UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO
DEPARTAMENTO DE SISTEMAS DE COMPUTAÇÃO

Programação de Robôs Móveis - SSC0712
Prof. Dr. Matheus Machados dos Santos

Uma introdução ao Robot Operating System

Autores:	**N°USP:**
Ana Rita Marega Gonçalves	15746365
Beatriz Aparecida Diniz	11925430
Felipe de Oliveira Gomes	14613841
Gabriel dos Santos Alves	14614032

30 de junho de 2025

Sumário

1 Apresentação

Robot Operating System, abreviado para ROS, é o software com ferramentas e rotinas voltadas e organizadas para o desenvolvimento prático de aplicações robóticas. Em outras palavras, consiste em um framework de prototipagem na área de robótica.

A proposta do ROS é ser um ecossistema para o avanço de pesquisas e implementações em robótica, pois, além de sua comunidade de desevolvedores e usuários, constitui um acervo de programas "de código aberto" (*open source*), ou seja, que são publicados para que possam ser reutilizados e adaptados para os mais diversos fins por outros usuários.

O ROS é lançado na forma de distribuições. Cada uma delas apresenta ferramentas e métodos próprios, além de questões de compatibilidade com outros programas e sistemas operacionais. Atualmente, se destacam duas distribuições por serem versões *LTS* (*long-term support*): o ROS2 Jazzy Jalisco e o ROS2 Humble Hawksbill. Como é possível notar pelas figuras a seguir, a tartaruga é uma espécie de mascote.

Figura 1 – Logo do ROS2 Jazzy Jalisco

Figura 2 – Logo do ROS2 Humble Hawksbill

2 Conceitos do ROS

O ROS é uma poderosa plataforma para o desenvolvimento de aplicações robóticas, baseada em uma arquitetura de processamento distribuído em rede *peer-to-peer*, na forma de um grafo. Para aproveitar todo o seu potencial, é fundamental compreender os principais conceitos que estruturam sua comunicação e organização interna.

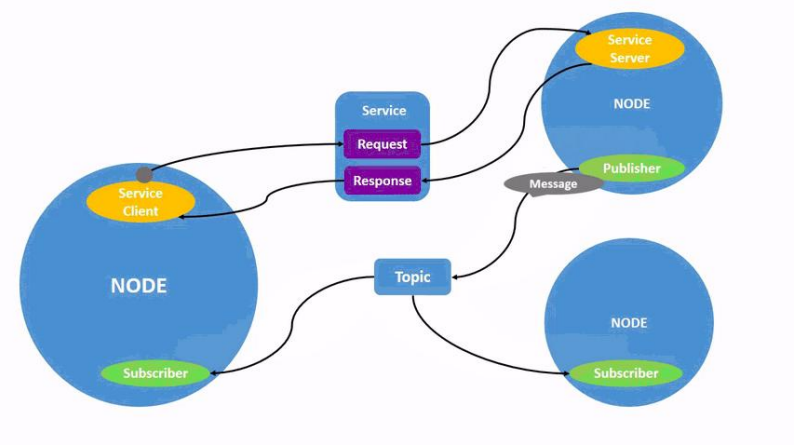

Figura 3 – Grafo de comunicação do ROS, disponível em sua documentação

2.1 Nodo

O ROS adota uma arquitetura modular, na qual sistemas robóticos complexos são divididos em múltiplos **nodos**. Cada nodo representa um processo em execução, dedicado a uma função específica dentro do sistema. Esta abordagem facilita o desenvolvimento, manutenção e reutilização de componentes de software.

Como exemplo, em um robô móvel, um nodo pode ser responsável pelo controle das rodas, enquanto outro executa a leitura de sensores. Essa modularização permite que diferentes partes do sistema operem de forma independente e paralela, aumentando a robustez e a flexibilidade do robô.

Os nodos no ROS comunicam-se entre si por meio de tópicos, serviços, ações ou parâmetros. Dessa maneira, cada nodo pode enviar ou receber dados utilizando esses mecanismos, colaborando para o funcionamento integrado do sistema robótico.

Em geral, um nodo corresponde a um processo único, desenvolvido em C++ ou Python. Embora seja possível que um processo execute múltiplos nodos, na prática, costuma-se adotar a abordagem de um processo por nodo.

2.2 Tópicos

No ROS, **tópicos** funcionam como um barramento de comunicação que permite a troca de informações entre diferentes nodos do sistema. Cada tópico é identificado por um nome exclusivo e serve como um canal por onde as mensagens são transmitidas.

A comunicação via tópicos utiliza o padrão "anuncia/publica/se inscreve" (*advertise/publish/subscribe*). Um nodo pode publicar informações em um tópico específico, enquanto outros nodos podem assinar esse mesmo tópico para receber os dados publicados.

As mensagens trocadas por meio de tópicos são fortemente tipadas, ou seja, seguem definições explícitas nos arquivos de descrição de mensagem do *ROS*. Isso

garante que apenas dados compatíveis sejam compartilhados, promovendo segurança e integridade na comunicação.

Além disso, a troca de informações por tópicos é caracterizada por ser não bloqueante. Ou seja, o envio ou recebimento de mensagens não depende do status imediato dos demais nodos, oferecendo maior flexibilidade e desempenho para sistemas robóticos distribuídos.

2.3 Serviços

Serviços no ROS utilizam o paradigma de requisição e resposta para a troca de informações entre dois nodos. Diferente dos tópicos, a comunicação por serviços é bloqueante: o nodo que faz a requisição aguarda pela resposta do nodo servidor antes de continuar sua execução. Esse mecanismo é indicado para operações pontuais e síncronas no sistema robótico.

2.4 *Actions*

No ROS, as **ações** são utilizadas para executar tarefas de maior duração, permitindo acompanhamento contínuo e possibilidade de cancelamento. Cada ação envolve um objetivo, um *feedback* durante a execução e um resultado ao finalizar. O mecanismo de ações combina características de tópicos e serviços, proporcionando comunicação assíncrona com monitoramento em tempo real da execução, diferentemente dos serviços tradicionais.

2.5 *Bag*

No ROS, o formato de arquivo ***bag*** é utilizado para armazenar mensagens trocadas entre nodos. Ele representa o principal mecanismo de registro e armazenamento de dados no sistema. As *bags* permitem criar *datasets* a partir de robôs reais, possibilitando a reprodução de experimentos em laboratório sem a necessidade do robô físico. Entre as principais ferramentas estão o `rosbag` e o `rqt_bag`.

2.6 Arquivos de Lançamento (*Launch Files*)

Os ***launch files*** do ROS são arquivos fundamentais que definem quais nós devem ser executados e como devem ser inicializados para que o sistema de um ou mais robôs entre em operação. Eles podem ser organizados de forma hierárquica, onde cada arquivo é responsável por inicializar uma parte específica do sistema, composta por múltiplos nós. Por exemplo, um *launch file* pode inicializar todos os nós relacionados aos drivers de um robô, responsáveis pela aquisição de dados, enquanto outro pode iniciar o sistema de planejamento e navegação.

Além disso, é possível criar estruturas flexíveis de arquivos de lançamento que se adaptam a diferentes situações. Um mesmo sistema pode operar tanto em um simulador quanto no hardware real. Nesse contexto, um *launch file* pode ser configurado para iniciar apenas os nós de simulação, enquanto outro executa exclusivamente os nós dos drivers do hardware real. Um principal pode decidir qual versão deve ser carregada, dependendo do contexto (simulação ou experimento real).

No ROS1, os arquivos de lançamento eram escritos exclusivamente em XML. Já no ROS 2, os *launch files* podem ser escritos em XML, YAML ou Python. O Python, em particular, traz maior flexibilidade, pois permite executar comparações lógicas, laços e verificações de condições para inicializar os nós de acordo com critérios específicos, algo que não é possível implementar apenas com XML.

3 Configuração de Ambiente

Antes de iniciar o desenvolvimento com o ROS, é imprescindível preparar corretamente o ambiente de trabalho. Essa preparação envolve desde a instalação do sistema operacional e das dependências essenciais, até a definição de variáveis de ambiente e a organização dos diretórios de trabalho (***workspaces***).

A seguir, são apresentados os passos recomendados para configurar o seu ambiente, assegurando o bom funcionamento do sistema tanto em simulações quanto na aplicação em hardware real. Foi utilizado como referência o ROS2 Humble funcionando em Ubuntu 22.04.

3.1 Arquivo `setup.bash`

Toda vez que um terminal for aberto com o intuito de executar um comando da distruibuição ROS, é necessário que seja executado:

```
$ source /opt/ros/humble/setup.bash
```

Para otimizar esse processo e garantir que todo terminal execute esse comando ao inicializar, execute uma única vez o comando:

```
$ echo "source /opt/ros/humble/setup.bash" >> ~/.bashrc
```

Toda vez que desejar interagir com *workspaces*, como por exemplo o `ros2_ws`, execute:

```
$ source ~/ros_ws/install/setup.bash
```

3.2 Workspaces

Um *workspace* é o diretório principal onde o projeto está sendo desenvolvido e onde se encontram todos os pacotes (*packages*) que compõem o sistema. Além do

workspace principal, existe também o chamado *core workspace*, correspondente ao diretório onde o ROS propriamente dito está instalado.

3.2.1 Criando um ROS Workspace

No Linux você precisa criar uma pasta com o nome do *workpace* desejado com uma pasta `src` dentro. Utilize o comandos:

```
#Para criar a pasta na home do usuario
$ mkdir -p ~/ros2_ws/src

#Clone pacotes de exemplo do ROS2 Humble
$ cd ~/ros2_ws/src && git clone https://github.com/ros/
   ros_tutorials.git -b humble

#Resolva possíveis dependencias
$ rosdep install -i --from-path src --rosdistro humble -y

#Na raíz do workspace, contrua-o
$ cd ~/ros2_ws && colcon build
```

4 ID de Domínio

Por padrão, o ROS se comunica com qualquer máquina que esteja na mesma rede e utilize o mesmo ID de domínio.

Esse ID é definido pela variável de ambiente (ROS_DOMAIN_ID). O valor padrão dessa variável é zero.

No entanto, caso não se deseje que as máquinas executando ROS na mesma rede interfiram entre si, é necessário atribuir um ID de domínio diferente para cada uma delas, configurando a variável (ROS_DOMAIN_ID) individualmente.

Dessa forma, ao definir valores distintos para (ROS_DOMAIN_ID) em cada máquina, apenas os nós que compartilham o mesmo valor poderão se comunicar entre si.

4.1 Escolhendo o ID de domínio

A descoberta do IP das máquinas que executam nós do *ROS* na mesma rede é realizada por meio de comunicação UDP. O ID de domínio escolhido é utilizado para calcular as portas UDP utilizadas pelo sistema.

É importante notar que alguns sistemas operacionais podem utilizar portas UDP que podem conflitar com as portas usadas pelo *ROS*. Por esse motivo, recomenda-se escolher um valor para o ID de domínio (ROS_DOMAIN_ID) dentro da faixa de 0 a 101, de modo a evitar possíveis conflitos.

4.2 Definindo o ID de Domínio

Para definir o ID de domínio, atribua o valor desejado à variável de ambiente (ROS_DOMAIN_ID) com o comando:

```
$ export ROS_DOMAIN_ID=<id_desejado>
```

Para que essa configuração seja aplicada automaticamente em todos os terminais abertos no futuro, insira o comando abaixo ao final do arquivo `bashrc`:

```
$ echo "export ROS_DOMAIN_ID=<id_desejado>" >> ~/.bashrc
```

4.3 Comunicação local

Para evitar que uma máquina interfira no funcionamento do ROS em outras máquinas na mesma rede, é possível definir a variável `(ROS_LOCALHOST_ONLY)`. Isso garante que as mensagens do ROS não sejam enviadas para a rede, permanecendo restritas à máquina local.

Para ativar essa funcionalidade no terminal atual, utilize o comando:

```
$ export ROS_LOCALHOST_ONLY=1
```

Para que a configuração seja carregada automaticamente em todas as novas sessões do terminal, adicione o comando ao final do arquivo `.bashrc`:

```
$ echo "export ROS_LOCALHOST_ONLY=1" >> ~/.bashrc
```

5 TurtleSim, o "simulador de tartaruga"

Uma primeira interação com as potencialidades do ROS podem ocorrer através do nodo turtlesim. Execute:

```
$ ros2 run turtlesim turtlesim_node
```

Agora, em um novo terminal, execute:

```
$ ros2 run turtlesim turtle_teleop_key
```

Nele, o usuário é capaz de controlar a tartaruga do simulador através do teclado. Recomenda-se explorar a lista de nodos e tópicos ativos e compreender as relação entre eles. É possível visualizar o grafo de comunições através do comando:

```
$ rqt_graph
```

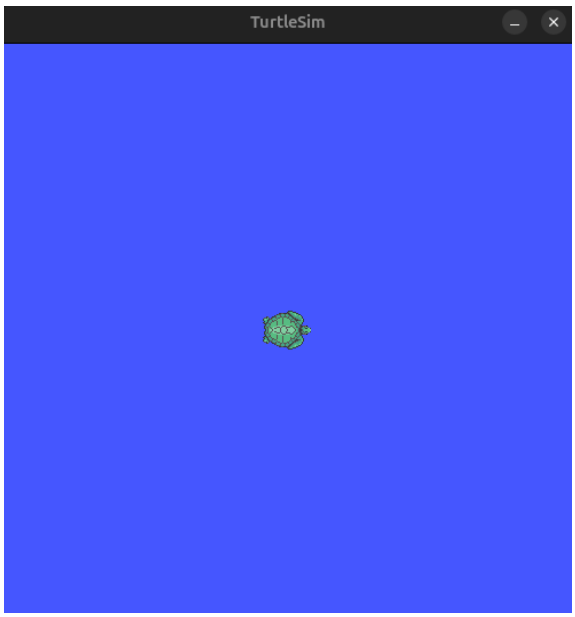

Figura 4 – Simulador TurtleSim

6 Comandos úteis

Com o intuito de guiar novos usuários, segue uma lista de comandos úteis a serem executados em shell Linux (Ubuntu 22.04), compatíveis com o ROS2 Humble. A escolha dessa distribuição ROS se dá por sua utilização nas aulas de "Programação de Robôs Móveis" (SSC0712), ministradas durante o primeiro semestre letivo do ano de 2025 e sua proximidade com distribuições ROS1.

```
#Instala a distribuição ROS2 Humble
$ sudo apt update
$ sudo apt install ros-humble-desktop
$ sudo apt install '~nros-humble-rqt*'
```

```
#Define o ambiente de operação
$ source /opt/ros/humble/setup.bash
```

```
#Executa o ''simulador de tartaruga''
$ ros2 run turtlesim turtlesim_node
```

```
#Lista os nodos ativos
$ ros2 node list
```

```
#Apresenta informações adicionais sobre um nodo
$ ros2 node info <nome_do_nodo>
```

```
#Exibe a lista de tópicos ativos e seus tipos
$ ros2 topic list -t
```

```
#Exive o tipo de mensagem sendo publicada em um tópico
$ ros2 topic echo <nome_do_topico>
```

```
#Exibe a lista de serviços ativos e seus tipos
$ ros2 service list
```

```
#Exibe a lista de actions ativos e seus tipos
$ ros2 action list
```

```
#Exibe o grafo de comunicações
$ rqt_graph
```

Para outros comandos e instruções, recomenda-se a verificação da documentação, disponível em:

```
https://docs.ros.org/en/humble/
```

7 Exemplo de aplicação

Através dos conhecimentos adquiridos pela disciplina "Programação de Robôs Móveis" (SSC0712), ministrada pelo Professor Doutor Matheus Machado dos Santos, os autores desses texto adaptaram um *workspace* disponibilizado pelo professor e trabalharam sobre ele para que tivessem em mãos um algoritmo de navegação e de controle de um robô. O objetivo era que esse concluísse uma tarefa conhecida popularmente como "Catch the flag!", no qual o indivíduo (no caso, um robô) deve partir de sua base, navegar em uma arena a partir das informações de seus sensores (no caso, um LiDAR), encontrar a bandeira, capturá-la, retornar à base e depositar a bandeira.

O projeto baseou-se no framework ROS2 Humble, integrado a um software de simulação 3D chamado Gazebo. Os arquivos desenvolvidos e instruções para executar o projeto estão disponíveis no repositório GitHub:

```
https://github.com/Lipe-Gomes/trabalho1_prm
```

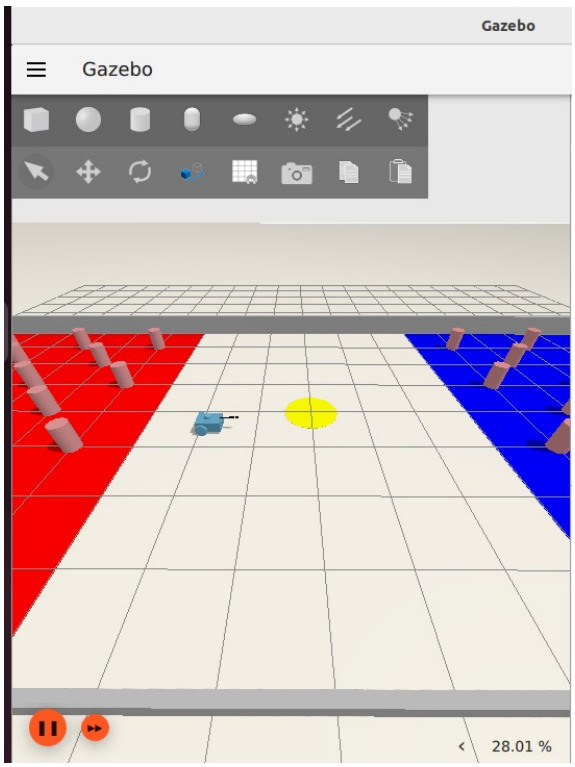

Figura 5 – Frame do simulador Gazebo para o exemplo de aplicação

Referências

OPEN ROBOTICS. **ROS 2 Documentation: Humble Hawksbill**. 2022. Disponível em: https://docs.ros.org/en/humble/. Acesso em: 29 jun. 2025.

OPEN ROBOTICS. **Why ROS?**. [S. l.], [s.d.]. Disponível em: https://www.ros.org/blog/why-ros/. Acesso em: 29 jun. 2025.

GOMES, F. de O. **trabalho1_prm**. 2025. Repositório GitHub. Disponível em: https://github.com/Lipe-Gomes/trabalho1_prm. Acesso em: 30 jun. 2025.