity*. 2013. Disponível em: <http://mars.jpl.nasa.gov/msl/news/pdfs/MSL_Fact_Sheet.pdf>. 21, 22

NEHMZOW, U. *Mobile Robotics, A Practical Introduction*. [S.l.]: Springer Heidelberg, 2003. 20

OLIVEIRA, D. V. G. d.; PETREK, F. J. Sistema de automação residencial controlado via web. Curitiba, 2014. 28

OSÓRIO, F. et al. Pesquisa e desenvolvimento de robôs táticos para ambientes internos. In: *Internal Workshop of INCT-SEC, n. i, São Carlos, SP*. [S.l.: s.n.], 2011. 17, 18

PERAHIA, E. Ieee 802.11 n development: History, process, and technology. *Communications Magazine, IEEE*, IEEE, v. 46, n. 7, p. 48–55, 2008. 34

PIERI, E. R. d. Curso de robótica móvel. *UFSC - Universidade Federal de Santa Catarina*, 2002. 21

PRATT, P.; LAST, M. *A guide to SQL*. [S.l.]: Cengage Learning, 2008. 32

QUIGLEY, M. et al. Ros: an open-source robot operating system. In: *ICRA workshop on open source software*. [S.l.: s.n.], 2009. v. 3, n. 3.2, p. 5. 28

RIBEIRO, M. I. Obstacle avoidance. *Instituto de Sistemas e Robótica, Instituto Superio Técnico*, Citeseer, p. 1, 2005. 25, 26

ROBOT, Y. *Kobuki*. s.d. Disponível em: <http://yujinrobot.com/eng/?portfolio=kobuki>. 37

ROSSUM, G. V. *Comparing python to other languages*. 2002. Disponível em: <https://www.python.org/doc/essays/comparisons/>. 32

ROSSUM, G. V.; JR, F. L. D. *Python reference manual*. [S.l.]: Centrum voor Wiskunde en Informatica Amsterdam, 1995. 32

SABBATINI, R. Imitação da vida: A história dos primeiros robós. *Revista Cérebro & Mente*, n. 9, 1999. Disponível em: <http://www.cerebromente.org.br/n09/historia/turtles.htm>. 21, 22

SAVANTY. *Frenquently Asked Questions - FAQs*. 2014. Disponível em: <http://www.agvsystems.com/faqs/>. 21

SECCHI, H. A. *Una Introducción a los Robots Móviles*. Instituto de Automática - INAUT, 2008. Disponível em: <http://www.obr.org.br/wp-content/uploads/2013/04/Uma_Introducao_aos_Robos_Moveis.pdf>. 20

STROUSTRUP, B. An overview of the c++ programming language. *The Handbook of Object Technology*, Boca Raton: CRC Press LLC, 1999. 32

THRUN, S.; BURGARD, W.; FOX, D. *Probabilistic robotics*. [S.l.]: MIT press, 2005. 24

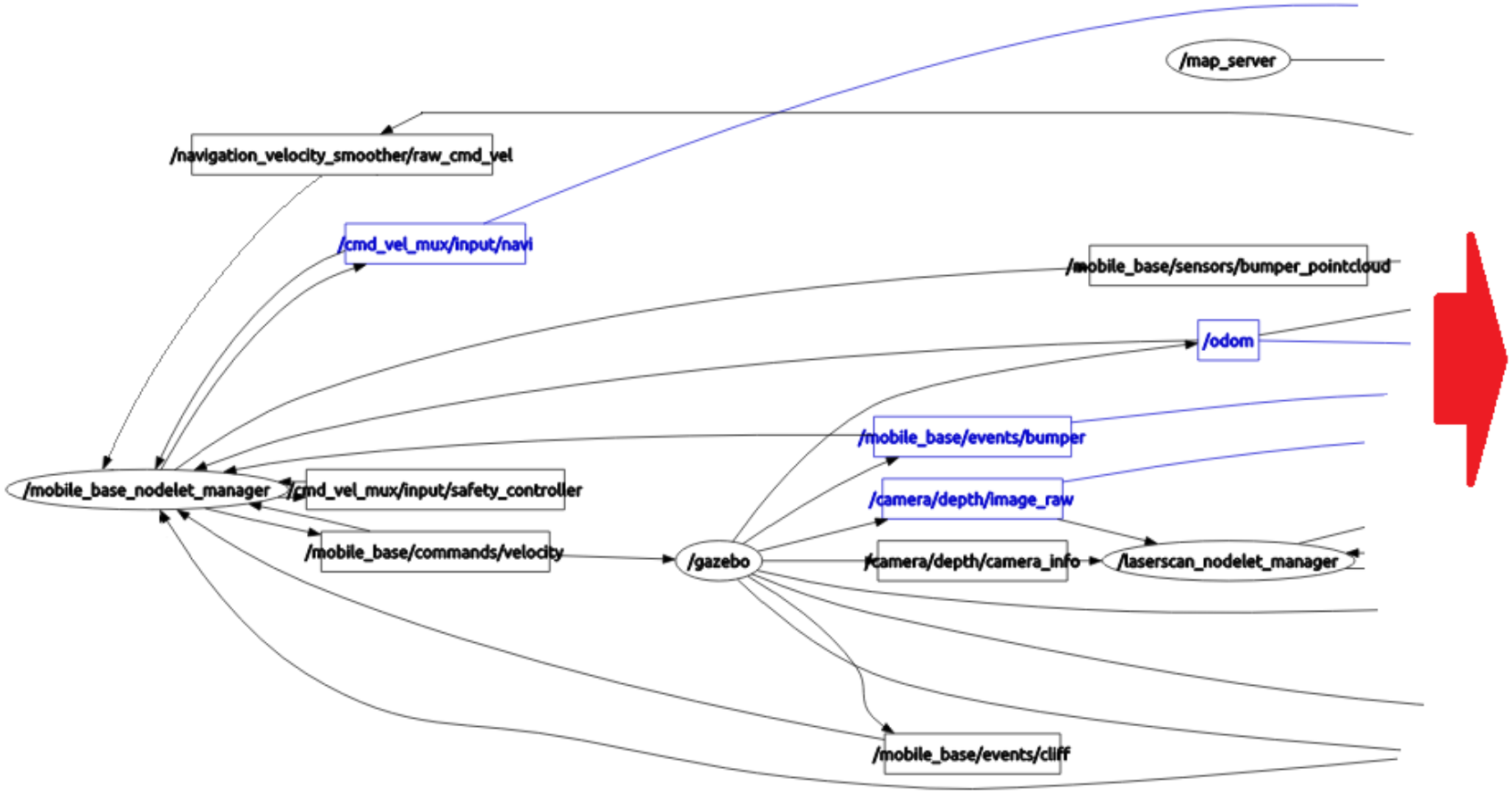
TÖLGYESSY, M.; HUBINSKÝ, P. The kinect sensor in robotics education. In: *Proceedings of 2nd International Conference on Robotics in Education*. [S.l.: s.n.], 2011. p. 143–146. 37, 38

WOLF, D. F. et al. Robótica móvel inteligente: Da simulação às aplicações no mundo real. In: *Mini-Curso: Jornada de Atualização em Informática (JAI), Congresso da SBC*. [S.l.: s.n.], 2009. 23, 24, 25

YIRKA, B. *Willow Garage introduces affordable, programmable robot - TurtleBot*. 2011. Disponível em: <http://phys.org/news/2011-04-willow-garage-programmable-robot-.html>. 36

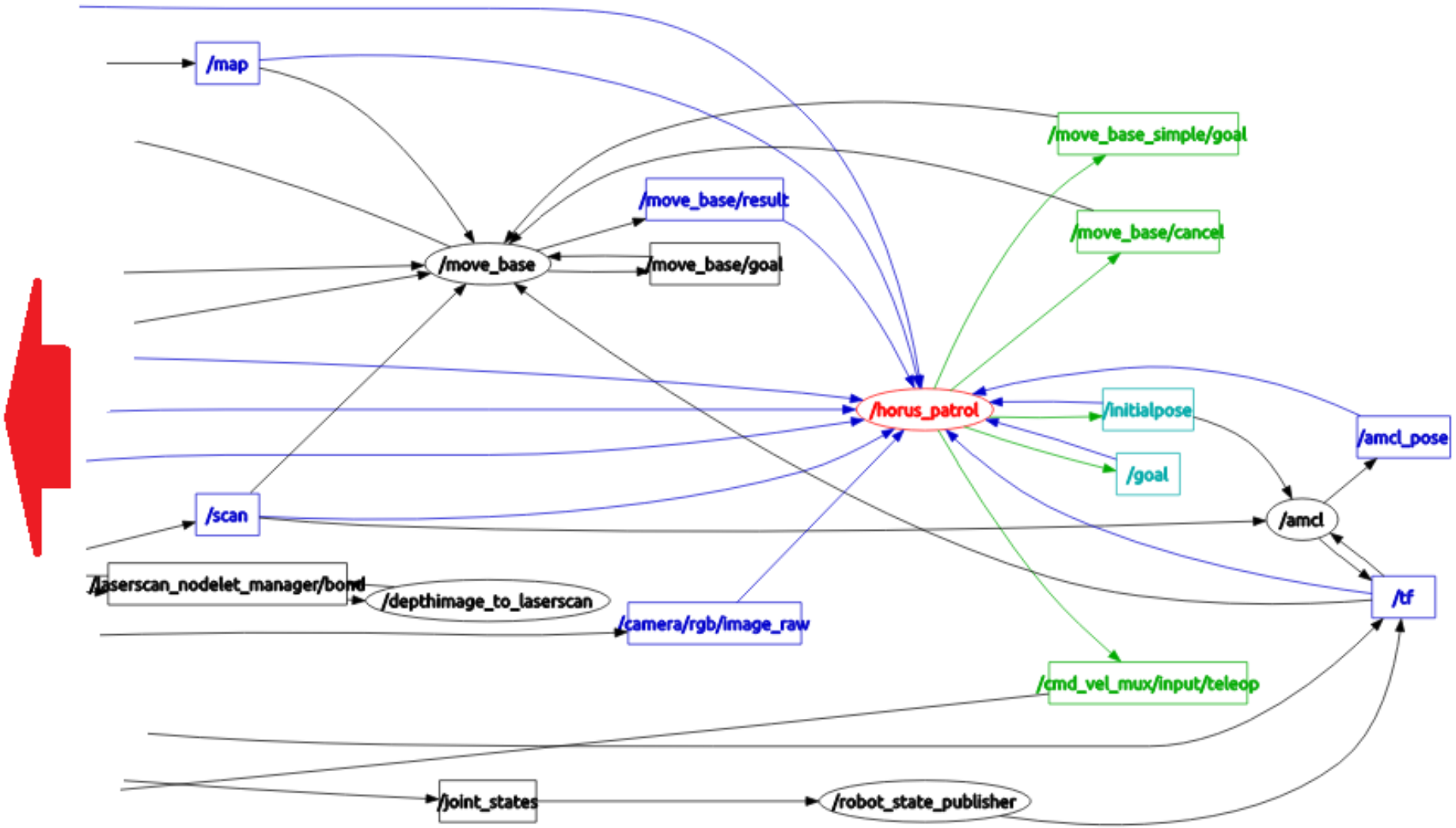
APÊNDICE A – ROS GRAPH DO SISTEMA SIMULADO

Figura 38 – Parte A: Representação da interação entre os nós e tópicos do sistema



Fonte: Autor

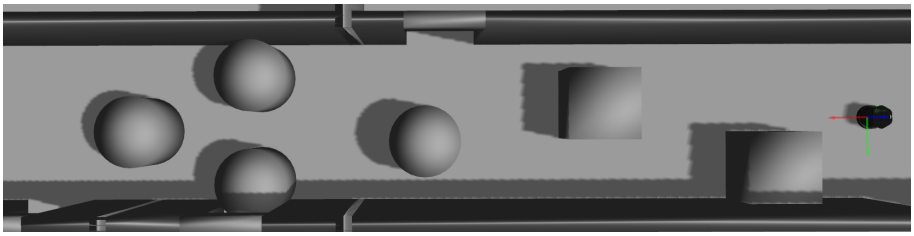
Figura 39 – Parte B: Representação da interação entre os nós e tópicos do sistema



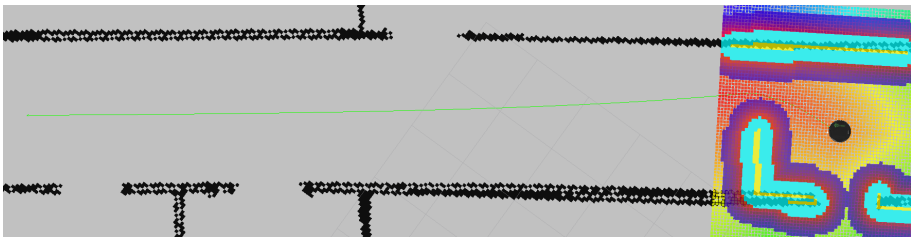
Fonte: Autor

APÊNDICE B – NAVEGAÇÃO COM OBSTÁCULOS

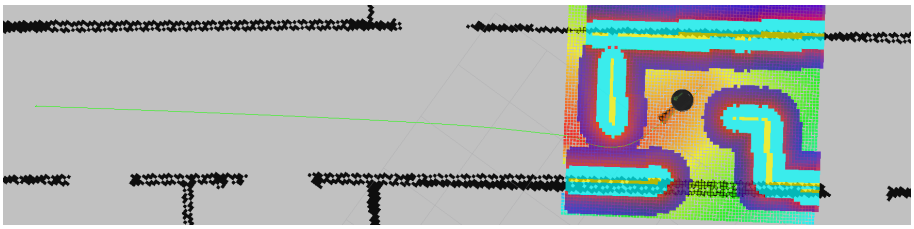
Figura 40 – Evolução do deslocamento do Turtlebot 2 em um ambiente com obstáculos



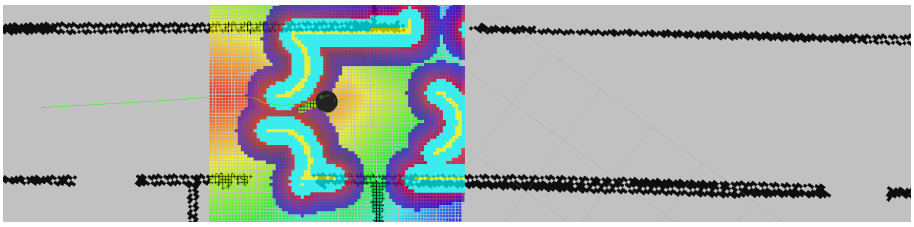
(a) Corredor com obstáculos a ser percorrido



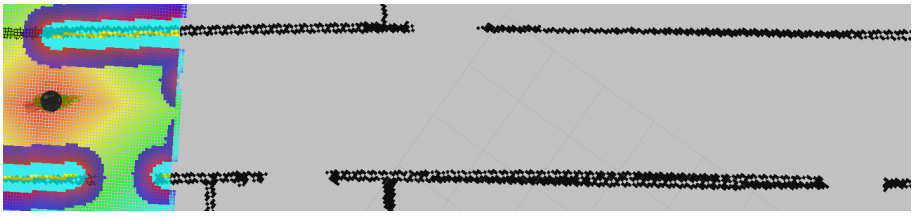
(b) Início



(c) Após superar o primeiro obstáculo



(d) Ao identificar o último obstáculo



(e) Ao alcançar seu objetivo

Fonte: Autor

APÊNDICE C – IMPLEMENTAÇÃO DA CLASSE ROSNODE

```
1  #include "../include/horus_patrol/ros_node.hpp"
2  namespace HorusPatrol
3  {
4
5  ROSNode::ROSNode(int argc,char** argv,const std::string &name):
6        init_argc(argc),
7        init_argv(argv),
8        manual_control_enabled(false),
9        patrolling_mode(false),
10       patrolling_mode_run(false),
11       linear_(0),
12       angular_(0),
13       patrolplace_it(0),
14       at_base(0),
15       streaming(0),
16       custom_move_requested(false),
17       turn360_requested(false),
18       custom_move_counter(0),
19       nrept(0),
20       managing_route(false),
21       robot_localized(false),
22       custom_goal(false),
23       threat_detected_(false),
24       node_name(name)
25 {}
26
27 //destructor
28 ROSNode::~ROSNode()
29 {
30   if(ros::isStarted())
31   {
32   ros::shutdown();
33   ros::waitForShutdown();
34   }
```

```
35 wait();
36 }
37
38 //LOCAL
39 bool ROSNode::init()
40 {
41  ros::init(init_argc,init_argv,node_name);
42  if (!ros::master::check())
43  {
44   return false;
45  }
46  rosCommsInit();
47  return true;
48 }
49
50 //NETWORK
51 bool ROSNode::init(const std::string &master_url, const std::
      string &host_url)
52 {
53  std::map<std::string,std::string> remappings;
54  remappings["__master"] = master_url;
55  remappings["__hostname"] = host_url;
56  ros::init(remappings,"horus_patrol");
57  if (!ros::master::check())
58  {
59  return false;
60  }
61  rosCommsInit();
62 }
63
64 void ROSNode::rosCommsInit()
65 {
66  ros::start();
67  ros::NodeHandle n;
68  image_transport::ImageTransport it(n);
69
70  log4cxx::Logger::getLogger(ROSCONSOLE_DEFAULT_NAME)->setLevel(
       ros::console::g_level_lookup[ros::console::levels::Warn]);
71  ros::console::notifyLoggerLevelsChanged();
```

```
72
73  vel_pub = n.advertise<geometry_msgs::Twist>("cmd_vel_mux/input/
        teleop", 5, true);
74  initial_pose_pub = n.advertise<geometry_msgs::
        PoseWithCovarianceStamped>( "/initialpose", 5 ,true);
75  goal_pub = n.advertise<geometry_msgs::PoseStamped>("/
        move_base_simple/goal", 5, true);
76  goal_cancel_pub = n.advertise<actionlib_msgs::GoalID>("/
        move_base/cancel", 5, true);
77  threat_pub = n.advertise<geometry_msgs::PointStamped>("/
        clicked_point", 5, true);
78  pose_sub = n.subscribe("/amcl_pose", 3, &ROSNode::PoseCallback,
         this);
79  goalresult_sub = n.subscribe("/move_base/result", 3, &ROSNode::
        GoalResultCallback, this);
80  kobuki_sub = n.subscribe("/mobile_base/events/bumper",3, &
        ROSNode::KobukiCallback, this);
81  kinect_rgb_sub = it.subscribe("/camera/rgb/image_raw", 3, &
        ROSNode::KinectRGBCallback, this);
82  kinect_depth_sub = it.subscribe("/camera/depth/image_raw", 3, &
        ROSNode::KinectDepthCallback, this);
83  odom_sub = n.subscribe("/odom", 3, &ROSNode::SensorCallback,
        this);
84  laser_scan_sub = n.subscribe("/scan",3, &ROSNode::
        LaserScanCallback, this);
85  velocities_sub = n.subscribe("cmd_vel_mux/input/navi",3, &
        ROSNode::VelocitiesCallback, this);
86  initial_pose_sub = n.subscribe("/initialpose", 3, &ROSNode::
        InitialPoseCallback, this);
87  goal_sub = n.subscribe("/goal", 3, &ROSNode::GoalTempCallback,
        this);
88  start();
89 }
90
91 void ROSNode::run()
92 {
93  ros::Rate rate(31);
94  while ( ros::ok() )
95  {
```

```
96      ros::spinOnce();
97      if(custom_move_requested)
98      {
99       customMovement();
100     }
101     if(turn360_requested)
102     {
103      turn360();
104     }
105     if( (manual_control_enabled)||
106     (custom_move_requested)||
107     (turn360_requested) )
108     {
109      rate.sleep();
110      publishVel(angular_,linear_);
111     }
112     ros::spinOnce();
113   }
114   //ros::spin();
115 }
116
117 //callback do sensor de toque do Turtlebot 2
118 void ROSNode::KobukiCallback(const kobuki_msgs::BumperEventPtr&
        msg)
119 {
120  Q_EMIT bumperEvent(msg->state);
121 }
122
123 //callback da odometria, calcula da distância e angulo acumulado
124 void ROSNode::SensorCallback(const geometry_msgs::
        PoseWithCovarianceStampedPtr& msg)
125 {
126  pose_x = msg->pose.pose.position.x;
127  pose_y = msg->pose.pose.position.y;
128  orientation = tf::getYaw(msg->pose.pose.orientation);
129  orientation = 3.14159 + orientation;
130  if( (custom_move_requested)||(turn360_requested) )
131  {
```

```
132    acc_dist = sqrt( pow( (pose_y - Start_Point_y), 2) + pow( (
           pose_x - Start_Point_x), 2) );
133    if( (!sentido_horario)&&(5*orientation < orientation_old) )
134    {
135     limit++;
136    }
137    if( (sentido_horario)&&(orientation > 5*orientation_old) )
138    {
139     limit--;
140    }
141    acc_angle = sqrt( pow((orientation - Start_Orientation
           +2*3.14159*limit),2) );
142    orientation_old = orientation;
143   }
144 }
145
146 //callback do pseudo sensor de laser
147 double right_dist, left_dist, dist, test;
148 void ROSNode::LaserScanCallback (const sensor_msgs::LaserScan::
       ConstPtr& msg){
149       dist = msg->ranges[320];
150       left_dist = msg->ranges[639];
151       right_dist = msg->ranges[1];
152
153       for(int i=270;i<=370;i++){
154           if( (msg->ranges[i]>0.45)&&(msg->ranges[i]<11) ){
155               if(dist > msg->ranges[i]) dist = msg->ranges[i];
156           }
157       }
158 }
159
160 //conversor de imagens no formato Mat para QImage
161 QImage ROSNode::cvtCvMat2QImage(const cv::Mat & image)
162 {
163   QImage qtemp;
164   if(!image.empty() && image.depth() == CV_8U)
165   {
166     const unsigned char * data = image.data;
167     qtemp = QImage(image.cols, image.rows, QImage::Format_RGB32);
```

```
168    for(int y = 0; y < image.rows; ++y, data += image.cols*image.
           elemSize())
169    {
170     for(int x = 0; x < image.cols; ++x)
171     {
172      QRgb * p = ((QRgb*)qtemp.scanLine (y)) + x;
173      *p = qRgb(data[x * image.channels()+2], data[x * image.
             channels()+1],
174          data[x * image.channels()]);
175     }
176    }
177   }
178   return qtemp;
179 }
180
181 //callback da câmera RGB do Kinect onde é feito a detecção da
      cor vermelha
182 void ROSNode::KinectRGBCallback(const sensor_msgs::ImageConstPtr
      & msg)
183 {
184      uchar *p;
185      cv::Mat thresh_image;
186      cv_bridge::CvImagePtr cv_ptr;
187      try
188      {
189          cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::
                 image_encodings::BGR8);
190
191      }
192      catch (cv_bridge::Exception& e)
193      {
194          ROS_ERROR("cv_bridge exception: %s", e.what());
195          return;
196      }
197      cv::inRange(cv_ptr->image, cv::Scalar(0,0,100), cv::Scalar
             (50,50,255), thresh_image);
198      cv::morphologyEx(thresh_image, threat_image, cv::MORPH_OPEN,
199          cv::getStructuringElement(cv::MORPH_RECT, cv::Size(7, 7)
                 ));
```

```
200       //captured_image = QPixmap::fromImage(cvtCvMat2QImage(
              threat_image));
201       captured_image = QPixmap::fromImage(cvtCvMat2QImage(cv_ptr->
              image));
202       Q_EMIT updateImage(&captured_image);
203   }
204
205   //callback da câmera RGB do Kinect onde é feito a detecção da
          ameaças
206   void ROSNode::KinectDepthCallback(const sensor_msgs::
          ImageConstPtr& msg)
207   {
208    if((!selected_route.isEmpty())&&(!manual_control_enabled))
209    {
210     if(!threat_image.empty())
211     {
212     uchar *p;
213     double n_points = 0, score = 0;
214     cv::Mat depth_mat;
215     cv_bridge::CvImagePtr cv_ptr;
216     try
217     {
218      cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::
             image_encodings::TYPE_32FC1);
219     }
220     catch (cv_bridge::Exception& e)
221     {
222      ROS_ERROR("cv_bridge exception: %s", e.what());
223      return;
224     }
225     cv::convertScaleAbs(cv_ptr->image, depth_mat, 100, 0.0);
226     QImage depth_image( depth_mat.data, depth_mat.cols, depth_mat.
          rows,
227     depth_mat.step, QImage::Format_Indexed8 );
228     static QVector<QRgb>  sColorTable;
229     for(short int i=0; i<256; ++i) sColorTable.push_back(qRgb(i, i
          , i));
230     depth_image.setColorTable(sColorTable);
231
```

```
232     for(short int i=0;i<480;++i)
233     {
234      p = threat_image.ptr<uchar>(i);
235      for(short int j=0;j<640;++j)
236      {
237       if(p[j]>0)
238       {
239        n_points++;
240        score += depth_image.pixelIndex(j,i);
241       }
242       else
243       {
244        depth_image.setPixel(j,i,0);
245       }
246      }
247     }
248     if(n_points > 3000)
249     {
250      double detection_thresh = 1.5 - log(score/n_points)/3.5; //
           linearização
251      if(detection_thresh < 0.03) detection_thresh = 0.02;
252      if( (!threat_detected_)&&(n_points/(480*640) >
           detection_thresh) )
253      {
254       threat_detected_ = true;
255       Q_EMIT threatDetected();
256      }
257     }
258     else
259     {
260      threat_detected_ = false;
261     }
262     }
263   }
264 }
265
266 //callback de localização do robô
267 void ROSNode::InitialPoseCallback(const geometry_msgs::
       PoseWithCovarianceStamped& msg)
```

```
268 {
269     robot_localized = true;
270     custom_initial_pose = msg;
271     initial_pose = custom_initial_pose;
272     Q_EMIT  robotLocalized();
273 }
274
275 //callback da posição estimada pelo amcl
276 void ROSNode::PoseCallback(const geometry_msgs::
      PoseWithCovarianceStampedPtr& msg)
277 {
278     custom_initial_pose = *msg;
279     if(!selected_route.isEmpty())
280     {
281       if ( (!at_base)&&( sqrt(pow( (msg->pose.pose.position.y -
            initial_pose.pose.pose.position.y), 2)
282          + pow( (msg->pose.pose.position.x - initial_pose.pose.
               pose.position.x), 2)) < 0.3 ) )
283       {
284          at_base = true;
285          Q_EMIT updatePatrolInfoBox(42);
286       }
287       if ( (at_base)&&( sqrt(pow( (msg->pose.pose.position.y -
            initial_pose.pose.pose.position.y), 2)
288       + pow( (msg->pose.pose.position.x - initial_pose.pose.pose
            .position.x), 2)) > 0.3 ) )
289       {
290          at_base = false;
291        }
292     }
293 }
294
295 //callback para publicar o goal enviado pela interface
296 void ROSNode::GoalTempCallback(const geometry_msgs::
      PoseStampedPtr& msg)
297 {
298     goal_pub.publish(msg);
299 }
300
```

```
301 //callback para controle dos movimentos de uma sequência de
       patrulha
302 void ROSNode::GoalResultCallback(const move_base_msgs::
       MoveBaseActionResultPtr& msg)
303 {
304     if( (patrolling_mode_run)&&(!turn360_requested) )
305     {
306         if(msg->status.status == 3)
307         {
308             Q_EMIT TakeAScreenShot();
309             if(patrolplace_it < total_patrol_places)
310             {
311                 if( (patrolplace_it!=0)&&(!custom_goal) )
312                 {
313                     turn360_requested = true;
314                 }
315                 patrol_pose.pose.position.x = patrol_places_x[
                       patrolplace_it];
316                 patrol_pose.pose.position.y = patrol_places_y[
                       patrolplace_it];
317                 patrol_pose.pose.orientation = tf::
                       createQuaternionMsgFromYaw(
                       patrol_places_orientation[patrolplace_it]);
318                 ros::Duration(1).sleep();
319                 goal_pub.publish(patrol_pose);
320                 patrolplace_it++;
321             }
322             else if( patrolplace_it == total_patrol_places )
323             {
324                 ros::Duration(1).sleep();
325                 moveToInitialPosition();
326             }
327             else
328             {
329                 patrolplace_it = 0;
330                 Q_EMIT enablePatrolSystem(true);
331             }
332             Q_EMIT updatePatrolInfoBox(3);
333         }
```

```
334     }
335     else if(msg->status.status == 3)      //Custom movement
336     {
337         patrolplace_it++;
338         Q_EMIT updatePatrolInfoBox(3);
339     }
340     if( (msg->status.status == 3)&&(custom_goal) )
341     {
342         custom_goal = false;
343     }
344     if(msg->status.status == 4)
345     {
346         updatePatrolInfoBox(4);
347     }
348 }
349
350 //callback para coleta das velocidades do robô
351 void ROSNode::VelocitiesCallback(const geometry_msgs::TwistPtr&
      msg)
352 {
353     if( (!manual_control_enabled)||(!turn360_requested) )
354     {
355         Q_EMIT updateLinearVelocity(floor(msg->linear.x * 100) /
              100);
356         Q_EMIT updateAngularVelocity(floor(msg->angular.z * 100)
              / 100);
357     }
358 }
359
360 //setup do sistema de patrulha
361 void ROSNode::patrolSystemSetup()
362 {
363     setInitialPoses();
364     patrolling_mode_run = false;
365     initial_pose_pub.publish(initial_pose);
366 }
367
368 //set pose inicial
369 void ROSNode::setInitialPoses()
```

```
370 {
371     initial_pose.header.frame_id = "map";
372     initial_pose.header.stamp = ros::Time::now();
373     if(patrol_initial_pose_name == "Custom Initial Pose")
374     {
375         initial_pose = custom_initial_pose;
376     }
377     else
378     {
379         initial_pose.pose.pose.position.x =
              patrol_initial_pose_x;
380         initial_pose.pose.pose.position.y =
              patrol_initial_pose_y;
381         initial_pose.pose.pose.position.z = 0;
382         initial_pose.pose.pose.orientation = tf::
              createQuaternionMsgFromYaw(
              patrol_initial_pose_orientation);
383     }
384     // set first goal pose
385     patrol_pose.header.frame_id = "map";
386     patrol_pose.header.stamp = ros::Time::now();
387     patrol_pose.pose.position.x = patrol_places_x[patrolplace_it
          ];
388     patrol_pose.pose.position.y = patrol_places_y[patrolplace_it
          ];
389     patrol_pose.pose.position.z = 0;
390     patrol_pose.pose.orientation = tf::
          createQuaternionMsgFromYaw(patrol_places_orientation[
          patrolplace_it]);
391 }
392
393 //função para mover o Turtlebot para a posição inicial
394 void ROSNode::moveToInitialPosition()
395 {
396     patrol_pose.pose.position.x = initial_pose.pose.pose.
          position.x;
397     patrol_pose.pose.position.y = initial_pose.pose.pose.
          position.y;
```

```
398     patrol_pose.pose.orientation = initial_pose.pose.pose.
            orientation;
399     patrolplace_it = total_patrol_places + 1;
400     actionlib_msgs::GoalID goal_id;
401     goal_cancel_pub.publish(goal_id);
402     goal_pub.publish(patrol_pose);
403 }
404
405 //função para mover o Turtlebot para uma posição da sequência
406 void ROSNode::moveToPosition(int index)
407 {
408     patrol_pose.pose.position.x = patrol_places_x[index];
409     patrol_pose.pose.position.y = patrol_places_y[index];
410     patrol_pose.pose.orientation = tf::
            createQuaternionMsgFromYaw(patrol_places_orientation[
            index]);
411     actionlib_msgs::GoalID goal_id;
412     goal_cancel_pub.publish(goal_id);
413     usleep(1000);
414     goal_pub.publish(patrol_pose);
415 }
416
417 //rotacionar o Turtlebot 2 360
418 void ROSNode::turn360()
419 {
420     switch(custom_move_counter)
421     {
422     case 0:
423         move('r');
424         custom_move_counter = 1;
425         break;
426     case 1:
427         if(acc_angle >= 2*3.14159)
428         {
429             custom_move_counter = 0;
430             move('s');
431             turn360_requested = false;
432         }
433         break;
```

```
434     }
435 }
436
437 //implementa uma máquina de estados para a função customMove
438 void ROSNode::customMovement()
439 {
440  if(nrept != 4*nlaps)
441  {
442      switch(custom_move_counter)
443      {
444      case 0:
445          move('f');
446          custom_move_counter = 1;
447          break;
448      case 1:
449          if( (acc_dist >= side1)&&(!(nrept%2)) )
450          {
451              if(!clock_rotation)
452              {
453                  move('r');
454              }
455              else
456              {
457                  move('l');
458              }
459              custom_move_counter = 2;
460          }
461          else if( (acc_dist >= side2)&&(nrept%2) )
462          {
463              if(!clock_rotation)
464              {
465                  move('r');
466              }
467              else
468              {
469                  move('l');
470              }
471              custom_move_counter = 2;
472          }
```

```
473             break;
474         case 2:
475             if(acc_angle >= 3.14159/2 + sup_turn +
                    vertices_rotation*2*3.141)
476             {
477                 nrept++;
478                 custom_move_counter = 0;
479                 move('s');
480             }
481             break;
482         }
483     }
484     else
485     {
486         custom_move_requested = false;
487         custom_move_counter = 0;
488         nrept = 0;
489         Q_EMIT customMoveEnded(true);
490     }
491 }
492
493 //função para movimentar o Turtebot 2
494 void ROSNode::move(char direction)
495 {
496     if(direction != 'o')
497     {
498         angular_ = linear_ = 0;
499     }
500     acc_angle = 0;
501     acc_dist = 0;
502     sentido_horario = false;
503     Start_Point_x = pose_x;
504     Start_Point_y = pose_y;
505     Start_Orientation = orientation;
506     orientation_old = orientation;
507     limit = 0;
508     switch(direction)
509     {
510     case 'f':
```

```
511             linear_ = 0.5;
512             break;
513         case 'b':
514             linear_ = -0.5;
515             break;
516         case 'l':
517             angular_ = 0.7;
518             break;
519         case 'r':
520             angular_ = -0.7;
521             sentido_horario = true;
522             break;
523         case 'o': //other speed
524             if(angular_ < 0)
525             {
526                 sentido_horario = true;
527             }
528             break;
529         case 's':
530             angular_ = 0;
531             linear_ = 0;
532             break;
533         }
534 }
535
536 //publicar as velocidades solicitadas
537 void ROSNode::publishVel(double angular, double linear)
538 {
539     vel.angular.z = angular;
540     vel.linear.x = linear;
541     vel_pub.publish(vel);
542     return;
543 }
544 }
```