Gabriel Medeiros Coelho

Interface web para monitoramento e controle de segurança com robôs utilizando ROS

Natal – RN

Dezembro de 2020

Gabriel Medeiros Coelho

Interface web para monitoramento e controle de segurança com robôs utilizando ROS

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

Orientador: Prof. Dsc. Bruno Marques Ferreira da Silva

Universidade Federal do Rio Grande do Norte – UFRN

Departamento de Engenharia de Computação e Automação – DCA

Curso de Engenharia de Computação

Natal – RN

Dezembro de 2020

À minha família, por todo o apoio, motivação e carinho oferecidos durante meu processo de graduação. Meu amor por eles é imensurável.

AGRADECIMENTOS

Agradeço, primeiramente, a Deus, por me abençoar todos os dias e por ser, através da fé, meu porto seguro.

Agradeço aos meus pais, Sandra e José, e aos meus irmãos, Daniel, Raphael e Gabriela, por todo o suporte e carinho.

Agradeço aos meus companheiros de curso, em especial a Josué e Alexandre, que trilharam essa jornada comigo durante todos os momentos. Sem vocês eu não chegaria até aqui.

Agradeço ao meu orientador, Bruno Silva, por toda a paciência e os aprendizados compartilhados comigo na etapa final desse ciclo.

Agradeço, por fim, aos professores da Universidade Federal do Rio Grande do Norte, que me moldaram como profissional e me ensinaram valores que vou levar para toda a vida.

"Porque para Deus nada é impossível"
(Lucas 1:37)

RESUMO

Esse trabalho tem como objetivo implementar um sistema web para monitoramento e controle de segurança de um local utilizando múltiplos robôs. Na implementação, foram usadas tecnologias disponibilizadas pelo ROS - *Robot Operating System* - para simular um ambiente sendo patrulhado por robôs, bem como fazer todo o processo de comunicação entre uma interface web e o sistema. Primeiramente, foi desenvolvido um servidor para controlar as ações de segurança e realizar a comunicação com os robôs. Uma vez que o servidor estava pronto, uma interface web foi construída para poder interagir com o servidor e representar visualmente, em um navegador, as ações dos robôs e o mapa do sistema sendo patrulhado. Verificou-se, por fim, que com o avanço da comunicação, dos navegadores e dos sistemas web, é possível realizar todo o controle de segurança de um ambiente por meio do uso de robôs e de uma simples tela de computador ou de celular. Para a simulação, foi utilizado um software chamado Gazebo, com robôs do tipo turtlebot. O servidor foi desenvolvido usando a linguagem de programação Python. Por fim, na interface web foi utilizado a linguagem de programação JavaScript, bem como um servidor websocket e um protocolo chamado Rosbridge, para fazer a comunicação com o servidor.

Palavras-chaves: ROS. Múltiplos robôs. Segurança. Sistema web. Rosbridge

ABSTRACT

This work aims to implement a web system to monitor and control the security of a environment using multiple robots. In the implementation, technologies provided by ROS - *Robot Operating System* - were used to simulate an environment being patrolled by robots, as well as to do the entire communication process between the web interface and the system. First, a server was implemented to control the security actions and to communicate with the robots. Once the server was ready, a web interface was built to be able to interact with it and show, in a browser, the actions of the robots and the map of the system being patrolled. Finally, it was concluded that, nowadays, with the advancement of communication, browsers and web systems, it is possible to perform all the security control of an environment through the use of robots and a simple computer or cell phone screen. For the simulation, a software called Gazebo was used, with turtlebot robots. The server was developed using the Python programming language. Finally, the JavaScript programming language was used in the web interface, as well as a websocket server and a protocol called Rosbridge, to communicate with the server.

Keywords: ROS. Multiple Robots. Security. Web system. Rosbridge

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

ROS	*Robot Operating System*
RaaS	*Robot as a Service*
HTML	*Hypertext Markup Language*
URL	*Uniform Resource Locator*
HTTP	*Hypertext Transfer Protocol*
AMCL	*Adaptive Monte Carlo Localization*
SLAM	*Simultaneous Localization and Mapping*
JSON	*JavaScript Object Notation*

SUMÁRIO

1 INTRODUÇÃO

No decorrer dos últimos anos, aplicações com robôs vem se tornando cada vez mais frequentes [26]. Um dos motivos para isso é o surgimento de tecnologias que facilitam o desenvolvimento de softwares com robôs, como, por exemplo, o ROS [22], um middleware que atua similarmente à um sistema operacional, abstraindo algumas informações complexas e de baixo nível e fornecendo um conjunto de softwares e bibliotecas que facilitam a comunicação entre os programas que irão rodar no sistema.

Esse crescimento acarretou no estudo e na criação de novos conceitos relativos ao processo de robotização. O RaaS - Robot as a Service - [12] é um exemplo disso. Segundo Koubaa [10], A principal ideia do RaaS é expor os recursos de software e hardware dos robôs por meio de um serviço na web, de modo que usuários consigam acessar, interagir e manipular os robôs por meio de um navegador. Para esse trabalho, o foco será desenvolver um serviço de sistema de segurança, bem como, prover uma interface web para o usuário poder interagir com o sistema.

1.1 Trabalhos Relacionados

Sistemas que trabalham com robôs para segurança buscam fornecer diferentes soluções, que variam desde ambientes domésticos [11] até comerciais [4].

A comunidade científica tem buscado cada vez mais evoluir nessa área. Um exemplo disso é o estudo feito em [33], que desenvolve um simulador de segurança exposto por meio de um serviço, onde é possível patrulhar com robôs, traçar rotas de emergência e reagir a acidentes, como incêndios e apagões.

Quando é pensado na comunicação com esses serviços por meio da internet, alguns problemas podem vir à tona, como a largura de banda do site e os atrasos na troca de mensagens pela rede. Pensando nesses problemas, Hu [7] desenvolve um novo modelo de comunicação com "robôs inteligentes", que aprendem a tomar decisões sozinhos com o tempo, fazendo com que a comunicação com o navegador web do cliente seja cada vez menos relevante.

1.2 Motivação

Empregar máquinas para realizar processos de vigilância e reconhecimento de intrusos apresenta algumas vantagens. Entre elas, pode-se citar a limitação de riscos ao pessoal e a redução de custos [28].

Além disso, levando em conta o poder dos navegadores web e o avanço da comunicação na internet nos últimos tempos e buscando facilitar a interação do usuário com o sistema, uma interface web, executada em um navegador, será implementada.

Ao final do trabalho, é esperado que o mesmo sirva como pontapé para a criação de outros sistemas que sejam baseados no conceito RaaS e que necessitem de uma interface web para comunicação com os robôs.

1.3 Objetivos

O objetivo do trabalho é construir um sistema de segurança, baseado no mapa de um local, com múltiplos robôs. O sistema deve ser capaz de reportar ocorrências no mapa para um cliente, bem como receber requisições do mesmo. Uma ocorrência pode ser entendida como uma atividade suspeita, como, por exemplo, intrusos transitando pelo ambiente. O cliente deve, por sua vez, possuir uma interface para monitorar a segurança do ambiente. Ademais, o cliente deve poder interagir com o sistema, colocando os robôs, individualmente, para investigar posições específicas no mapa e patrulhar determinadas áreas do mesmo.

Para o local a ser monitorado, será utilizado um cenário já pronto [1], que simula robôs do tipo turtlebot [17] usando os softwares Gazebo e Rviz. Todo o processo de comunicação no sistema será estabelecido por meio das bibliotecas e dos frameworks disponíveis no ROS.

1.4 Estrutura do Trabalho

Inicialmente, o Capítulo 2 aborda os principais tópicos referentes aos sistemas web, e ilustra o tipo de arquitetura e de comunicação que será usado no projeto. O Capítulo 3, por sua vez, exemplifica algumas principais características e funcionalidades do middleware ROS, tendo em vista que ele será a base para os algoritmos desenvolvidos. Logo após, vem o Capítulo 4, tratando sobre o Robot Web Tools e suas tecnologias que irão permitir a criação da interface web do sistema de segurança. Ao final do trabalho, é apresentado o Capítulo 5, com os detalhes da implementação, o Capítulo 6 demonstrando os experimentos que foram realizados e possíveis pontos de melhoria e, por fim o Capítulo 7, que traz as principais conclusões e contribuições deste trabalho.

2 SISTEMAS WEB

Neste capítulo, serão apresentadas as principais características de um sistema web, tendo em vista que entender seu funcionamento, sua arquitetura e seus protocolos de comunicação, é estritamente necessário para o desenvolvimento da interface do sistema de segurança.

2.1 World Wide Web

A World Wide Web, frequentemente abreviada para WWW ou, apenas, Web, foi criada com o intuito de ser uma biblioteca virtual, de páginas estáticas, de modo que pesquisadores da comunidade científica pudessem compartilhar documentos entre si por meio da rede [25].

Conforme pode ser visto em [27], páginas estáticas servem para mostrar um conteúdo fixo na tela e não dão suporte a interações com o usuário, diferindo do conceito de uma página dinâmica, que possui scripts rodando por trás da página responsáveis por mudar o conteúdo da mesma com base nas ações dos usuários.

Com o surgimento de novas tecnologias e novos protocolos de comunicação pela internet, o conceito de web evoluiu drasticamente. Não só pesquisadores, mas um número massivo de pessoas a utilizam atualmente. Páginas dinâmicas tem ganhado cada vez mais o espaço daquilo que inicialmente foi projetado para servir páginas estáticas. Motores de busca, bancos de dados, servidores e até mesmo outras aplicações web, podem ser consultadas em tempo real, garantindo que um conteúdo diferente seja retornado para o usuário, de acordo com os critérios especificados pelo mesmo. Em decorrências desses avanços, novos modelos de se estruturar uma aplicação surgiram, como, por exemplo, o modelo cliente-servidor.

2.2 Modelo cliente-servidor

Esse tipo de arquitetura constitui um sistema distribuído de máquinas, comunicando-se através de algum protocolo na rede (ver Figura 1). As máquinas que atuam como servidores providenciam o acesso a algum recurso ou serviço, que poderá ser requisitado por alguma máquina cliente. Uma ideia por trás desse modelo é que, ao invés de concentrar toda a carga de trabalho no servidor, esse processamento pode ser distribuído entre os diferentes servidores e clientes que estão interligados no sistema.

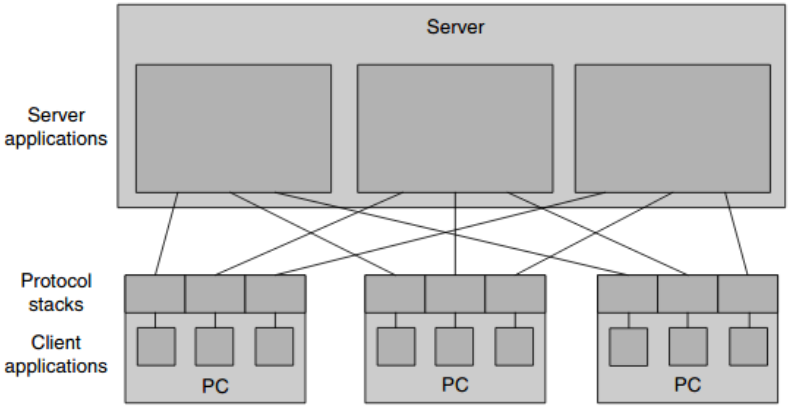

Figura 1 – Exemplo de aplicação cliente-servidor

Fonte – [25, p. 224]

Os sistemas web nada mais são do que sistemas que seguem esse modelo, onde, especificamente, o cliente interage com o servidor por meio de um browser (navegador). A ideia é que o browser realize requisições para o servidor, que irá interpretar essas requisições e devolver uma resposta para o cliente. O navegador se encarrega então de renderizar essa resposta, de uma forma visual, para o usuário.

De forma que todo esse processo ocorra, três componentes básicos devem estar presentes no sistema, conforme descrito em [25]:

- Uma linguagem de marcação capaz de formatar os documentos na página.
- Uma notação uniforme capaz de endereçar os recursos na rede, de forma que os mesmos consigam ser acessados.
- Um protocolo de comunicação responsável por transportar as mensagens na rede.

Em [9] é possível ver quais tecnologias atuam como esses componentes e é fornecido uma breve descrição para cada uma delas. São elas, respectivamente, o **HTML** [13], o **URL** [15] e o **HTTP** [14].

- HTML (Hypertext Markup Language)

  Consiste em uma linguagem de marcação, capaz de auxiliar o navegador em como os recursos devem aparecer na tela. Documentos escritos em HTML podem possuir links clicáveis para outros documentos e possuem instruções de como o navegador deve formatar e posicionar o conteúdo na página.

- URL (Uniform Resource Locator)

O URL nada mais é do que uma padronização para acessar os recursos disponíveis na internet. Ela é composta de três partes: a identificação do computador (tipicamente, o endereço IP do mesmo ou algum DNS), a identificação do recurso e qual será o protocolo de comunicação usado.

- HTTP (Hypertext Transfer Protocol)

  É um protocolo que especifica um conjunto de regras a serem seguidas para comunicação dos dados na web. Ele atua em um esquema de requisição-resposta entre cliente e servidor. Além disso, o protocolo especifica oito possíveis operações que podem ser usadas nas requisições: OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE, e CONNECT. Dentre elas, as principais são o GET e o POST, que servem, respectivamente, para acessar um determinado recurso de um URL e para mandar dados para o servidor.

2.3 Protocolo Websocket

O modelo cliente-servidor, apesar de ter sido amplamente adotado nos sistemas web, possuía um problema. Como visto anteriormente, ele se baseava em um protocolo no estilo requisição-resposta, onde todo o processo de comunicação originava-se de requisições feitas pelo cliente. Logo, não existia uma forma do servidor mandar informações sem que elas fossem requisitadas primeiro. Para superar essa dificuldade, algumas alternativas foram desenvolvidas. Dentre elas, destacam-se o HTTP long polling [6] e, mais recentemente, o protocolo **websocket** [8].

O protocolo websocket utiliza uma via de comunicação full-duplex entre um navegador e um servidor web. Isso significa que, através de uma mesma conexão, mensagens podem ser passadas tanto no sentido direto (cliente mandando para servidor), quanto no inverso. A comunicação websocket inicia por um processo conhecido como handshake, onde o navegador envia uma requisição solicitando a conexão para o servidor. Caso o servidor responda positivamente, a via de comunicação é estabelecida. Um exemplo de comunicação com websocket pode ser visto a seguir:

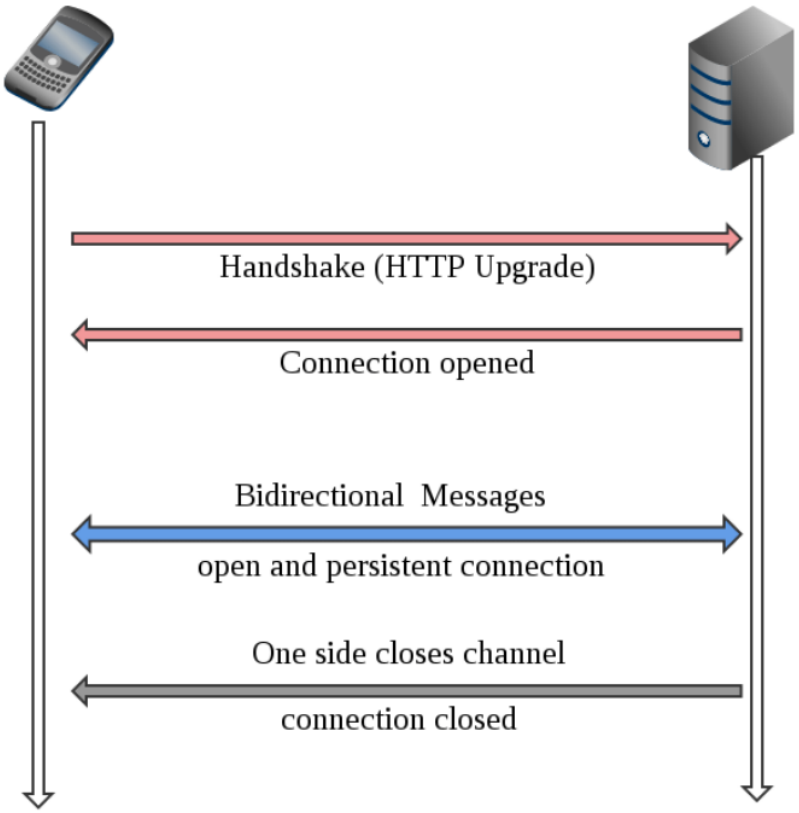

Figura 2 – Comunicação via websocket

Fonte – [16]

Uma vantagem do websocket sobre o long polling é a latência [18]. O protocolo http introduz uma latência a mais nas comunicações, devido à presença de inúmeros cabeçalhos nas mensagens trocadas, que fornecem informações adicionais durante a troca de recursos. Como o long polling é baseado no http, ele acaba sofrendo esse aumento. Isso pode ser um ponto negativo em aplicações que necessitem de informações em tempo real, como por exemplo, em um sistema de monitoramento, onde dados referentes à imagens de câmeras são atualizados a todo instante.

3 ROS

Neste capítulo, serão discutidos os pontos principais à respeito do ROS e de algumas tecnologias que ele disponibiliza, tendo em vista que todos os programas neste trabalho dependem desse middleware para comunicarem entre si.

3.1 Conceito

Conforme foi introduzido no Capítulo 1, o ROS é um middleware que fornece um conjunto de ferramentas que simplificam o desenvolvimento de software com robôs. Abstração de hardware, mecanismos de troca de mensagem e gerenciamento de pacotes são exemplos dessas ferramentas [24]. Ele foi criado em 2007 através de uma parceria entre o laboratório de inteligência artificial da Universidade de Stanford, a Willow Garage [34] e a Open Source Robotics Foundation [20]. Segundo [22], a principal motivação do projeto era remover as dificuldades iniciais intrínsecas ao desenvolvimento de software com robôs, criando um ecossistema de código fonte aberto e colaborativo que encorajasse o desenvolvimento de aplicações com robótica, onde laboratórios de pesquisas poderiam contribuir e compartilhar os seus trabalhos através do mesmo meio.

3.2 Componentes

Conforme mencionado em [19], o ROS possui cinco componentes principais em sua estrutura. São eles:

- Um conjunto de drivers para ler os dados de sensores e mandar comandos para atuadores (um motor por exemplo).
- Vários algoritmos fundamentais de robótica, para construir mapas do mundo, navegar neles, interpretar dados de sensores, planejar movimentos, manipular objetos, dentre outras atividades.
- Uma infraestrutura para fazer a conexão e a manipulação de dados entre os diversos programas rodando no sistema.
- Ferramentas que permitem visualizar o estado dos robôs, dos programas e os dados dos sensores, e que, além disso, facilitam o processo de debugar o código.

- Uma coleção de recursos que facilitam o desenvolvimento, incluindo um site com várias documentações, um fórum de perguntas e respostas e uma comunidade ativa de usuários e desenvolvedores.

3.3 Funcionamento

A troca de mensagens no ROS funciona no formato peer-to-peer. Isso quer dizer que a comunicação entre os programas ocorre diretamente, sem necessidade de passar por um servidor central para, só então, distribuir os dados. Devido à esse formato, um sistema ROS acaba apresentando uma estrutura em grafo, como pode ser visto na Figura 3, onde cada nó representa um programa e cada aresta é uma via de comunicação.

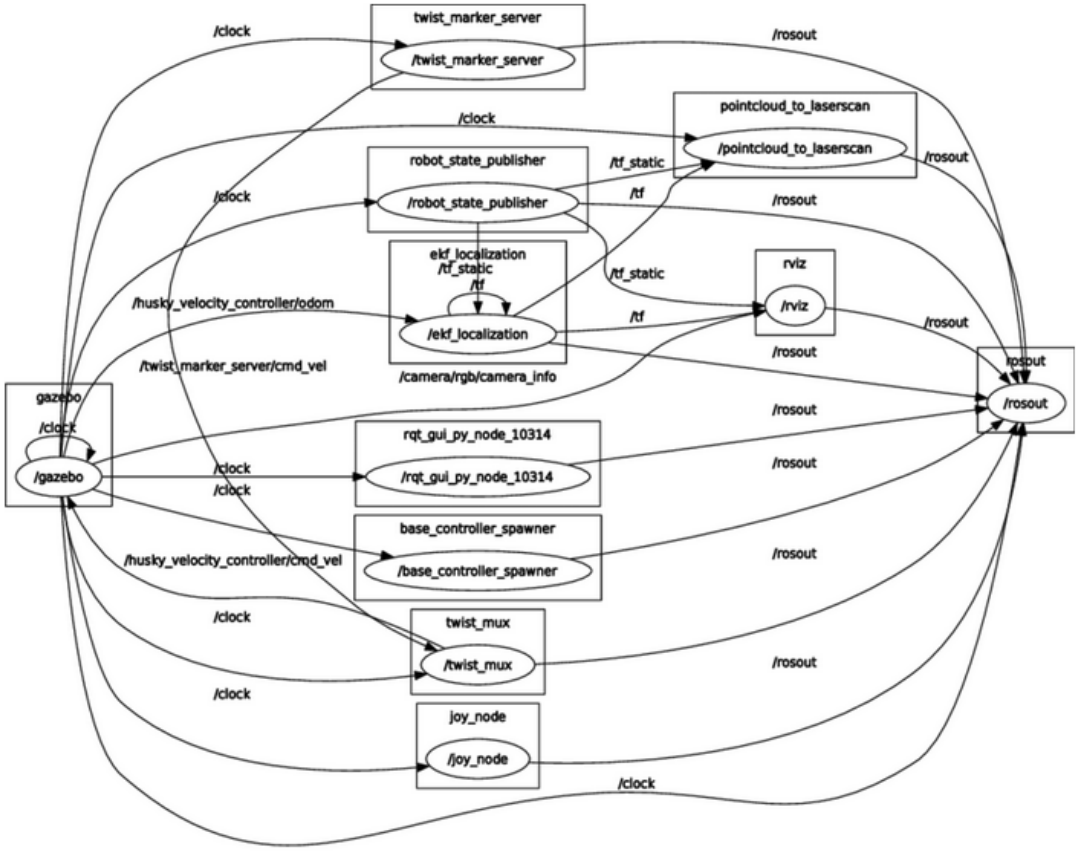

Figura 3 – Exemplo de um programa ROS

Fonte – [3]

Antes de qualquer sistema ROS rodar, é necessário inicializar o **roscore**. De uma maneira resumida, o roscore é um conjunto de programas responsáveis por criar as conexões entre os nós. Na figura acima, cada conexão pode ser entendida como um canal de comunicação, que, no ROS, pode ser de três tipos diferentes: tópicos, serviços ou ações.

Por fim, as mensagens trocadas devem obedecer uma descrição específica, estabelecida pelo próprio ROS.

3.4 Comunicação

3.4.1 Mensagens

Toda mensagem no ROS deve possuir um arquivo descrevendo-a. Esse arquivo deve conter o nome de todos os campos da mensagem, bem como seus respectivos tipos. Os tipos podem ser valores primitivos, como string, int8, float32, entre outros, ou, até mesmo, outras mensagens.

Um exemplo de mensagem ROS amplamente utilizado nesse trabalho, é a Pose, que irá ser utilizada pelo move_base para locomover o robô. Conforme definido em [21], a pose possui a seguinte forma:

```
Point position
Quaternion orientation
```

Ou seja, a mensagem possui dois campos: position e orientation, cujos tipos são, respectivamente, Point e Quaternion, que, por sua vez, são outras mensagens ROS.

3.4.2 Tópicos

Em um tópico, a troca de mensagens é no formato publisher-subscriber. Um mesmo tópico pode ter múltiplos nós inscritos e múltiplos nós publicando. Os programas que desejarem coletar dados do tópico, se inscrevem nele. Sempre que um programa publica no canal, todos os nós inscritos recebem o dado. É importante ressaltar o fato de que um tópico define o tipo de mensagem que irá trafegar por ele, de modo que todos os publishers e subscribers devem respeitar esse formato.

3.4.3 Serviços

Serviços funcionam de maneira similar a funções, dentro do contexto de programação. Nele, é definido um tipo de mensagem para representar a entrada da função e outro para representar a saída. A troca de dados em um serviço é similar ao modelo de comunicação com o protocolo http (ver seção 2.2): um nó cliente requisita a computação de um determinado serviço, passando para o mesmo a mensagem esperada como entrada. Um servidor, responsável por atender a chamadas desse serviço, interpreta a requisição, realiza a computação e, após finalizar, devolve o resultado para o cliente que, mais uma vez, deve respeitar a definição da mensagem de saída para o mesmo.

3.4.4 Ações

Ações são bem parecidas com serviços, porém, possuem duas características a mais: feedbacks e cancelamentos. Eles fornecem uma interface para o cliente cancelar requisições, que pode ser útil caso o servidor demore a responder. Além disso, é possível definir um tipo de mensagem de feedback que será passada pelo servidor, fornecendo informações periódicas para o cliente ter noção de como anda o progresso da requisição.

Em uma ação, a comunicação é controlada por um cliente de ação e um servidor de ação, que para este trabalho serão chamados de ActionClient e ActionServer. Eles fornecem uma interface entre os códigos definidos no lado do cliente e no lado do servidor, como pode ser visto na Figura 4.

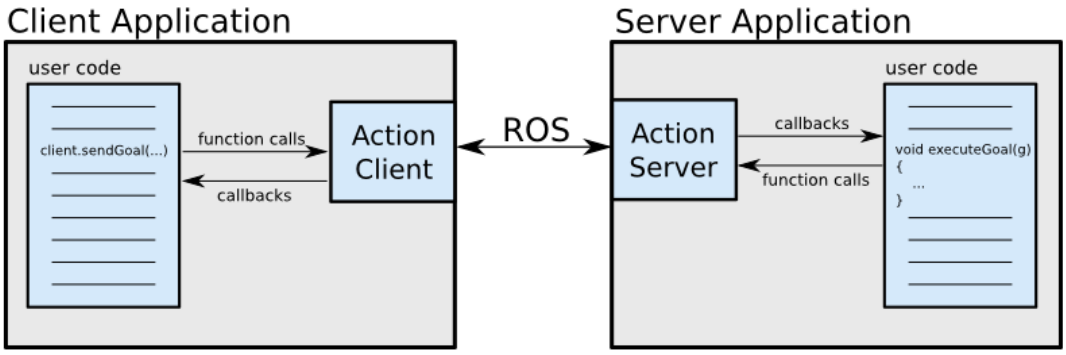

Figura 4 – Modelo de comunicação entre cliente e servidor utilizando ações

Fonte – [24]

Para que toda essa comunicação seja feita, é preciso definir três tipos de mensagens:

- **Goal:** especifica o tipo de mensagem da ação em si. Quando o cliente requisita uma ação do servidor, ele manda um Goal para o mesmo.
- **Feedback:** especifica o tipo de mensagem que será enviada periodicamente pelo servidor.
- **Result:** especifica o tipo de mensagem que o servidor irá mandar para o cliente, uma vez que a ação seja finalizada.

Essas mensagens são definidas dentro do projeto em um arquivo com a extensão .action, conforme mostra o modelo a seguir:

```
# Define the goal
uint32 dishwasher_id  # Specify which dishwasher we want to use
---
# Define the result
uint32 total_dishes_cleaned
---
# Define a feedback message
float32 percent_complete
```

Figura 5 – Estrutura de um arquivo .action

Fonte – [24]

3.5 Pacotes

O ROS possui uma organização feita em pacotes. Um pacote constitui um módulo, que pode conter nós, bibliotecas, arquivos de configuração ou outros arquivos e/ou softwares que são necessários para compor um programa. Além de fornecer suporte para o usuário poder criar seus próprios pacotes, o ROS já vem com alguns pacotes prontos em sua instalação. Para esse trabalho, a compreensão do AMCL, move_base e map_server são fundamentais.

3.5.1 AMCL

AMCL - *Adaptive Monte Carlo Localization* - é um sistema de localização 2D para os robôs. Ele é baseado em um mapa pré-definido, diferente de outros algoritmos mais complexos como o SLAM [2], que mapeia o cenário à medida que o robô se move. Quando o robô se desloca, o algoritmo AMCL tentar estimar sua posição e sua orientação no ambiente, através do uso de lasers scans.

3.5.2 move_base

Esse pacote fornece uma implementação de uma ação capaz de mover o robô. Para essa ação, as mensagens especificam qual a posição no mapa para onde o robô deverá se mover, bem como qual a orientação que ele deverá assumir.

De modo a realizar a movimentação do robô, o move_base trabalha com dois mapas de custo e dois planejadores, cada um sendo do tipo local ou global. A ideia de cada mapa de custo é fornecer informações sobre possíveis obstáculos com os quais o robô pode se deparar, enquanto que os planejadores leem essas informações e calculam a melhor rota para o robô seguir. O mapa de custo global é baseado nos obstáculos já definidos no mapa, e o local é calculado dinamicamente à medida que o robô se move, com base na leitura do seu laser. Uma exemplificação de como é feita a comunicação dos programas nesse pacote, pode ser vista a seguir:

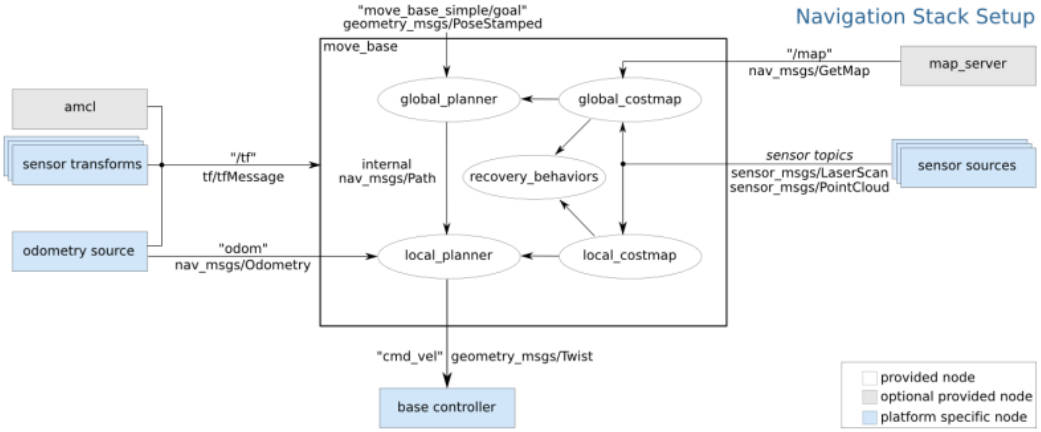

Figura 6 – Comunicação dos programas no move_base

Fonte – [24]

Um outro fator interessante do move_base é que, quando o robô fica preso em algum local do mapa, o nó de recovery_behaviour (ver Figura 7) entra em ação, para tentar colocar o robô de volta na rota.

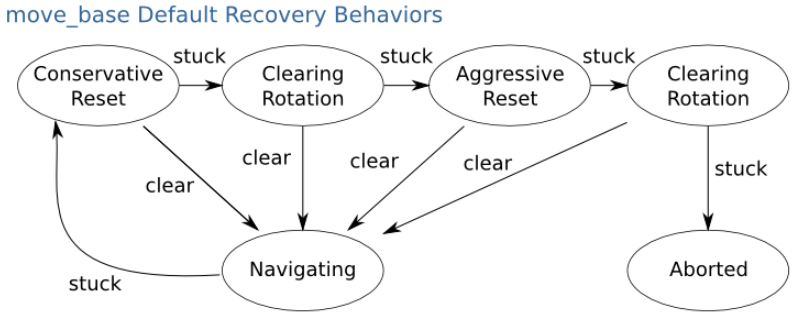

Figura 7 – Modo de Recovery Behaviour do move_base

Fonte – [24]

3.5.3 map_server

O map_server é o pacote responsável por fazer a leitura de um mapa, previamente computado por algum algoritmo, como o SLAM por exemplo, e disponibilizar seus dados para os outros nós. O mapa deve ser fornecido em um par de arquivos: um arquivo no formato YAML, representando alguns meta-dados do mapa, e um arquivo de imagem, que suporta a maioria dos formatos de imagens atuais.

O arquivo YAML fornece algumas informações cruciais para o map_server. Dentre seus parâmetros de configuração, existem cinco cuja presença é obrigatória:

- **image:** o caminho para o arquivo de imagem do mapa. Pode ser um caminho relativo ou absoluto.

- **resolution:** A resolução do mapa, no fator de metros/pixel.
- **origin:** A pose (posição e orientação) do pixel do canto inferior esquerdo do mapa, no formato (x, y, yaw), onde x e y representam as coordenadas nos respectivos eixos e yaw representa a rotação.
- **occupied_thresh:** Um fator para considerar no cálculo da ocupação dos pixeis. Qualquer probabilidade acima desse fator, considerará o pixel como totalmente ocupado.
- **free_thresh:** O exato oposto do occupied_thresh. Probabilidades abaixo desse fator, considerarão o pixel como totalmente livre.

Quando o map_server faz a leitura do arquivo de imagem, ele mapeia cada pixel em diferentes categorias, com base na sua cor: livre (0), ocupado (100), e desconhecido (-1). Após todo o processo, as informações relativas aos meta-dados são publicadas no tópico /map_metadata e o mapeamento de cada pixel, mencionado anteriormente, é publicado no tópico /map.

4 ROBOT WEB TOOLS

Neste capítulo, será feita uma pequena introdução ao Robot Web Tools e suas principais ferramentas, que foram imprescindíveis para o desenvolvimento da interface web deste trabalho.

4.1 Descrição

O site oficial [30] do Robot Web Tools apresenta a seguinte descrição:

> *"Robot Web Tools é uma coleção de módulos e ferramentas de código fonte aberto para criação de aplicações com robôs baseadas em web."[30, tradução nossa].*

Em outras palavras, esse conjunto de ferramentas cria uma interface de comunicação entre a web e um middleware para aplicações com robôs, como, por exemplo, o ROS. Utilizando o poder das linguagens HTML5 e Javascript, o Robot Web Tools define uma camada de transporte que será usada na troca de mensagens entre o navegador e o servidor da aplicação, bem como disponibiliza várias bibliotecas para expor funcionalidades ROS do lado do cliente. Essas tecnologias são definidas em um pacote ROS (ver seção 3.5) chamado de rosbridge_suite.

4.2 rosbridge_suite

Esse pacote define um protocolo de comunicação em sua camada de transporte, chamado de rosbridge protocol. O protocolo especifica que as mensagens trocadas devem estar no formato JSON. Uma vantagem disso é que, como o JSON é um formato de troca de dados universalmente difundido, qualquer cliente, independente da linguagem de programação que ele implementa, pode interagir com o ROS. O rosbridge protocol oferece suporte de várias funcionalidades, ROS, como inscrever-se e publicar em tópicos, chamar serviços, interagir com ações, dentre outras. Um exemplo de mensagem que segue essa forma de comunicação pode ser visto a seguir:

```
{ "op": "subscribe",
  "topic": "/cmd_vel",
  "type": "geometry_msgs/Twist"
}
```

Figura 8 – Formato de mensagem do rosbridge protocol

Fonte – [24]

O rosbridge_suite possui três implementações principais:

- **rosbridge_library:** Consiste em outro pacote, responsável por traduzir as mensagens do formato JSON para o formato padrão do ROS, ou vice-versa.
- **rosapi:** É responsável por expor as funcionalidades do ROS para um sistema não-ROS. O navegador web de um cliente, por exemplo, constitui esse tipo de sistema. Essa api possibilitou o surgimento das bibliotecas mencionadas na seção 4.1, dentre as quais, destacam-se, no escopo desse trabalho, a **roslibjs** e a **ros2djs**.
- **rosbridge_server:** Representa a camada de transporte. Na implementação padrão do pacote, essa camada é estabelecida por uma conexão websocket (ver seção 2.3), porém, segundo a página na wiki do rosbridge_server, esse tipo de conexão não é obrigatório e outros tipos de conexão podem ser implementados.

4.3 roslibjs

Uma biblioteca javascript que expõe as principais funcionalidades do ROS para o browser. Ela disponibiliza uma variável global chamada de **ROSLIB**, que possui várias classes diferentes, como pode ser visto na documentação do roslibjs [32]. Para a interface do sistema de segurança, as classes utilizadas foram:

- **ROS:** É a classe principal para fazer interface com o ROS. Além de gerenciar toda a conexão com o servidor websocket, é utilizada como referência por várias outras classes
- **Topic:** Cria uma conexão com algum determinado tópico. É através dessa conexão que o cliente pode publicar mensagens no tópico, ou inscrever-se nele para receber as mensagens publicadas pelos outros programas.
- **ActionClient:** Estabelece um canal com algum Action Server. Usando esse canal, o cliente pode mandar ações, cancelá-las, e solicitar feedback do servidor.
- **Goal:** Cria um Goal para ser enviado por meio de algum Action Client. O tipo de mensagem do goal deve ser compatível com o formato esperado pelo Action Client.
- **Vector3:** Constrói um vetor 3d de coordenadas, baseados nas posições do robô no mapa. É necessário fazer esse processo antes de construir uma mensagem do tipo Pose.
- **Quaternion:** Forma um objeto do tipo quaternion, usando as orientações do robo com relação ao mapa. Assim como o Vector3, é preciso fazer isso para criar uma mensagem Pose.

- **Pose:** Constrói uma mensagem do tipo Pose, que será lida pelo move_base (ver subseção 3.5.2).

4.4 ros2djs

Essa biblioteca é responsável por gerar um mapa 2D do local sendo navegado pelos robôs, bem como mostrar a posição de cada robô no mapa. De maneira análoga ao roslibjs, uma variável global com a implementação de várias classes é exposta ao cliente [31]. As classes usadas foram:

- **Viewer:** Desenha um mapa na interface, com as dimensões especificadas pelo usuário.
- **OccupancyGridClient:** Lê os dados de ocupação do mapa publicados no tópico /map (ver subseção 3.5.3) e preenche o Viewer.
- **NavigationArrow:** Desenha uma seta no mapa. Útil para representar a posição atual de cada robô.

4.5 nav2djs

É uma implementação melhorada do ros2djs, que, além de mostrar o cenário, também possui uma funcionalidade para navegar com os robôs [29]. Quando o usuário clica em determinado local do mapa, essa biblioteca cria uma mensagem do tipo Pose baseada nas coordenadas do clique e envia uma action para o move_base do robô, fazendo com que ele se desloque até o local. As principais classes utilizadas foram:

- **OccupancyGridClientNav:** Similar ao OccupancyGridClient do ros2djs, mas com alguns wrappers para que seja possível realizar a locomoção dos robôs.
- **Navigator:** Fornece os métodos necessários para mover os robôs. Cada robô possui seu próprio navigator.

5 IMPLEMENTAÇÃO

Esse capítulo irá mostrar o passo a passo da implementação do sistema de segurança, passando pelos pré-requisitos de instalação, ambiente de simulação, desenvolvimento do servidor e, por fim, desenvolvimento do cliente.

5.1 Instalação

O projeto foi baseado numa distribuição do ROS chamada de Kinetic Kame e desenvolvido em um sistema operacional Ubuntu na versão 16.04. Instruções sobre o processo de instalação do ROS Kinetic podem ser encontrados em [1].

Como mencionado anteriormente, para o ambiente que será monitorado foi utilizado um cenário já pronto, disponível em [1]. Esse cenário utiliza uma simulação de robôs turtlebot. Para essa simulação funcionar apropriadamente, duas dependências adicionais precisam ser instaladas, o que pode ser feito com os seguintes comandos:

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

Por fim, para a comunicação com o cliente da aplicação funcionar, é necessário instalar o rosbridge:

```
sudo apt-get install ros-kinetic-rosbridge-server
```

5.2 Arquitetura

A arquitetura do sistema segue o modelo cliente-servidor, descrito na seção 2.2, onde o cliente é implementado em um navegador web e o servidor é o próprio sistema de segurança. Além desses dois componentes, uma camada a mais de comunicação é necessária: a comunicação entre o sistema de segurança e cada um dos robôs. Um esquemático mostrando a arquitetura final, pode ser visto a seguir:

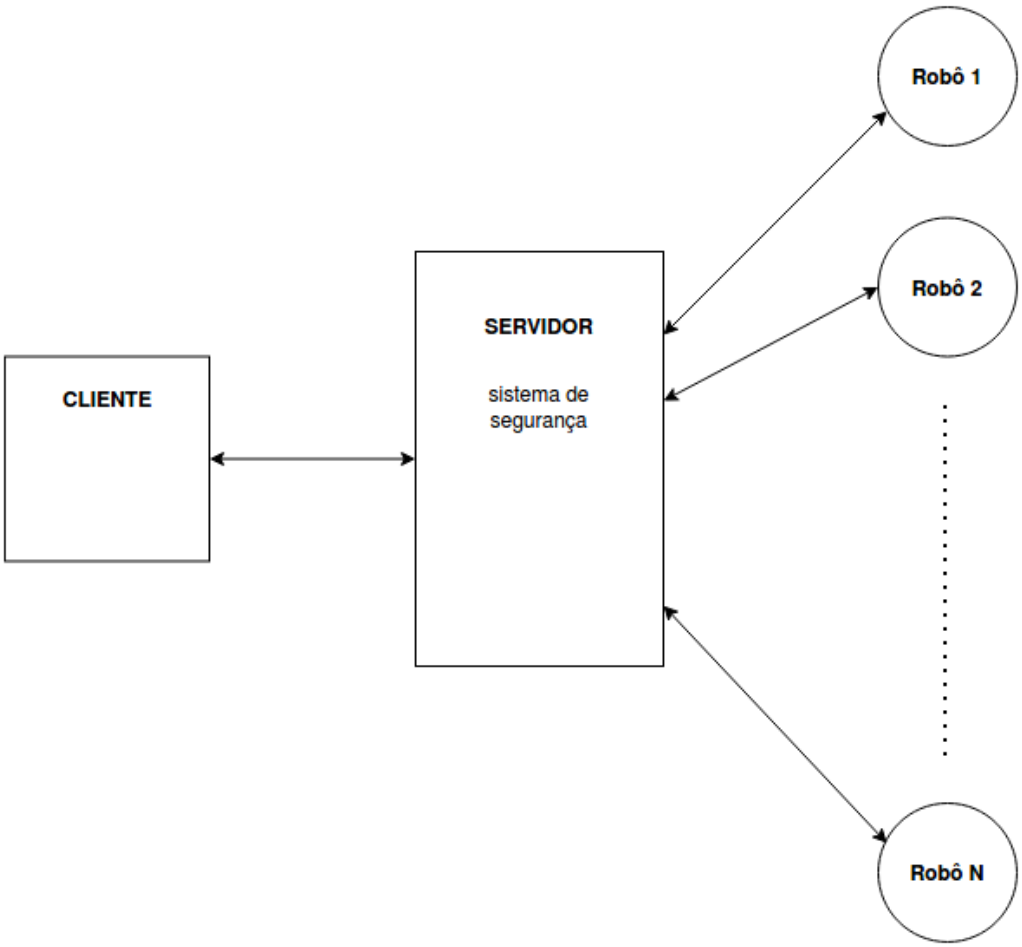

Figura 9 – Arquitetura do sistema de segurança

Fonte – Elaborada pelo autor

Para realizar as ações de navegação do robô, tanto a comunicação entre o cliente e o servidor quanto entre servidor e robôs é feita por meio de actions. Já para registro de ocorrências, é estabelecido um tópico. Como resultado final, os seguintes canais de comunicação são criados:

cliente-servidor

```
/robot1/web
/robot2/web
.
.
.
/robotN/web
/occurrence/report/channel
```

servidor-robôs

```
/robot1/patrol
/robot2/patrol
.
.
.
/robotN/patrol
```

5.3 Robôs

Na simulação do sistema de segurança desse trabalho, foram utilizados dois robôs, para efeito de simplicidade. O objetivo não foi implementar um sistema com o maior número de robôs possíveis, e sim, apenas demonstrar que existe um suporte para implementação com múltiplos robôs. Durante a execução de suas atividades, cada robô pode estar em um determinado estado diferente. São eles:

- OCIOSO

  Indica que o robô está parado, sem executar nenhum tipo de ação de segurança. Nesse estado, ele está completamente disponível para receber alguma ação do usuário.

- PATRULHANDO

  Caracteriza uma rotina automática de patrulhamento. O robô possui em sua memória uma lista de coordenadas para onde ele deve se mover. Sempre que chega em uma coordenada, ele se move para a próxima. Quando a lista acaba, ele repete o processo a partir da primeira posição.

- INDO INVESTIGAR

  Quando alguma atividade suspeita está acontecendo em determinado local do mapa, o usuário pode mandar algum robô investigar essa localização. Esse estado indica que o robô está se movendo para uma determinada localização solicitada pelo usuário.

- INVESTIGANDO

  Uma vez que o robô chega na localização para ser investigada, ele muda seu estado para investigando. Nessa situação, ele fica parado, semelhante ao comportamento ocioso, porém, em uma situação real, seria razoável pensar que alguns recursos a mais estariam ativos no robô, como, por exemplo, câmeras embutidas, sistemas de detecção facial, alarmes de segurança, dentre outros.

É importante destacar que, por se tratar de um sistema de segurança, cada segundo para tomar uma ação pode ser essencial. Logo, não importa qual é a atividade do robô, se ele

recebe um novo comando, o mesmo deve parar de executar qualquer ação e imediatamente executar a nova ação solicitada pelo usuário. A Figura 10 mostra a máquina de estados dos robôs e suas possíveis transições.

Na prática, cada robô descrito na Figura 9 é, também, um servidor, que deve ser capaz de manipular os estados de cada robô e interpretar os comandos recebidos. Esses servidores foram desenvolvidos utilizando a linguagem Python. Para a máquina de estados, foi usado o pacote smach [23]. Ademais, um Action Server (ver subseção 3.4.4) foi implementado para lidar com as ações provenientes do servidor.

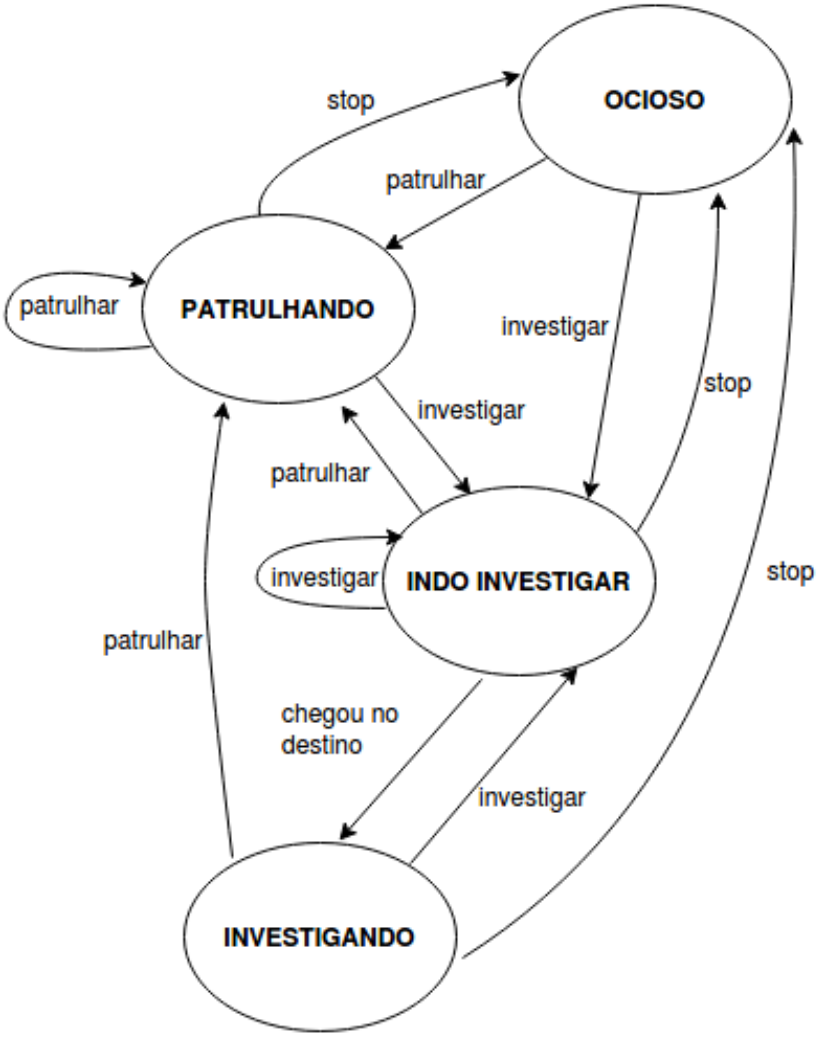

Figura 10 – Máquina de estados dos robôs

Fonte – Elaborada pelo autor

5.4 Robot Patrol

Os comandos enviados para os robôs foram implementados por meio de uma action chamada RobotPatrol. Para definir a action, foi criado um arquivo .action, disponível

no Apêndice A. Cada goal é composto por uma lista de mensagens do tipo Pose. Se a lista contém apenas um item, o robô irá investigar a posição. Caso a lista possua mais de um item, o robô irá patrulhar cada uma das posições sequencialmente. Por fim, uma lista vazia indica que o robô deve parar qualquer ação, resultando em um estado ocioso. Quando a action finaliza, ela manda um status como resultado para o cliente, indicando qual a ação que o robô passou a executar.

5.5 Servidor

Duas tarefas principais devem ser realizadas pelo servidor: reportar ocorrências no mapa para o cliente e desempenhar uma interface entre a comunicação do cliente com os robôs. Para atingir esses objetivos, foram criadas três threads que rodam simultaneamente. São elas:

- **ActionServerController:** Para cada robô, um ActionServerController é criado. Cada vez que o cliente manda ou cancela uma ação, esse controller é responsável por repassar o comando para os controllers de cada robô. Além disso, uma vez que os robôs finalizam suas ações, o ActionServerController deve enviar o resultado para o cliente.
- **RobotController:** Para cada robô, um RobotController é criado. Ele promove uma interface para repassar os comandos para os robôs e, uma vez que as ações são finalizadas, ele avisa algum controlador responsável pela comunicação com o cliente (que, para esse caso, é o ActionServerController).
- **OccurrenceController:** É responsável por reportar ocorrências para o cliente. Isso acontece através de um tópico chamado /occurrence_report_channel. As mensagens publicadas no tópico são coordenadas x e y do local no mapa onde a ocorrência aconteceu.

5.6 Cliente

Pensando em um sistema de segurança, o cliente web precisa ter uma interface mostrando o mapa do local sendo monitorado, a localização atual de cada robô, o local das ocorrências reportadas pelo servidor e o estado de cada um dos robôs. Ademais, ele deve poder interagir com o sistema, colocando os robôs, individualmente, para investigar posições específicas no mapa e patrulhar determinadas áreas do mesmo.

5.6.1 Navegação

Para realizar as ações de navegação com o robô (investigar e patrulhar), foi utilizado como base o código fonte da biblioteca nav2djs (ver seção 4.5), porém, alguns trechos desse código precisaram ser re-escritos devido a dois fatores:

- Só existia suporte para navegação com apenas um robô. Por se tratar de um sistema com múltiplos robôs, foi necessário ajustar a implementação de algumas funções para refletir essa necessidade.
- A biblioteca inicializava um Action Client diretamente com o Action Server do move_base dos robôs. Para a arquitetura desse trabalho, isso não deve ser feito. O Action Client precisa ser estabelecido primeiro com o servidor do sistema de segurança, para que o mesmo seja responsável por passar (ou não) as ações para os robôs. Por fim, as actions disparadas precisam ser do tipo RobotPatrol, então isso também foi ajustado. O Action Client resultante pode ser visto na Figura 11.

```
// setup the actionlib client
var securityActionClient = new ROSLIB.ActionClient({
  ros : ros,
  actionName : 'multi_robots_security_system/RobotPatrolAction',
  serverName : '/robot' + (index+1) + '/web'
});
```

Figura 11 – Action Client para realizar a navegação com os robôs

Fonte – Elaborada pelo autor.

5.6.2 Ocorrências

As ocorrências são registradas através de um tópico, conforme mostra a Figura 12. Sempre que uma nova ocorre, ela é representada no mapa por uma marcação. As mensagens trocadas pelo tópico são do tipo Coordinate2D, que possuem dois valores float: x e y, relativos à coordenada da ocorrência no mapa.

```
var occurrenceListener = new ROSLIB.Topic({
  ros : this.ros,
  name : '/occurrence_report_channel',
  messageType : 'multi_robots_security_system/Coordinate2D',
  throttle_rate : 100
});
```

Figura 12 – Tópico para receber ocorrências do servidor

Fonte – Elaborada pelo autor.

5.6.3 Interface

Algumas outras tecnologias foram usadas para ajudar na disposição dos elementos da página de uma maneira geral. Seus nomes e suas respectivas funções são mencionados a seguir:

- **Bootstrap:** Ajuda na estilização dos componentes da página.
- **Jquery:** Manipula o quadro sobre os avisos de ocorrências recebidas pelo servidor.
- **VueJS:** Na aplicação desse trabalho, alguns elementos da página precisam ser dinâmicos, como, por exemplo, o estado atual dos robôs. Para atingir isso, foi utilizado o framework VueJS. De maneira resumida, cada elemento dinâmico é representado por uma variável. Quando o valor dessa variável muda, o VueJS detecta essa mudança e renderiza novamente o trecho HTML responsável por mostrar esse valor, sem precisar recarregar a página. Além disso, o framework em questão possui algumas diretivas que facilitam a construção dos componentes na tela. Por exemplo, ao invés de replicar o código dos cards de cada robô (ver Figura 13), a diretiva de v-for foi utilizada, para compilar vários itens de uma vez com base em uma lista.

A interface final está representada na Figura 13. Ela possui três seções principais: o mapa, os cards de cada robô e os logs da aplicação. O usuário pode selecionar um robô clicando em seu respectivo card. Com o robô selecionado, é possível realizar três ações:

- patrulhar, clicando no botão verde "Start Patrol".
- parar, clicando no botão vermelho "Stop".
- investigar, clicando em alguma localização no mapa.

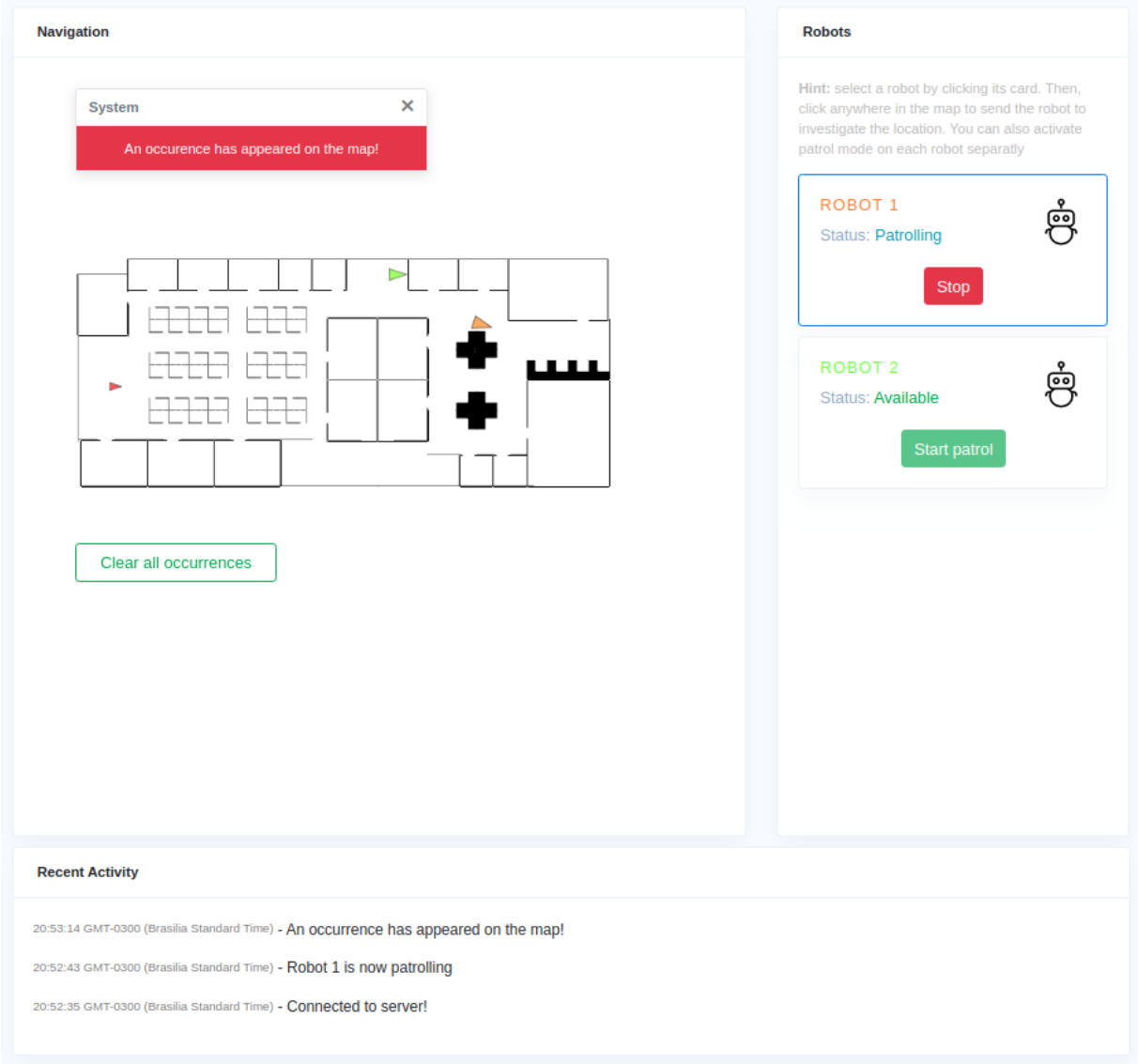

Figura 13 – Interface do sistema de segurança

Fonte – Elaborada pelo autor.

Vale ressaltar que existe uma validação para o aparecimento de cada botão. Por exemplo, não faz sentido mostrar o botão de Stop para o robô 1 se o mesmo já estiver ocioso.

Sempre que uma ocorrência é reportada pelo servidor, uma marca mostrando sua localização aparece no mapa e uma mensagem vermelha surge no topo da tela. O usuário pode limpar as marcas desenhadas no mapa clicando no botão "Clear all occurrences".

Na seção de logs, são mostradas informações como: aparecimento de ocorrências, ações realizadas por robôs, estado de conexão com o ROS, etc.

Todos os arquivos que desempenham as funções mencionadas nesse capítulo, acompanhados de instruções para rodar o projeto, estão disponíveis em [5].

6 EXPERIMENTOS

Esse capítulo tem como objetivo mostrar algumas imagens e vídeos do sistema de segurança funcionando e discutir sobre os parâmetros utilizados nas simulações.

6.1 Cenário

Reforçando o que foi explicado antes, todo o cenário desse trabalho foi baseado no projeto desenvolvido em [1]. Quando a simulação é iniciada, dois softwares são executados: Gazebo e RViz.

6.1.1 Gazebo

Esse software mostra em uma perspectiva 3D o ambiente da simulação, conforme ilustra a Figura 14. É Possível visualizar as paredes, obstáculos, os cômodos e ver os robôs se movimentando no cenário.

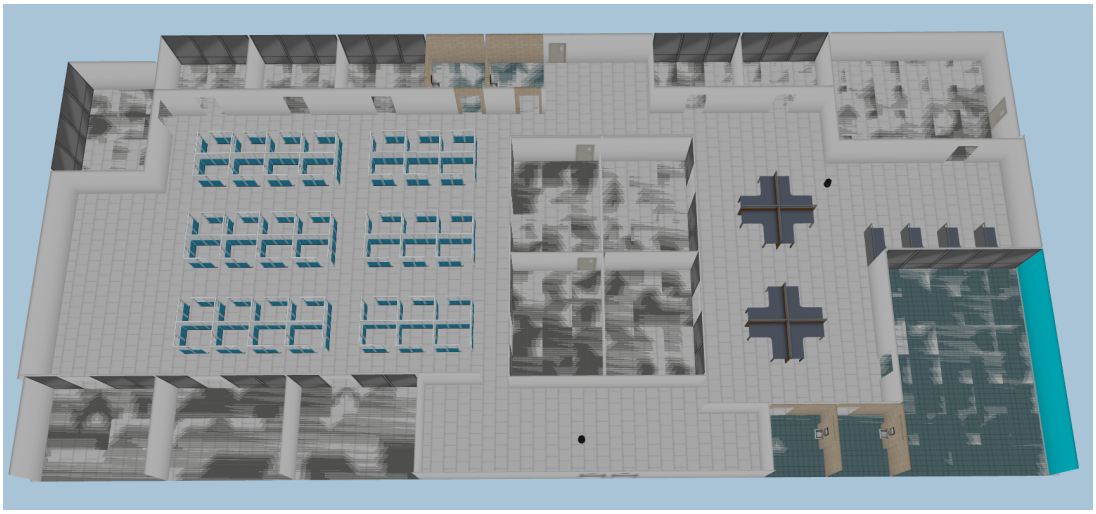

Figura 14 – Ambiente simulado no Gazebo

Fonte – Elaborada pelo autor

6.1.2 Rviz

O mesmo cenário também é representado no Rviz, porém, do ponto de vista dos robôs. Ele mostra as leituras dos sensores dos robôs e uma linha indicando a rota que eles irão tomar, o que se tornou bem útil, pois, à medida que eram testados comandos enviados pelo cliente, era possível visualizar no software se os robôs de fato estavam se locomovendo

para as coordenadas esperadas. Na Figura 15, é possível ver, sob a perspectiva do RViz, os robôs se movimentando e os seus destinos finais.

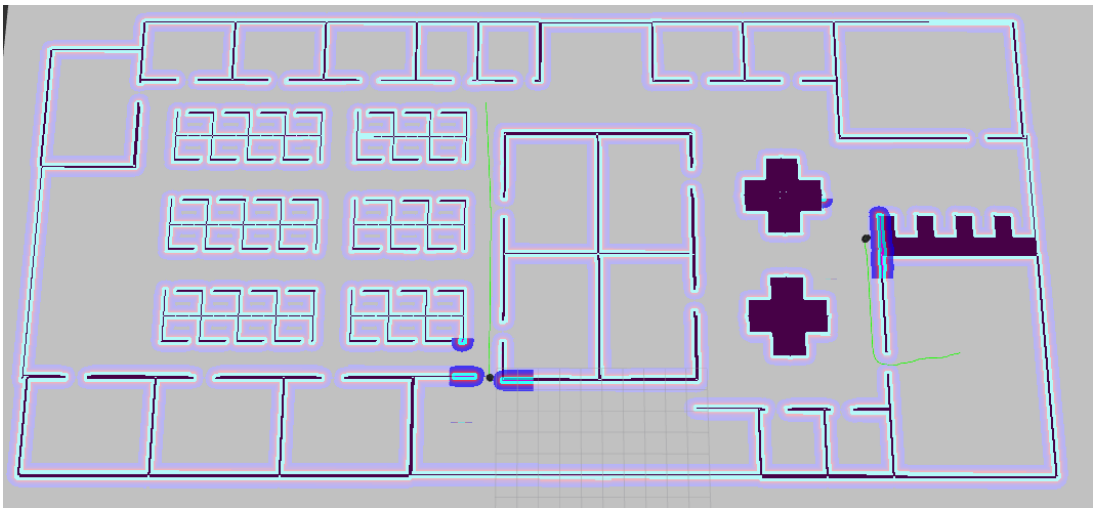

Figura 15 – Ambiente simulado no RViz

Fonte – Elaborada pelo autor

6.2 Cliente

De modo a realizar os experimentos no cliente, foi utilizado o navegador Google Chrome em sua versão 87, com o auxílio de um servidor local de testes, para representar localmente um site rodando na web.

Para simular ocorrências, a cada cinco segundos, era rodado um programa no servidor que gerava um número aleatório. Se o número fosse menor que 0.05, ou seja, com uma chance de 5% de acontecimento, o servidor escolhia uma coordenada de uma lista com lugares pré-definidos e publicava essa ocorrência em um tópico. O cliente escutava esse tópico e, assim que recebia uma nova informação, desenhava a ocorrência no mapa, representada por uma seta vermelha. Na Figura 16 é possível ver várias ocorrências no mapa.

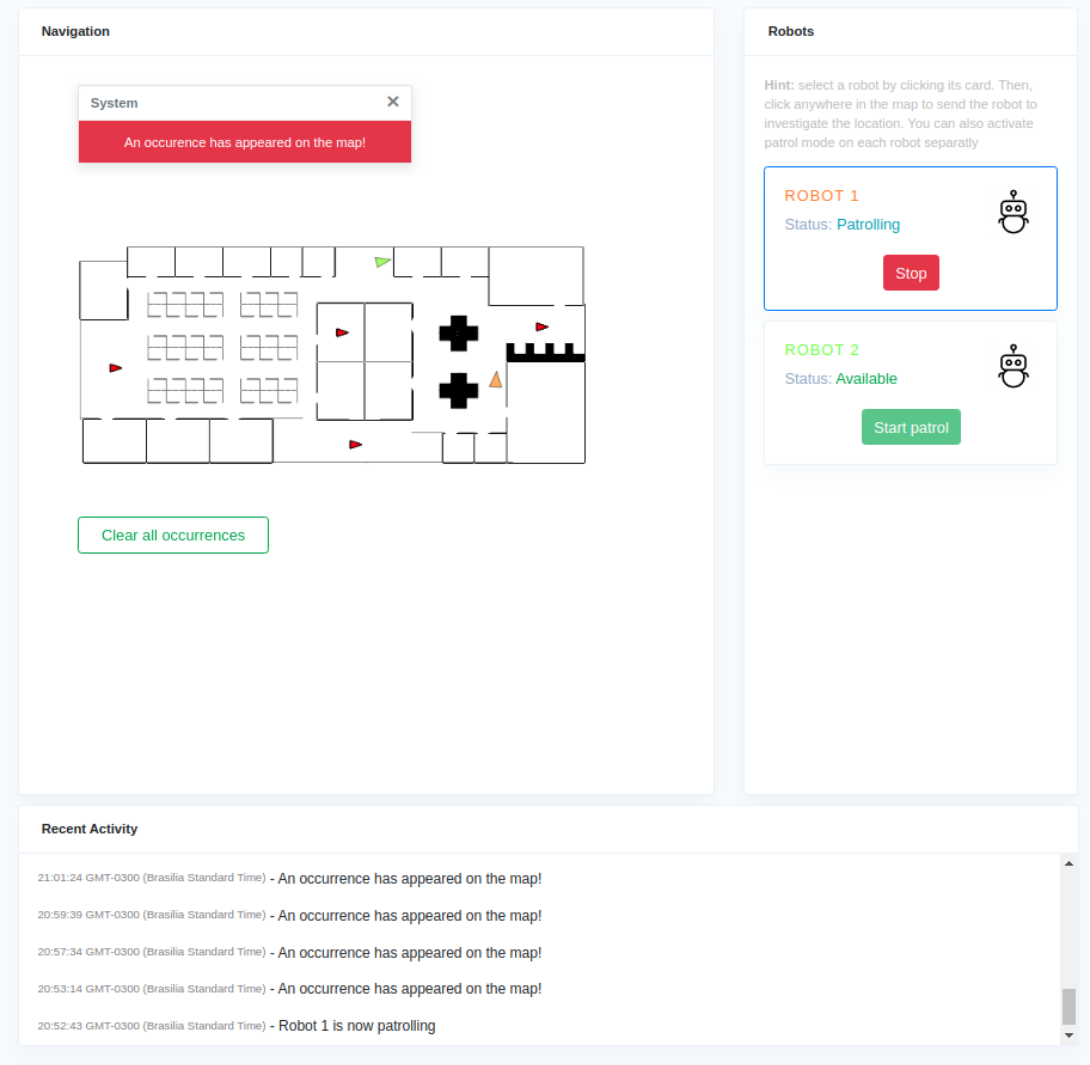

Figura 16 – Ocorrências no mapa do cliente web

Fonte – Elaborada pelo autor

Ambos os robôs foram deixados em modo patrulha por 8 horas. Foram usadas como base quatro posições para cada robô, definidas no código. O robô 1 foi designado para patrulhar o lado direito do mapa, ao redor das duas mesas em formato de cruz, enquanto que o robô 2 patrulhou o lado esquerdo, em torno das seis mesas retangulares. Após todo o período de patrulha, verificou-se que ambos continuavam em atividade, sem apresentar nenhum tipo de colisão ou problema. Dois vídeos representando o sistema em funcionamento podem ser vistos em:

- Interface do cliente e ambiente de simulação

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

- Interface do cliente, logs do servidor e máquina de estados dos robôs

  https://www.youtube.com/watch?v=f260OQCfnkU

6.3 Possíveis melhorias

6.3.1 Interface

Algumas variáveis do projeto são escritas manualmente no código, como por exemplo as coordenadas para os robôs patrulharem e o endereço do servidor websocket. Uma possível melhoria no trabalho seria inserir campos de input na interface, para que o próprio usuário consiga entrar com esses valores, o que tornaria a aplicação mais dinâmica. Além disso, seria visualmente agradável para o usuário se ele pudesse ver linhas no mapa representando as rotas dos robôs, de maneira similar a como é mostrado no RViz.

6.3.2 Robot Patrol

A action RobotPatrol, utilizada para fazer a navegação do robô nesse trabalho, só possui mensagens de Goal e de Result. Uma melhoria no sistema seria implementar as mensagens de Feedback, que podem ser particularmente úteis para enviar periodicamente ao cliente algumas informações a respeito da patrulha dos robôs, por exemplo, qual foi último local patrulhado, que horas isso ocorreu, qual vai ser o próximo local a ser patrulhado, dentre outros fatores.

6.3.3 Aplicação real

A ideia inicial do trabalho era fornecer uma base de estudo para que futuramente um sistema de segurança real possa ser aplicado na UFRN. De modo que isso ocorra, o cenário utilizado nas simulações precisa ser substituído por mapas reais dos ambientes internos da universidade. Além disso, em um sistema de segurança, poder navegar com robôs e mandar eles se moverem para determinado local é importante, porém não suficiente. Para que a aplicação atinja um patamar em que ela possa ser colocada na prática, outros módulos são necessários, como, por exemplo, câmeras de segurança embutidas nos robôs e algoritmos de detecção facial. A interface por sua vez, precisaria de uma nova seção para mostrar a imagem dessas câmeras, e outras informações que agreguem informação para o usuário, como o nível de bateria dos robôs e das câmeras, por exemplo.

7 CONCLUSÃO

O ROS se demonstrou uma ferramenta poderosa para desenvolver aplicações com robôs. Utilizando as suas tecnologias, foi possível implementar, em pouquíssimo tempo, um sistema de segurança com um módulo de navegação e fazer toda a comunicação entre cliente, servidor e robôs.

As simulações apresentaram resultados bem satisfatórios. Garantindo uma conexão estável através da rede, seria possível colocar os robôs em modo de patrulha por oito horas seguidas. Isso poderia caracterizar um turno da madrugada, por exemplo, que, por ser o horário menos movimentado, possivelmente é o que a segurança mais precisa ser reforçada. Para atingir mais horas seguidas de patrulha, poderia ser implementado uma rotina no robô para que ele detectasse quando sua bateria estivesse baixa e fosse, de forma autônoma, até um ponto de energia carregar a si mesmo, podendo então voltar as suas atividades.

Por fim, aproveitando a arquitetura e as formas de comunicação desse trabalho, outras aplicações de robô de serviço, seguindo o modelo cliente-servidor e que necessitem de uma interface web do lado cliente, poderiam ser desenvolvidas. Outra opção seria aprimorar o sistema já existente, introduzindo novos módulos de segurança, conforme descrito na subseção 6.3.3, para que, futuramente, o mesmo possa ser colocada em prática em um ambiente interno da Universidade.

REFERÊNCIAS

[1] Josué Araújo. multi_robot_sim. https://github.com/josuearaujo/multi_robot_sim, 2019. Acessado em: 2020-12-02. 12, 27, 35

[2] T. Bailey and H. Durrant-Whyte. Simultaneous localization and mapping (slam): part ii. *IEEE Robotics Automation Magazine*, 13(3):108–117, 2006. 21

[3] Héctor Cadavid, Alexander Pérez Ruiz, and Camilo Rocha. Reliable control architecture with plexil and ros for autonomous wheeled robots. pages 611–626, 08 2017. 18

[4] Cobalt. Cobalt Robotics. https://www.cobaltrobotics.com/. Acessado em: 2020-12-02. 11

[5] Gabriel Coelho. multi_robots_security_system. https://github.com/gabrielmcoelho/multi_robots_security_system, 2020. Acessado em: 2020-12-01. 34

[6] Joe Hanson. What is HTTP Long Polling? https://www.pubnub.com/blog/http-long-polling/. Acessado em: 2020-12-03. 15

[7] Huosheng Hu, Lixiang Yu, Pui Wo Tsui, and Quan Zhou. Internet-based robotic systems for teleoperation, Jun 2001. 11

[8] IETF. The WebSocket Protocol. https://tools.ietf.org/html/rfc6455. Acessado em: 2020-12-03. 15

[9] M. Jazayeri. Some trends in web application development. In *Future of Software Engineering (FOSE '07)*, pages 199–213, 2007. 14

[10] Anis Koubaa. A service-oriented architecture for virtualizing robots in robot-as-a-service clouds. In Erik Maehle, Kay Römer, Wolfgang Karl, and Eduardo Tovar, editors, *Architecture of Computing Systems – ARCS 2014*, pages 196–208, Cham, 2014. Springer International Publishing. 11

[11] Gašparík Marek and Šolek Peter. Design the robot as security system in the home. *Procedia Engineering*, 96:126 – 130, 2014. Modelling of Mechanical and Mechatronic Systems. 11

[12] Bernard Marr. Robots As A Service: A Technology Trend Every Business Must Consider. https://www.forbes.com/sites/bernardmarr/2019/08/05/

robots-as-a-service-a-technology-trend-every-business-must-consider/?sh=4dd569b424ea. Acessado em: 2020-12-02. 11

[13] MDN. HTML. https://developer.mozilla.org/pt-BR/docs/Web/HTML. Acessado em: 2020-12-03. 14

[14] MDN. HTTP. https://developer.mozilla.org/pt-BR/docs/Web/HTTP. Acessado em: 2020-12-03. 14

[15] MDN. URL. https://developer.mozilla.org/pt-BR/docs/Web/API/URL. Acessado em: 2020-12-03. 14

[16] Lei Mu. Multi-purpose web framework design based on websocket over http gateway. 09 2016. 16

[17] OSRF OpenSourceRoboticsFoundation. Turtlebot 2. https://www.turtlebot.com/turtlebot2/. Acessado em: 2020-12-02. 12

[18] V. Pimentel and B. G. Nickerson. Communicating and displaying real-time data with websocket. *IEEE Internet Computing*, 16(4):45–53, 2012. 16

[19] Morgan Quigley, Brian Gerkey, and William D. Smart. *Programming Robots with ROS: A Practical Introduction to the Robot Operating System*. O'Reilly Media, Inc., 1st edition, 2015. 17

[20] Open Robotics. Open Robotics. https://www.openrobotics.org/. Acessado em: 2020-12-04. 17

[21] ROS. Pose Message. http://docs.ros.org/en/jade/api/geometry_msgs/html/msg/Pose.html. Acessado em: 2020-12-05. 19

[22] ROS. ROS Org. https://www.ros.org/about-ros/. Acessado em: 2020-12-02. 11, 17

[23] ROS. ROS smach. http://wiki.ros.org/smach. Acessado em: 2020-12-05. 30

[24] ROS. ROS Wiki. http://wiki.ros.org/. Acessado em: 2020-12-02. 17, 20, 21, 22, 24

[25] Leon Shklar and Richard Rosen. Web application architecture, 2003. 13, 14

[26] Manshi Shukla and Amar Nath Shukla. Growth of robotics industry early in 21st century, Sep 2012. 11

[27] WP Amelia Staff. Static vs Dynamic Website: What Is the Difference? https://wpamelia.com/static-vs-dynamic-website/. Acessado em: 2020-12-03. 13

[28] T. Theodoridis and H. Hu. Toward intelligent security robots: A survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(6):1219–1230, 2012. 11

[29] Robot Web Tools. nav2djs documentation. http://robotwebtools.org/jsdoc/nav2djs/current/. Acessado em: 2020-12-05. 26

[30] Robot Web Tools. Robot Web Tools. http://robotwebtools.org/. Acessado em: 2020-12-05. 24

[31] Robot Web Tools. ros2djs documentation. http://robotwebtools.org/jsdoc/ros2djs/current. Acessado em: 2020-12-05. 26

[32] Robot Web Tools. roslibjs documentation. http://robotwebtools.org/jsdoc/roslibjs/current. Acessado em: 2020-12-05. 25

[33] Wei-Han Hung, P. Liu, and Shih-Chung Kang. Service-based simulator for security robot. In *2008 IEEE Workshop on Advanced robotics and Its Social Impacts*, pages 1–3, 2008. 11

[34] Wikipedia. Willow Garage. https://pt.wikipedia.org/wiki/Willow_Garage. Acessado em: 2020-12-04. 17

APÊNDICE A – ROBOTPATROL.ACTION

```
# Goal
# patrol_poses: an Array of poses.
    # If it only has one pose, the robot will go investigate that pose and stay there.
    # If it has more than one pose, the robot will go check sequentially to each pose on the list, repeating the process after the last pose.

geometry_msgs/PoseArray patrol_poses

---
# Result
# status: status of the robot (what it is doing at the moment).
string status

---
# Feedback
# this action doesn't provide any feedback.
```