Introdução ao ROS 2

Alan José

THUNDERATZ

2025-11-26

Tópicos

Introdução

Motivação

- Do que é composto um robô ou um projeto de robótica?
  - Sensores: Câmeras, encoders, botões, giroscópios, ...
  - Atuadores: Motores, displays, LEDs, buzzers, ...
  - Sistemas de controle: Hardware + Software
  - Simulação

- Como podemos fazer tudo isso?

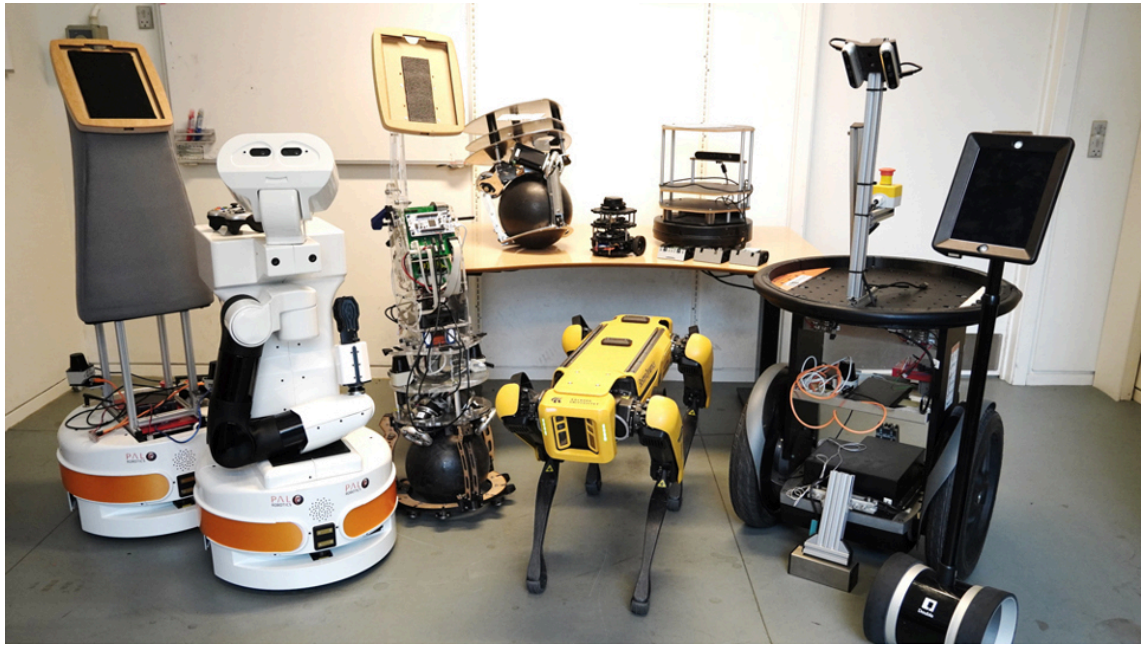
Motivação

- Do que é composto um robô ou um projeto de robótica?
  - Sensores: Câmeras, encoders, botões, giroscópios, ...
  - Atuadores: Motores, displays, LEDs, buzzers, ...
  - Sistemas de controle: Hardware + Software
  - Simulação

- Como podemos fazer tudo isso?
  - De muitas formas!

Motivação – O Problema

- Os projetos tendem a ficar complexos.
- Construir um robô já é difícil e fica ainda mais difícil quando temos que reinventar a roda.



Solução – ROS

- O ROS é um Robot Operating System de código aberto.
- Um conjunto de bibliotecas de software e ferramentas que ajudam a criar aplicações de robótica em todo tipo de plataforma.

O ROS tem dois “lados”

Solução – ROS

- O ROS é um Robot Operating System de código aberto.
- Um conjunto de bibliotecas de software e ferramentas que ajudam a criar aplicações de robótica em todo tipo de plataforma.

O ROS tem dois “lados”

- Provê serviços e ferramentas padronizadas
  - Abstração de hardware
  - Controle de baixo nível
  - Comunicação entre processos
  - Gerenciamento de pacotes
  - Biblioteca de funções comuns

Solução – ROS

- O ROS é um Robot Operating System de código aberto.
- Um conjunto de bibliotecas de software e ferramentas que ajudam a criar aplicações de robótica em todo tipo de plataforma.

O ROS tem dois “lados”

- Provê serviços e ferramentas padronizadas
  - Abstração de hardware
  - Controle de baixo nível
  - Comunicação entre processos
  - Gerenciamento de pacotes
  - Biblioteca de funções comuns
- Ecossistema de pacotes da comunidade de código aberto
  - SLAM
  - Visão
  - Controle de motores
  - IA
  - etc...

Filosofia ROS

- Peer to Peer
  - Uma aplicação ROS é composta de pequenos programas que funcionam em paralelo, trocando mensagens entre si.
- Baseado em Ferramentas
  - É incentivada a criação de programas pequenos e genéricos que realizam tarefas básicas, como visualização, plotagem, controle remoto, ...
- Multi-Linguagem
  - Pacotes ROS não estão limitados a uma única linguagem. É possível usar C++, Python, LISP, Java, JavaScript, MATLAB, Ruby, ...
- Código Aberto
  - A alta modularidade do ROS incentiva a criar programas e bibliotecas pequenas que utilizam outros pacotes ROS para implementar funcionalidades comuns.

Conceitos

Conceitos

- Nós (Nodes): Programas executáveis de propósito único.
- Tópicos (Topics): Mecanismo de comunicação assíncrono.
- Mensagens (Messages): Estruturas de dados para comunicação entre nós.
- Parâmetros (Parameters): Dados de configuração.
- Pacotes (Packages): Coleção de códigos que têm o mesmo propósito.
- Services
- Actions

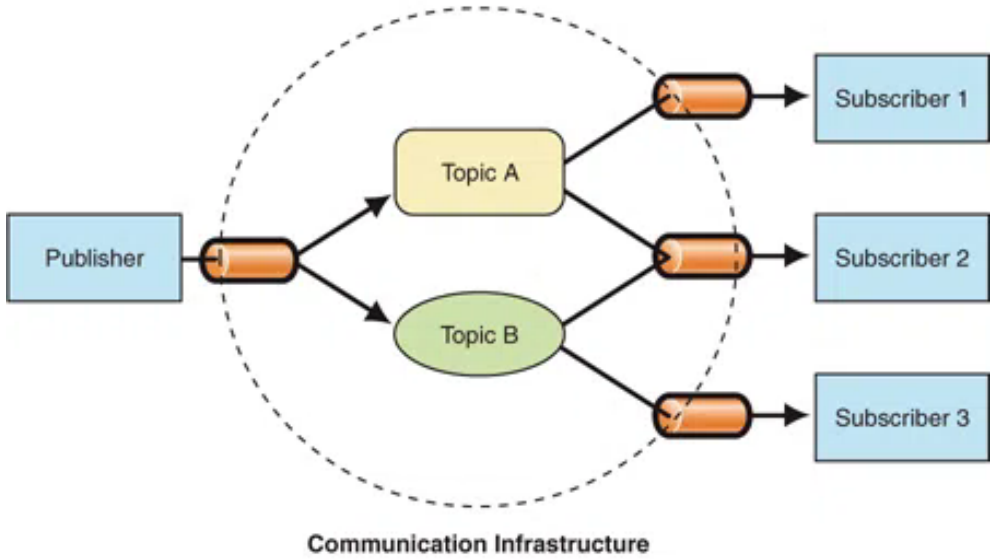
Conceitos

- Nós (Nodes): Programas executáveis de propósito único.
- Tópicos (Topics): Mecanismo de comunicação assíncrono.
- Mensagens (Messages): Estruturas de dados para comunicação entre nós.
- Parâmetros (Parameters): Dados de configuração.
- Pacotes (Packages): Coleção de códigos que têm o mesmo propósito.
- Services
- Actions

} Não será abordado hoje

Modelo Publisher/Subscriber

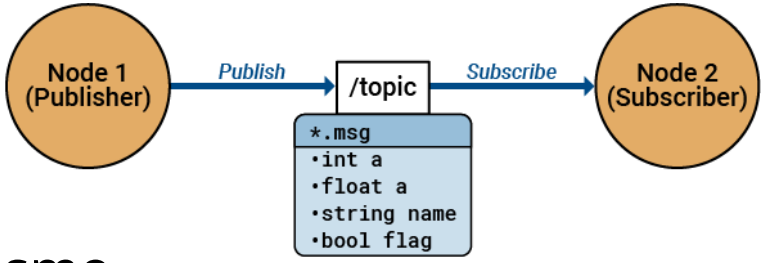
- É um modelo genérico, não exclusivo ao ROS
- Existem dois tipos de entidades: *Publishers* e *Subscribers*
  - Publishers: Publicam informações em um canal
  - Subscribers: Se inscrevem num canal para receber informações



THUNDERATZ

Modelo Publisher/Subscriber

Aplicado a ROS



- Qualquer node pode publicar/inscrever em qualquer tópico
- Múltiplos nodes podem publicar/inscrever no mesmo tópico
- Nodes podem publicar/inscrever em múltiplos tópicos

Mensagens

- Tópicos só podem transmitir um único tipo de mensagem
- Mensagens são compostas por tipos primitivos (`string`, `float`, ...) e tipos compostos (Ex: `Point`)

```
# Point.msg                    msg
float64 x
float64 y
float64 z
```

THUNDERATZ

Nós (Nodes)

- Programas executáveis de propósito único.
  - Driver de sensor, driver de atuador, mapeamento, interface, ...
- Compilado, executado e gerenciado de forma individual.
- Cada nó é um processo separado, ou seja:
  - Não compartilham variáveis.
  - Executam em paralelo ~~(ou quase[1])~~
- Os nós podem publicar ou se inscrever em tópicos.
- São escritos usando bibliotecas como:
  - `rclcpp` (C++)
  - `rclpy` (Python)

[1] O processador funciona tão rápido que parece paralelo.

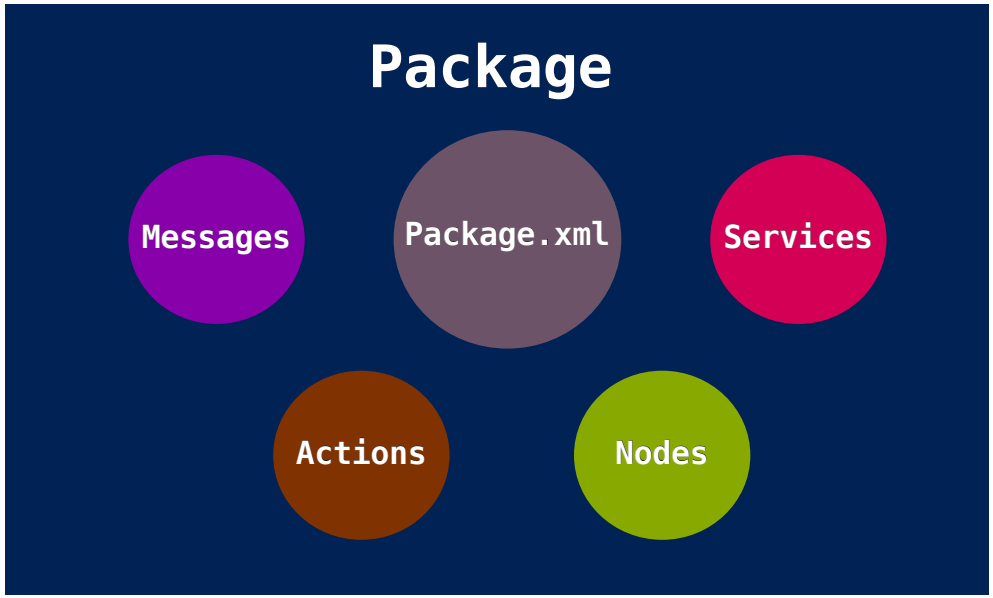
Tópicos

- Implementam um mecanismo de comunicação Publish/Subscribe.
- Permitem a transmissão 1-para-N.

Cada tópico só pode transmitir um único tipo de mensagem.

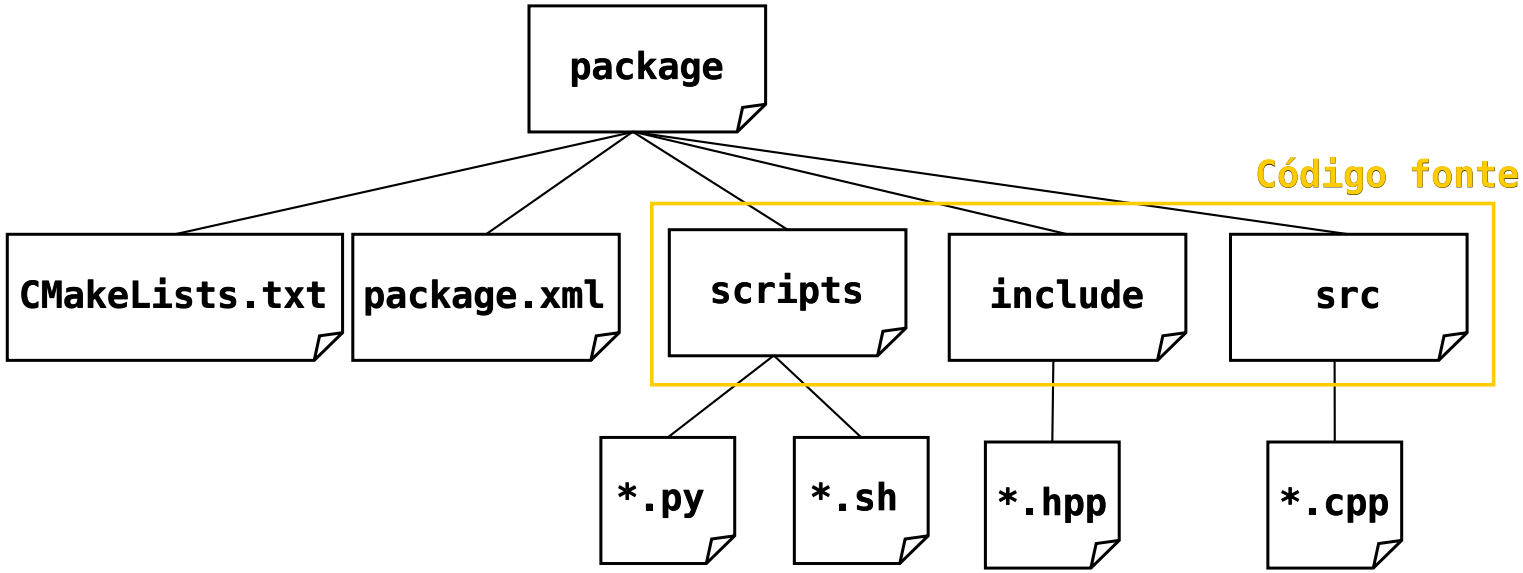
Workspace e Pacotes

- As aplicações ROS são organizadas em pacotes.
- Um pacote é um conjunto de nós, mensagens e serviços.
- Os pacotes são identificados pelo arquivo `package.xml`, que contém informações de identificação, como nome do pacote, autor, dependências, ...



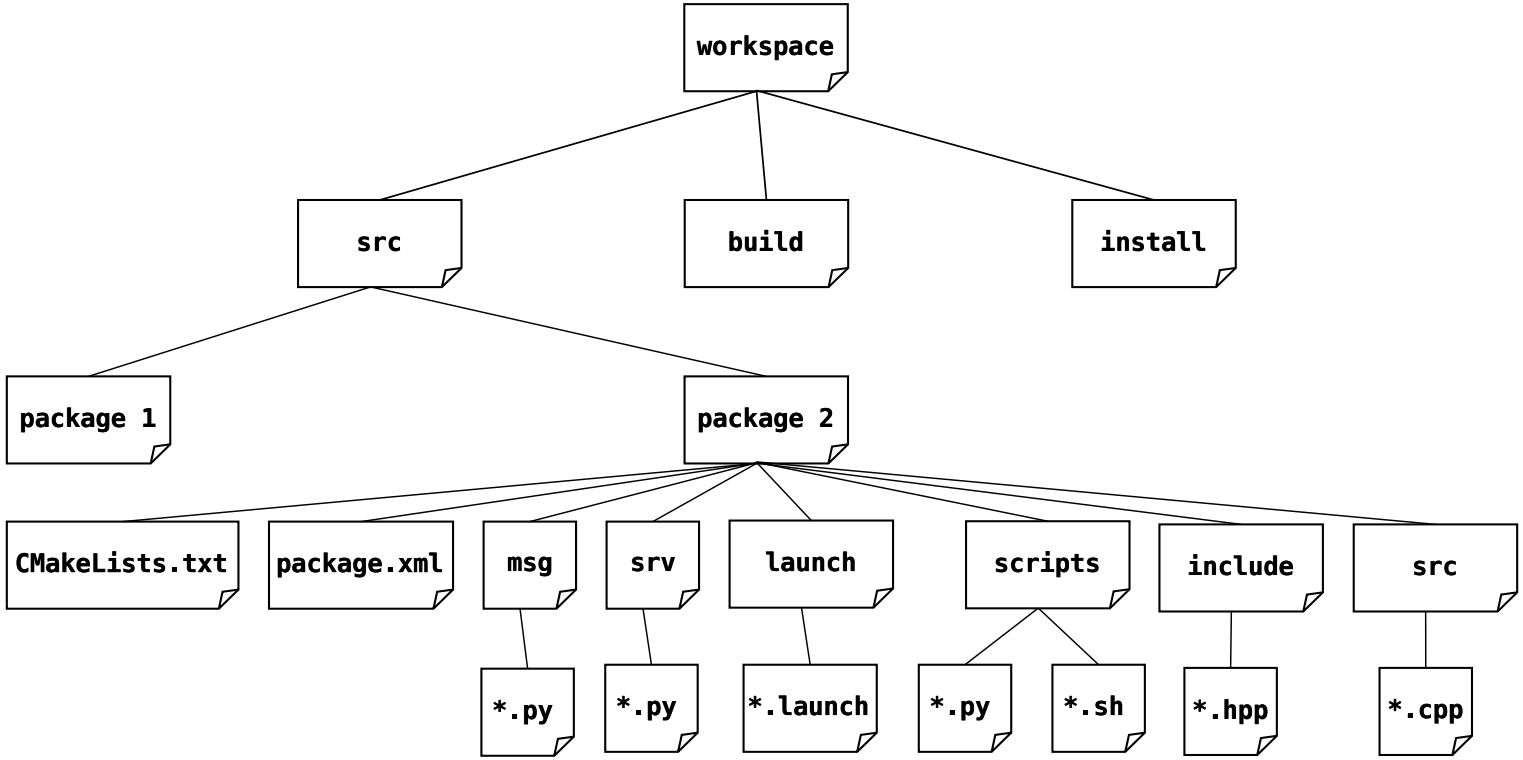
Workspace e Pacotes

- As aplicações ROS são organizadas em pacotes.
- Um pacote é um conjunto de nós, mensagens e serviços.
- Os pacotes são identificados pelo arquivo `package.xml`, que contém informações de identificação, como nome do pacote, autor, dependências, ...



Workspace e Pacotes

- Uma workspace é o ambiente que agrupa e compila um conjunto de pacotes.



Mão na massa

Instalação

- Existem várias distribuições do ROS lançadas anualmente.
- Todas elas suportam oficialmente uma versão específica do Ubuntu.
- Vamos utilizar o ROS 2 Humble.

Acesse o link abaixo para seguir o tutorial de instalação.

https://docs.ros.org/en/humble/Installation/Ubuntu-Install-Debs.html

Ambiente ROS

- Para facilitar a instalação de múltiplas versões do ROS ao mesmo tempo, é necessário ativar a versão desejada em todo novo terminal.
- Isso é feito carregando o arquivo `setup.bash`, localizado na pasta `/opt/ros/<distro>`

.

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

- Caso você só trabalhe com uma única versão do ROS (nosso caso), é vantajoso adicionar o comando acima no arquivo de inicialização do bash, `~/bashrc`.

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

Criando a Workspace

- Vamos criar a pasta da workspace (e a `src`, onde ficarão os pacotes).

```bash
mkdir -p ~/ros2_ws/src
cd ~/ros2_ws/src # Entra na pasta
```

- Antes de criarmos o nosso pacote, vamos clonar alguns pacotes de exemplo.

```bash
git clone https://github.com/ros/ros_tutorials.git -b humble
```

- Vamos voltar para a pasta da workspace.

```bash
cd ..
```

Dependências

- Os pacotes que acabamos de clonar dependem de programas e bibliotecas externas.
- A maneira ROS de gerenciar dependências é com o `rosdep`.
- Ele lê todos os arquivos `package.xml` e instala as dependências.

Ao lado, estão algumas das dependências do `turtlesim`.

```xml
<!--
  ros_tutorials/turtlesim/package.xml
-->
<build_depend>qt5-qmake</build_depend>
<build_depend>qtbase5-dev</build_depend>
<exec_depend>libqt5-core</exec_depend>
<exec_depend>libqt5-gui</exec_depend>
<depend>ament_index_cpp</depend>
<depend>geometry_msgs</depend>
<depend>rclcpp</depend>
<depend>rclcpp_action</depend>
<depend>std_msgs</depend>
```

Dependências

- Os pacotes que acabamos de clonar dependem de programas e bibliotecas externas.
- A maneira ROS de gerenciar dependências é com o `rosdep`.
- Ele lê todos os arquivos `package.xml` e instala as dependências.

```xml
<!--
  ros_tutorials/turtlesim/package.xml
-->
<build_depend>qt5-qmake</build_depend>
<build_depend>qtbase5-dev</build_depend>
<exec_depend>libqt5-core</exec_depend>
<!-- ... -->
```

```bash
rosdep install -i --from-path src --rosdistro humble -y
```

Compilando a workspace

- As workspaces ROS são compiladas com o `colcon`.

```bash
colcon build --packages-select turtlesim
```

- O que esse comando fez?
  - Leu o pacote `turtlesim` em `src` e o compilou.
  - Colocou todos os executáveis (nós) na pasta `build/`.
  - Colocou todos os arquivos não executáveis auxiliares (`*.launch`, arquivos de configuração, ...) na pasta `install/`.

```
Starting >>> turtlesim
Finished <<< turtlesim [5.49s]

Summary: 1 package finished [5.58s]
```

```
~/ros2_ws
❯ ls
build/  install/  log/  src/
```

Testando

- Sempre que criamos ou recompilamos uma workspace, é necessário ativá-la novamente.

```bash
source install/setup.bash
```

- Para testar se tudo funcionou, vamos executar o nó `turtlesim_node` do pacote `turtlesim` que acabamos de compilar.

```bash
ros2 run turtlesim turtlesim_node
```

Testando

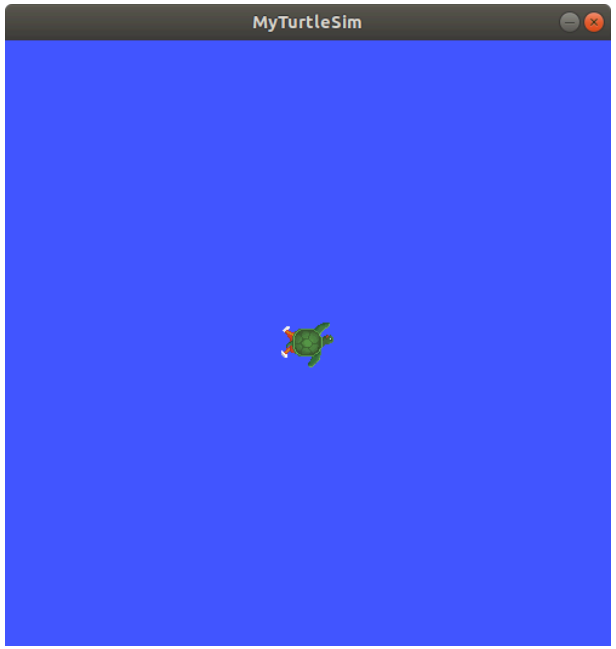
- Sempre que criamos ou recompilamos uma workspace, é necessário ativá-la novamente.

```bash
source install/setup.bash
```

- Para testar se tudo funcionou, vamos executar o nó `turtlesim_node` do pacote `turtlesim` que acabamos de compilar.

```bash
ros2 run turtlesim turtlesim_node
```

Isso deve abrir uma janela como esta.



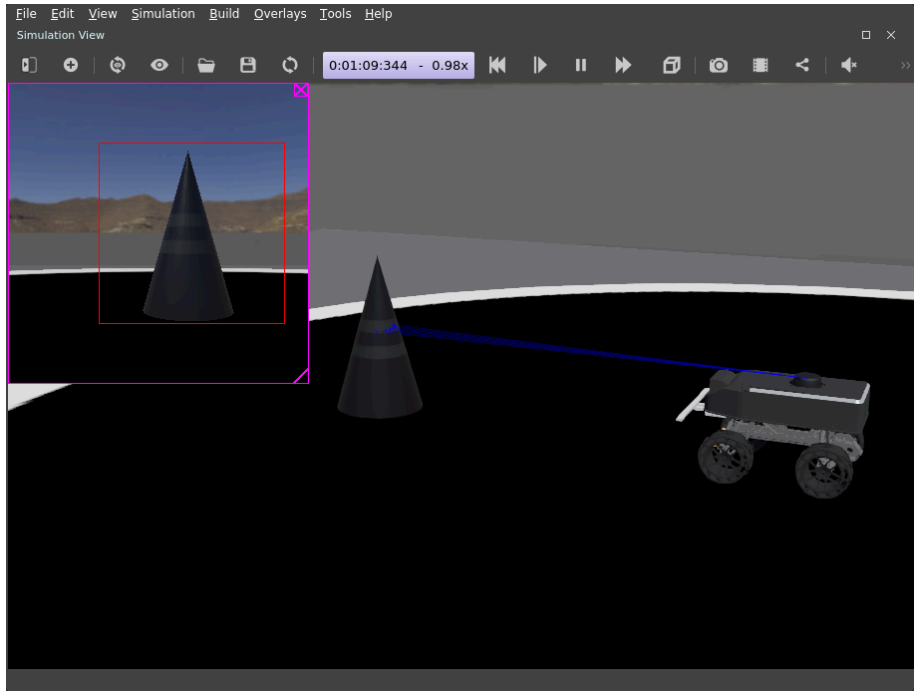
Exercício

Introdução

Vamos criar um código para ajudar o Perse a derrotar o seu maior inimigo, o Cone, em uma partida de sumô.

Para isso, ele estará equipado com alguns sensores, como:

- LiDAR
- Encoder
- IMU (Giroscópio)
- Sensor de linha



Objetivo

Empurrar o cone para fora do dohyo.[1]

[1] ~~borda do dohyo~~

Obtendo o código

- Clone o repositório da aula dentro da pasta `src`.[1]

```bash
# dentro da pasta ~/perse_ws
cd src
git clone https://github.com/ThundeRatz/aula-ros-2025
```

- Baixe as dependências.

```bash
cd ..
rosdep install -i --from-path src --rosdistro humble -y
```

- Compile o código.

```bash
colcon build
```

[1] Vamos considerar a workspace `perse_ws` para este exercício.

Verificando a Instalação

- Faça o source da workspace.

```bash
source install/setup.bash
```

- Verifique os pacotes.

```
~/perse_ws
❯ ros2 pkg list|grep perse
perse_brain
perse_sumo
```

Se esta for a saída, tudo deu certo. 😀👍

Testando a simulação

Abra um novo terminal e execute o comando abaixo.

```bash
ros2 launch perse_sumo sim_launch.py
```

Verifique os tópicos disponíveis.

```
~/perse_ws
❯ ros2 topic list
/parameter_events
/perse/actuator/cmd_vel
... Outros tópicos
/perse/sensors/encoder_fusion/distance
/perse/sensors/line_sensor
/remove_urdf_robot
/rosout
```

Testando a simulação

Vamos checar a posição dos cones identificados pela câmera.

```
~/perse_ws
❯ ros2 topic echo /perse/camera/recognitions
... Tem mais coisa aqui
      - 0.0
      - 0.0
      - 0.0
  bbox:
    center:
      position:
        x: 207.0
        y: 150.0
      theta: 0.0
    size_x: 185.0
    size_y: 211.0
  id: ''
---
```

Tente mover um cone no campo de visão do Perse usando a interface do Webots.

Testando a simulação

Agora, vamos controlar o Perse utilizando o teclado.

Para comandar os motores, é preciso publicar as velocidades lineares e angulares no tópico `/perse/actuator/cmd_vel`.

```
~/perse_ws
❯ ros2 topic info /perse/actuator/cmd_vel
Type: geometry_msgs/msg/Twist
Publisher count: 0
Subscription count: 1
```

Testando a simulação

Esse tópico é do tipo `geometry_msgs/Twist`.

geometry_msgs/Twist Message

File: `geometry_msgs/Twist.msg`

Raw Message Definition

```
# This expresses velocity in free space broken into its linear and angular parts.
Vector3 linear
Vector3 angular
```

Compact Message Definition

```
geometry_msgs/Vector3 linear
geometry_msgs/Vector3 angular
```

autogenerated on Mon, 28 Apr 2025 02:25:36

Figure 1: geometry_msgs/Twist Documentation

Testando a simulação

É possível fazer o Perse andar publicando uma velocidade linear de $0.2\ \mathrm{m/s}$.

```bash
ros2 topic pub /perse/actuator/cmd_vel geometry_msgs/Twist '{angular: {x: 0.0, y: 0.0, z: 0.0}, linear: {x: 0.2, y: 0.0, z: 0.0}}'
```

...Outra forma é utilizar o nó `teleop_twist_keyboard` para publicar essa mensagem utilizando o teclado.

Execute os comandos abaixo para baixar o nó utilizando o *apt* e executá-lo.

```bash
sudo apt install ros-humble-teleop-twist-keyboard
```

```bash
ros2 run teleop_twist_keyboard teleop_twist_keyboard --ros-args --remap cmd_vel:=/perse/actuator/cmd_vel
```

Criando o cérebro do Perse

Vamos modificar o nó `perse_brain` para implementar a lógica de empurrar o cone.

- Abra a pasta `~/perse_ws/src/perse_brain` no *VSCode*.

A pasta `perse_brain` tem a seguinte estrutura de arquivos:

```
src/perse_brain
├── CMakeLists.txt      Configuração do CMake
├── package.xml         Informações do pacote
├── config
│   └── brain_parameters.yaml    Parâmetros
├── include
│   └── perse_brain
│       └── brain.hpp       Código do nó
├── launch
│   └── brain.launch        Launchfile
└── src
    └── brain.cpp       Código do nó
```

Abra o arquivo `brain.cpp`

Criando o cérebro do Perse

Controle dos motores

Inicialmente, vamos fazer o Perse andar em linha reta

- Para isso, crie um publisher para o tópico `/perse/actuator/cmd_vel`

```cpp
auto cmd_vel_pub = ros_node->create_publisher<geometry_msgs::msg::Twist>(
  "/perse/actuator/cmd_vel", 1
);
```

- Crie uma mensagem e a publique usando o publisher.

```cpp
geometry_msgs::msg::Twist twist_msg;
twist_msg.linear.x = 0.3;
cmd_vel_pub->publish(twist_msg);
```

Após as mudanças, compile o código com `colcon build`.[1]

[1] Lembre-se de também rodar `source install/setup.bash`.

Criando o cérebro do Perse

Launchfile

Para executar o código, existem duas maneiras:

- Rodar um nó explicitamente:

```bash
# ros2 run <pacote> <executável>
ros2 run perse_brain brain
```

- Rodar com um launch file:

```bash
# ros2 launch <pacote> <nome do launchfile>
ros2 launch perse_brain brain.launch
```

Criando o cérebro do Perse

Launchfile

É preferível utilizar launch files, porque eles permitem rodar vários nós de uma única vez e é muito mais fácil definir os parâmetros.

Modifique o arquivo `src/perse_brain/launch/brain.launch`.[1][2]

```xml
<launch>
  <arg name="brain_parameters" default="$(find-pkg-share perse_brain)/config/
  brain_parameters.yaml"/>

  <node pkg="perse_brain" exec="brain" name="perse_brain" output="screen">
    <param from="$(var brain_parameters)"/>
  </node>
</launch>
```

[1] Note que já incluímos o arquivo de parâmetros que usaremos mais tarde.
[2] É preciso compilar o código novamente.

Criando o cérebro do Perse

Sensor de linha

Para evitar que o Perse caia do dohyo, crie um subscriber para o tópico `/perse/sensors/line_sensor`.

```cpp
auto line_sensor_sub = ros_node->create_subscription<sensor_msgs::msg::Range>(
          "/perse/sensors/line_sensor", 1, &line_sensor_callback
);
```

Subscribers precisam de uma função de callback para processar as mensagens.

```cpp
void line_sensor_callback(sensor_msgs::msg::Range msg) {
  line_sensor = msg.range;
}
```

Criando o cérebro do Perse

Agora, basta parar o Perse quando ele estiver em cima da linha.

```cpp
twist_msg.linear.x = 0.3;

  double line_sensor_threshold = 0.2;
  if (line_sensor < line_sensor_threshold) {
    twist_msg.linear.x = 0;
  }

  cmd_vel_pub->publish(twist_msg);
```

Criando o cérebro do Perse

LiDAR

Vamos usar o LiDAR para fazer o Perse andar na direção do cone.

```cpp
auto lidar_sub = ros_node->create_subscription<sensor_msgs::msg::LaserScan>(
          "/perse/sensors/lidar", 1, &lidar_callback);

// Já implementado no repositório da aula
void lidar_callback(sensor_msgs::msg::LaserScan msg) {
  // ...
  lidar_angle = //...
}
```

Criando o cérebro do Perse

LiDAR

Para a lógica principal, vamos fazer o ângulo das rodas[1] ser a diferença entre o ângulo que queremos e o ângulo do Perse[2].

```cpp
while (rclcpp::ok()) {
  double kp = 1.0;
  double angle_err = kp * (0 - lidar_angle);

  twist_msg.angular.z = -angle_err;
  // ...
  cmd_vel_pub->publish(twist_msg);
}
```

[1] Embora esteja escrito `angular.z`, isso é, na verdade, o ângulo das rodas, limitado de $-\frac{\pi}{4}$ a $\frac{\pi}{4}$.

[2] Isso é um controle proporcional.

Criando o cérebro do Perse

Parâmetros

Existem alguns valores que precisam ser calibrados, e é um pouco complicado ficar compilando o código o tempo inteiro para ver as mudanças.

Para resolver isso, vamos criar 3 parâmetros:

- `line_sensor_threshold`: threshold do sensor de linha para considerá-lo ativo.
- `kp`: multiplicador do erro angular.
- `max_velocity`: velocidade máxima.

Parâmetros podem ser passados para os nós através da linha de comando ou de arquivos de configuração.

Criando o cérebro do Perse

Vamos modificar o arquivo `perse_brain/config/brain_parameters.yaml`.

```yaml
perse_brain:
  ros__parameters:
    kp: 1.0
    line_sensor_threshold: 0.2
    max_velocity: 0.3
```

Agora, vamos obter os parâmetros no código principal.

```cpp
// É preciso declarar o parâmetro primeiro.
ros_node->declare_parameter(name: "kp", type: rclcpp::PARAMETER_DOUBLE);
// Agora é só guardá-lo em uma variável.
double kp = ros_node->get_parameter(name: "kp").as_double();
```

Recapitulando

Recapitulando

Nesta aula, nós vimos:

- Como criar uma workspace ROS.
- Como criar e baixar pacotes, e instalar suas dependências.
- Como gerenciar a workspace com os principais comandos do `ros2-cli`.
- Como criar e executar launch files.
- Como inspecionar tópicos.
- Como se inscrever e publicar em tópicos.
- Como usar parâmetros.
- Como derrotar o Cone!

Fim

Próximos passos

- Você consegue fazer o Perse realmente empurrar o cone para fora?
- Será que ele consegue ir mais rápido?
- Como você poderia usar a câmera?
- Como seria a implementação de um fail-safe?

Dúvidas?