Introdução ao ROS e ao simulador Gazebo

Rafael Gomes Braga

Bibliografia

- Livro “A gentle introduction to ROS”, que pode ser baixado gratuitamente no link: https://cse.sc.edu/~jokane/agitr/
- Documentação oficial do ROS: https://wiki.ros.org
- Documentação oficial do Gazebo: http://gazebosim.org/
- Tutorial “Robotic Simulation with ROS and Gazebo”: http://www.generationrobots.com/blog/en/2015/02/robotic-simulation-scenarios-with-gazebo-and-ros/

Aula 1: Introdução ao ROS

ROS - Robot Operating System

ROS é um framework (conjunto de programas e ferramentas) de código aberto desenvolvido para servir como base em aplicações de robótica. Ele fornece diversos serviços como abstração de hardware, implementação de funções comumente utilizadas, um sistema de comunicação entre processos, gerenciamento de pacotes, entre outros. Também fornece bibliotecas e ferramentas para criar código que seja capaz de ser executado através de várias máquinas simultaneamente.

Vantagens do ROS

- Computação distribuída: O ROS permite a criação com facilidade de aplicações que são executadas em várias máquinas simultaneamente.
- Reutilização de software: Muitas estruturas e algoritmos padrão estão disponíveis no ROS.
- Teste rápido: O ROS tem ferramentas que facilitam e agilizam o processo de teste do software desenvolvido.

Pode ser programado em C++, Python, Java, entre outras.

Intalação do ROS

- As instruções encontram-se no site: http://wiki.ros.org/indigo/Installation/Ubuntu
- Instalar a versão **`ros-kinetic-desktop-full`**
- No nosso curso utilizaremos alguns pacotes adicionais:

```
Sudo apt-get update

sudo apt-get install ros-kinetic-ros-control
ros-kinetic-gazebo-ros-pkgs
```

Exemplo

Usaremos um exemplo para estudar os conceitos básicos do ROS. Executar em três terminais diferentes:

```
roscore

rosrun turtlesim turtlesim_node

rosrun turtlesim turtle_teleop_key
```


TurtleSim


TurtleSim

Exemplo

- Esses comandos executarão o turtlesim, que é um simulador simples instalado junto com o ROS.
- Mantendo o terceiro terminal ativo, é possível controlar a tartaruga usando as setas do teclado.
- Esse exemplo apresenta dois nós se comunicando através da publicação de mensagens. Estudaremos esses conceitos a seguir.

Pacotes

- Todo o software no ROS é organizado em pacotes.
- Um pacote no ROS é uma coleção coerente de arquivos, geralmente incluindo tanto executáveis quanto arquivos de suporte.

Pacotes

- Listar todos os pacotes instalados:

```
rospack list
```

- Descobrir em qual pasta está instalado um pacote:

```
rospack find nome-do-pacote
```

- Exemplo:

```
rospack find turtlesim
```

Pacotes

- Todo pacote é definido por um manifesto, um arquivo chamado `package.xml`. Esse arquivo define alguns detalhes do pacote incluindo seu nome, versão, mantenedor e dependências.
- Inspecionar a pasta de um pacote:

```
rosls nome-do-pacote
```

- Ir para a pasta do pacote:

```
roscd nome-do-pacote
```

Pacotes

Exemplo: Ver as imagens das tartarugas do turtlesim:

```
rosls turtlesim

rosls turtlesim/images

roscd turtlesim/images

eog box-turtle.png
```

ROS Master

Um dos objetivos do ROS é permitir que os roboticistas projetem software como um grupo de pequenos programas independentes uns dos outros, chamados nós, que são executados ao mesmo tempo. Para isso, os nós precisam ser capazes de se comunicar uns com os outros. O ROS Master é o programa que permite e gerencia essa comunicação.

ROS Master

- Para iniciar o master:

  ```
  roscore
  ```

- O comando roscore deve ser executado no início da execução de uma aplicação do ROS e deve continuar aberto durante todo o tempo da execução.

Nós

- Um nó é uma instância de um programa que está sendo executado.
- Para iniciar um nó:

  ```
  rosrun nome-do-pacote nome-do-executavel
  ```

- No exemplo do turtlesim, iniciamos dois nós: `turtlesim_node` e `turtle_teleop_key`

Nós

- Para listar todos os nós que estão sendo executados:

  ```
  rosnode list
  ```

- Obs: O nó `/rosout` é um nó especial que é inicializado automaticamente pelo roscore.

Nós

- Obter informações sobre um nó:

```
rosnode info nome-do-no
```

- Encerrar um nó:

```
rosnode kill nome-do-no
```

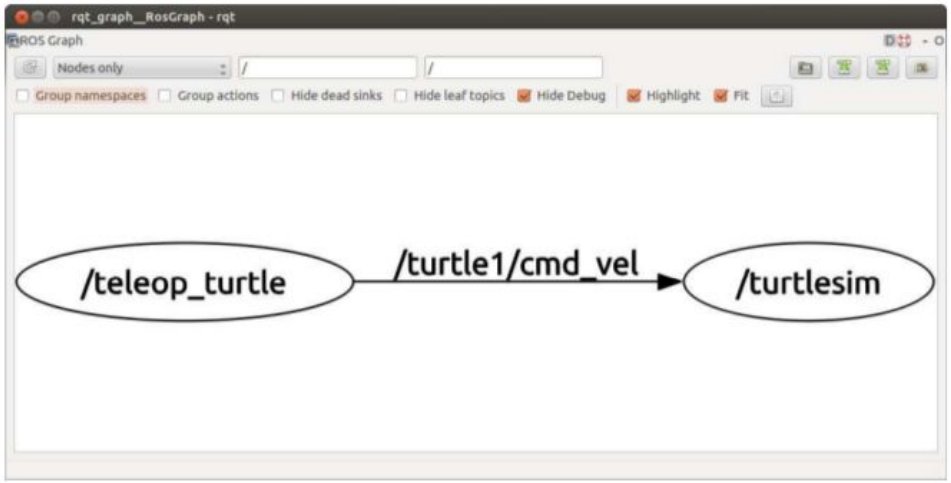
Tópicos e Mensagens

- No nosso exemplo, os nós `/turtlesim` e `/teleop_turtle` estão se comunicando de alguma forma.
- A forma mais básica que o ROS utiliza para fazer a comunicação entre os nós é enviando mensagens. As mensagens no ROS são organizadas em tópicos. A idéia é que os nós que querem compartilhar informação publicam mensagens no nó apropriado, enquanto que os nós que querem receber essa informação subscrevem naquele tópico. O ROS master garante que os nós publicadores e subscritores encontrem uns aos outros.

Tópicos e Mensagens

Visualizar uma representação gráfica dos nós e tópicos:

```
rqt_graph
```



Mensagens e Tipos de Mensagens

- Listar tópicos:

```
rostopic list
```

- Imprimir as mensagens de um tópico:

```
rostopic echo nome-do-topico
```

- Exemplo:

```
rostopic echo /turtle1/cmd_vel
```

Mensagens e Tipos de Mensagens

- Obter informações sobre um tópico:

```
rostopic info nome-do-topico
```

- Exemplo:

```
rostopic info /turtle1/color_sensor
```

Mensagens e Tipos de Mensagens

- Obter informações sobre um tipo de mensagem:

```
rosmsg show nome-do-tipo-de-mensagem
```

- Exemplos:

```
rosmsg show turtlesim/Color
```

```
rosmsg show geometry_msgs/Twist
```

- Para ver mais detalhes sobre a mensagem, usar a opção -r:

```
rosmsg show -r nome-do-tipo-de-mensagem
```

Mensagens e Tipos de Mensagens

- Publicar mensagens pela linha de comando:

```
rostopic pub nome-do-topico tipo-da-mensagem conteudo-da-mensagem
```

Mensagens e Tipos de Mensagens

- Exemplo:

```
rostopic pub /turtle1/cmd_vel geometry_msgs/Twist “linear:
    x: 2.0
    y: 0.0
    z: 0.0
angular:
    x: 0.0
    y: 0.0
    z: 0.0”
```

Um exemplo maior

```
rosrun turtlesim turtlesim_node __name:=A

rosrun turtlesim turtlesim_node __name:=B

rosrun turtlesim turtle_teleop_key __name:=C

rosrun turtlesim turtle_teleop_key __name:=D
```

- O que vai aparecer no rqt_graph?

Criar um Workspace

Antes de começarmos a criar nossos próprios pacotes é necessário criar um workspace, que é uma pasta onde todos os nossos pacotes ficarão.

```
mkdir -p ~/catkin_ws/src

cd ~/catkin_ws/src

catkin_init_workspace

cd ~/catkin_ws

catkin_make
```

Criar um Workspace

Para tornar os pacotes dentro do nosso workspace visíveis para o sistema do ROS, executar os comandos:

```
echo “source ~/catkin_ws/devel/setup.bash” > ~/.bashrc

source ~/.bashrc
```

Criando nosso primeiro pacote

- O comando para criar um pacote é:

```
catkin_create_pkg nome-do-pacote
```

- Criar um pacote para ser utilizado no curso:

```
cd ~/catkin_ws/src

catkin_create_pkg ros_e_gazebo
```

Criando nosso primeiro pacote

- `package.xml`: é o manifesto, que já foi explicado anteriormente
- `CMakeLists.txt`: é um script que será utilizado pelo catkin para construir os arquivos do projeto. Contém instruções como quais executáveis serão criados, quais arquivos fonte utilizar para criá-los e onde encontrar as bibliotecas que devem ser importadas.

Criando nosso primeiro pacote

No arquivo `package.xml`:

- A maioria dos campos é auto explicativa;
- `<build_depend> e <run_depend>` - Aqui são listadas as dependências do pacote. Editar o arquivo, adicionando roscpp, geometry_msgs e turtlesim como dependencias.

Criando nosso primeiro pacote

```
<build_depend>roscpp</build_depend>
<build_depend>geometry_msgs</build_depend>
<build_depend>turtlesim</build_depend>

<run_depend>roscpp</run_depend>
<run_depend>geometry_msgs</run_depend>
<run_depend>turtlesim</run_depend>
```

Criando nosso primeiro pacote

No arquivo CmakeLists.xml:

- `project(simuladores)` – nome do pacote
- `find_package(catkin REQUIRED)` – lista as dependencias do pacote. Editar essa linha deixando da seguinte forma:
  `find_package(catkin REQUIRED  COMPONENTS roscpp geometry_msgs turtlesim)`
- `catkin_package()` - Declara um pacote catkin

Criando nosso primeiro pacote

Compilar o pacote criado:

```
cd ~/catkin_ws

catkin_make
```

Criando nosso primeiro programa

hello.cpp

na pasta src

```
1  // This is a ROS version of the standard "hello, world"
2  // program.
3
4  // This header defines the standard ROS classes.
5  #include <ros/ros.h>
6
7  int main(int argc, char **argv) {
8    // Initialize the ROS system.
9    ros::init(argc, argv, "hello_ros");
10
11   // Establish this program as a ROS node.
12   ros::NodeHandle nh;
13
14   // Send some output as a log message.
15   ROS_INFO_STREAM("Hello, ROS!");
16 }
```

Criando nosso primeiro programa

- `#include <ros/ros.h>` - Inclui as classes padrão do ROS;
- `ros::init( argc, argv, “hello_ros” );` - Inicia o sistema do ROS, declarando um nó chamado “hello_ros”;
- `ros::NodeHandle nh;` - Cria um objeto NodeHandle para acessar as funções do ROS;
- `ROS_INFO_STREAM( “Hello ROS!”);` - Imprime a mensagem na tela;

Criando nosso primeiro programa

Editar o arquivo CMakeLists.txt:

Declarar executáveis:

```
add_executable(nome-do-executavel arquivos-fonte)
target_link_libraries(nome-do-executavel ${catkin_LIBRARIES})
```

No nosso caso:

```
add_executable(hello hello.cpp)
target_link_libraries(hello ${catkin_LIBRARIES})
```

Criando nosso primeiro programa

- Compilar o pacote

```
cd ~/catkin_ws
catkin_make
```

Criando nosso primeiro programa

Executar o programa

```
roscore

rosrun ros_e_gazebo hello
```

Aula 2: Publishers, Subscribers e Launch

Criando um Publisher

Agora nós vamos criar um programa que publica mensagens de comando de velocidade aleatórias para o turtlesim.

Criando um Publisher

pubvel.cpp

```
1  // This program publishes randomly-generated velocity
2  // messages for turtlesim.
3  #include <ros/ros.h>
4  #include <geometry_msgs/Twist.h>  // For geometry_msgs::Twist
5  #include <stdlib.h> // For rand() and RAND_MAX
6
7  int main(int argc, char **argv) {
8    // Initialize the ROS system and become a node.
9    ros::init(argc, argv, "publish_velocity");
10   ros::NodeHandle nh;
11
12   // Create a publisher object.
13   ros::Publisher pub = nh.advertise<geometry_msgs::Twist>(
14     "turtle1/cmd_vel", 1000);
15
16   // Seed the random number generator.
17   srand(time(0));
18
```

Criando um Publisher

```
19    // Loop at 2Hz until the node is shut down.
20    ros::Rate rate(2);
21    while(ros::ok()) {
22      // Create and fill in the message.  The other four
23      // fields, which are ignored by turtlesim, default to 0.
24      geometry_msgs::Twist msg;
25      msg.linear.x = double(rand())/double(RAND_MAX);
26      msg.angular.z = 2*double(rand())/double(RAND_MAX) - 1;
27
28      // Publish the message.
29      pub.publish(msg);
30
31      // Send a message to rosout with the details.
32      ROS_INFO_STREAM("Sending␣random␣velocity␣command:"
33        << "␣linear=" << msg.linear.x
34        << "␣angular=" << msg.angular.z);
35
36      // Wait until it's time for another iteration.
37      rate.sleep();
38    }
39  }
```

Criando um Publisher

Incluir os arquivos de cabeçalho necessários

- `#include <geometry_msgs/Twist.h>` - contém a classe necessária para criar mensagens do tipo que precisamos;
- `#include <stdlib.h>` - para usar `rand()` e `RAND_MAX`

Criando um Publisher

Criar um objeto da classe ros::Publisher

```
ros::Publisher nome-do-objeto = node-handle.advertise<tipo-da-mensagem>
    (nome-do-topico, tamanho-da-fila);
```

- *nome-do-objeto*: Usar um nome que faça sentido, como `cmdVelPub` ou apenas `pub` caso só exista um Publisher;
- *node-handle*: Objeto da classe ros::NodeHandle criado previamente;
- *tipo-da-mensagem*: Nome da classe do tipo de mensagem que será publicado;

Criando um Publisher

- *nome-do-topico*: Escolher um nome que faça sentido. No nosso caso vamos publicar em um tópico específico que foi criado pelo turtlesim;
- *tamanho-da-fila*: Caso mensagens estejam sendo publicadas mais rápido do que consumidas, o ROS vai guardar essas mensagens em uma fila. Usar um número grande como 1000 geralmente evita qualquer problema.

Criando um Publisher

Selecionar uma semente para o gerador de números aleatórios

```
srand(time(0));
```

Criando um Publisher

Criar e preencher a mensagem

```
geometry_msgs::Twist msg;
msg.linear.x = double(rand())/double(RAND_MAX);
msg.angular.z = 2*double(rand())/double(RAND_MAX) - 1;
```

Esse código preenche os campos velocidade linear com um valor entre 0 e 1 e velocidade angular com um número entre -1 e 1. O turtlesim ignora os outros campos.

Criando um Publisher

Publicar a mensagem:

```
pub.publish(msg);
```

Criando um Publisher

Para publicar as mensagens de forma contínua e periódica, usamos um loop while. A condição de repetição do loop é:

```
ros::ok()
```

Essa função retorna true enquanto o nosso nó estiver rodando corretamente. Ela só retornará false caso o nó seja encerrado, nos seguintes casos: o nó seja encerrado com rosnode kill, ou com Ctrl-C, ou chamando a função ros::shutdown() dentro do código, ou iniciando outro nó com o mesmo nome.

Criando um Publisher

Criar uma taxa de publicação:

```
ros::Rate rate(2);
```

E dentro do loop, chamar a função:

```
rate.sleep();
```

Isso vai fazer com que o ROS espere um tempo entre cada iteração do loop. O ROS vai calcular esse tempo automaticamente de forma que o loop seja executado 2 vezes por segundo.

Criando um Publisher

Usar ROS_INFO_STEAM para imprimir os valores publicados na tela.

```
ROS_INFO_STEAM(“Sending random velocity command:”
    << “ linear=” << msg.linear.x
    << “ angular=” << msg.angular.z);
```

Criando um Publisher

Para compilar o pubvel:

- Adicionar o novo executável no arquivo CmakeLists.txt.
- `catkin_make`

Criando um Publisher

Para executar:

```
roscore
rosrun ros_e_gazebo pubvel
rosrun turtlesim turtlesim_node
```

Verificar a frequência de publicação:

```
rostopic hz /turtle1/cmd_vel
```

Criando um Subscriber

Agora nós vamos criar um programa que subscreve ao tópico /turtle1/pose, no qual o turtlesim_node publica. As mensagens nesse tópico descrevem a **pose** da tartaruga, um termo que se refere à posição e orientação.

subpose.cpp

```cpp
// This program subscribes to turtle1/pose and shows its
// messages on the screen.
#include <ros/ros.h>
#include <turtlesim/Pose.h>
#include <iomanip> // for std::setprecision and std::fixed

// A callback function.  Executed each time a new pose
// message arrives.
void poseMessageReceived(const turtlesim::Pose& msg) {
  ROS_INFO_STREAM(std::setprecision(2) << std::fixed
    << "position=(" <<  msg.x << "," << msg.y << ")"
    << " direction=" << msg.theta);
}

int main(int argc, char **argv) {
  // Initialize the ROS system and become a node.
  ros::init(argc, argv, "subscribe_to_pose");
  ros::NodeHandle nh;

  // Create a subscriber object.
  ros::Subscriber sub = nh.subscribe("turtle1/pose", 1000,
    &poseMessageReceived);

  // Let ROS take over.
  ros::spin();
}
```

Criando um Subscriber

- Uma diferença importante entre publicar e subscrever é que o Subscriber não sabe quando as mensagens vão chegar, portanto nós precisamos escrever um código que será chamado automaticamente toda vez que uma nova mensagem chegue. Esse código é chamado de uma função callback.

```
void nome-da-funcao( const nome-do-pacote::nome-do-tipo &msg ) { … }
```

Criando um Subscriber

- O corpo da função tem acesso a todos os campos da mensagem recebida através da variável msg, e podemos utilizar esses dados da maneira que quisermos. No nosso caso, nós apenas imprimimos os campos da mensagem na tela.
- É necessário incluir o arquivo `turtlesim/Pose.h`
- A função callback sempre retorna `void`.

Criando um Subscriber

- Criar um objeto subscriber:

```
ros::Subscriber nome-do-objeto = node-handle.subscribe( nome-do-topico,
                tamanho-da-fila, ponteiro-para-funcao-callback );
```

- Para usar o ponteiro basta colocar um & antes do nome da função

Criando um Subscriber

O ROS só vai chamar a função callback quando passarmos o controle do programa para ele. Existem duas formas de fazer isso:

1ª Forma:

```
ros::spinOnce();
```

Essa forma pede para o ROS executar todos os callbacks e então retornar o controle para nós.

Criando um Subscriber

Essa forma é útil quando queremos fazer alguma coisa entre as execuções dos callbacks:

```
While (ros::ok()) {

    // Fazer alguma tarefa. Por exemplo, publicar mensagens

    ros::spinOnce();

}
```

Criando um Subscriber

2ª Forma:

```
ros::spin();
```

Essa forma diz para o ROS continuar executando os callbacks sempre que necessário indefinidamente, até que o nó seja encerrado.

Criando um Subscriber

Para compilar:

- Adicionar o novo executável no arquivo `CmakeLists.txt`.
- `catkin_make`

Criando um Subscriber

Para executar:

```
roscore

rosrun turtlesim turtlesim_node

rosrun ros_e_gazebo subpose

rosrun ros_e_gazebo pubvel
```

Arquivos Launch

Arquivos Launch

Esses arquivos nos permitem executar vários nós ao mesmo tempo. A idéia é listar todos os nós que queremos executar em uma sintaxe xml específica, podendo definir configurações para cada nó e passar argumentos.

Arquivos Launch

example.launch:

```
<launch>

    <node pkg=”turtlesim” type=”turtlesim_node” name=”turtlesim” />

    <node pkg=”turtlesim” type=”turtle_teleop_key” name=”teleop_key” />

    <node pkg=”simuladores” type=”subpose” name=”pose_subscriber”
        output=”screen” />

</launch>
```

Arquivos Launch

Esse arquivo executa todos os nós do exemplo anterior, mas com um único comando:

```
roslaunch simuladores exemplo.launch
```

Criando Arquivos Launch

- Tudo deve estar envolvido em uma tag launch: `<launch> … </launch>`
- Cada nó é chamado por uma tag node:

```
<node pkg=”pacote” type=”executavel” name=”nome-do-no” />
```

- O atributo `name` sobrescreve o nome definido no código do nó
- O atributo `output=”screen”` serve para que a saída do nó seja impressa na tela.

Criando Arquivos Launch

É possível incluir outros arquivos launch:

```
<include file=”caminho-para-o-arquivo-launch” />
```

Criando Arquivos Launch

É possível incluir argumentos:

```
<arg name=”nome-do-argumento” default=”valor-padrao” />
```

Essa tag define um argumento, que pode ser passado pelo comando roslaunch e utilizado em qualquer lugar do arquivo através da sintaxe:

```
$(arg nome-do-argumento)
```

Arquivos Launch

Por exemplo:

```
<launch>

    <arg name=”node_name” default=”hello” />

    <node pkg=”simuladores” type=”hello” name=”$(arg node_name)” />

</launch>
```

Arquivos Launch

Chamar esse arquivo com o comando:

```
roslaunch simuladores hello.launch node_name:=hi
```

Sobrescrevemos o valor padrão do argumento `node_name` que era `“hello”` com o novo valor `“hi”`.

Exercício

- Usando os conceitos que estudamos e os programas que escrevemos, crie um nó que move a tartaruga do turtlesim para uma posição (X, Y) determinada.
- Escreva um arquivo launch que permita executar o turtlesim e o nó que foi criado com apenas um comando.

Exercício 2

- Modifique o programa do Exercício anterior para subscrever a um tópico chamado /goal.
- Deverá ser possível, durante a execução do programa, publicar mensagens representando posições no tópico /goal.
- A tartaruga deverá tratar essa posição como seu novo objetivo e se mover para lá.

Gazebo Simulator

Gazebo

O Gazebo é um simulador 3D que tem a habilidade de simular, de forma precisa e eficiente, populações de robôs em ambientes indoor e outdoor complexos. Ele já vem com uma base contendo diversos modelos de objetos, robôs e sensores, mas permite também que criemos nossos próprios ambientes e modelos. É capaz de simular vários tipos de sensores como sonar, lidar, GPS e câmera.

Instalação do Gazebo

- Se, ao instalarmos o ROS, escolhermos o pacote **`ros-indigo-desktop-full`**, o Gazebo 2 já vem instalado. Essa é a versão indicada para o ROS Indigo e é a que vamos utilizar no curso.

- Para instalar o Gazebo de forma independente, e também numa versão mais nova, basta seguir as instruções na página a seguir: http://gazebosim.org/tutorials?tut=install_ubuntu&cat=install

Iniciar o Gazebo

- Para iniciar o Gazebo, basta abrir um terminal e executar o comando:

  ```
  gazebo
  ```

- Isso inicializará o Gazebo como um programa independente do ROS.
- Para utilizar o Gazebo juntamente com o ROS, é necessário o pacote `gazebo_ros`.
- Caso o pacote ainda não esteja instalado:

  ```
  sudo apt-get install ros-indigo-gazebo-ros
  ```

Iniciar o Gazebo

- Para iniciar o Gazebo como parte do ROS, executar os seguintes comandos em dois terminais independentes:

```
roscore
```

```
rosrun gazebo_ros gazebo
```

- Dessa vez o Gazebo será iniciado como um nó do ROS, capaz de publicar e subscrever em tópicos.

```
rosnode list
```

Componentes do Gazebo

Mundos

O mundo que será simulado pelo Gazebo pode conter diversos objetos, robôs e sensores. Diversas características podem ser alteradas, como vento, luminosidade e mesmo as regras da física. Os mundos são descritos em arquivos com extensão `.world`, que são escritos numa linguagem de marcação chamada SDF (Simulation Description Format).

Mundos

Exemplo: `empty.world`

```
<?xml version="1.0" ?>
<sdf version="1.4">
  <world name="default">
    <!-- A global light source -->
    <include>
      <uri>model://sun</uri>
    </include>
    <!-- A ground plane -->
    <include>
      <uri>model://ground_plane</uri>
    </include>
  </world>
</sdf>
```

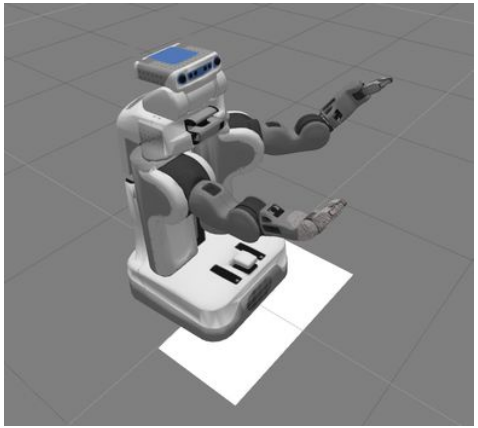
Mundos

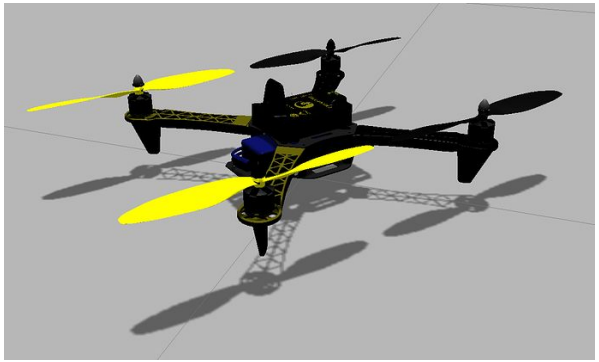
Outros exemplos de mundos:

```
roslaunch gazebo_ros willowgarage_world.launch

roslaunch gazebo_ros shapes_world.launch

roslaunch gazebo_ros rubble_world.launch
```

Modelos

Modelos representam elementos da simulação: objetos, sensores ou mesmo robôs. Modelos são descritos em arquivos com a extensão `.sdf`, e devem contar uma única tag `<model> … </model>`. São escritos usando a mesma linguagem SDF dos arquivos `world`.





Modelos de Robôs no ROS

- O ROS também utiliza arquivos para representar modelos de robôs, porém utiliza uma linguagem diferente, a URDF - Universal Robotic Description Format.
- Essa linguagem é muito parecida com a SDF do Gazebo, porém mais limitada: ela só pode ser usada para representar robôs, e não objetos estáticos, e não possui alguns elementos exclusivos de simulação.
- Quando o Gazebo encontra um arquivo URDF, ele primeiro converte para SDF e só então carrega aquele modelo no ambiente de simulação.

Plugins

- Plugins são programas que nos permitem interagir com o ambiente de simulação do Gazebo.
- Através de plugins é possível controlar robôs, simular sensores, modificar as leis da física, criar novos robôs ou objetos, etc.
- Plugins são escritos em C++.

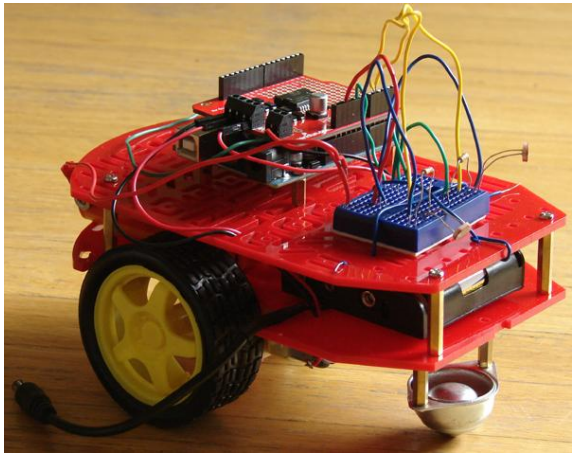
Construindo nossa própria simulação

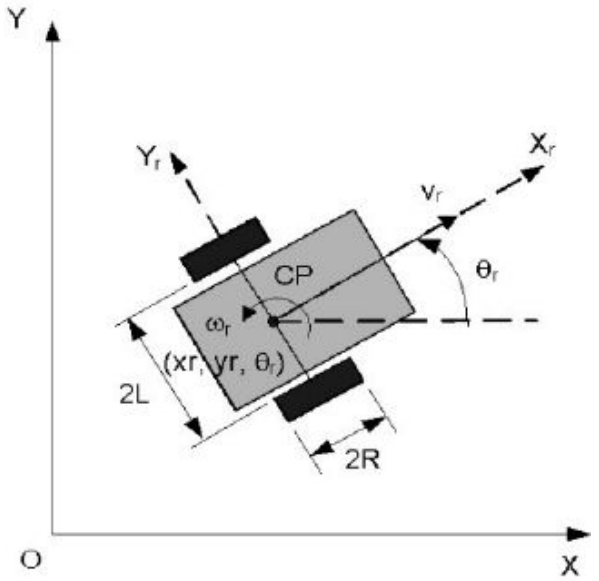
Construindo nossa simulação

- Para compreendermos melhor como utilizar os componentes do Gazebo, vamos criar alguns pacotes que definem uma simulação e executar no Gazebo.
- Vamos criar os arquivos que definem um robô de direção diferencial. Esse tipo de robô possui duas rodas motorizadas que se movem de forma independente e uma caster wheel, uma roda que se move livremente e serve para sustentar o robô.

Construindo nossa simulação

- Como resultado o robô é capaz de se mover para frente e para trás, e também de girar em torno do eixo Z. São movimentos muito similares aos da tartaruga no turtlesim.





Construindo nossa simulação

- Primeiro, vamos criar três novos pacotes:

```
cd ~/catkin_ws/src

catkin_create_pkg mybot_gazebo gazebo_ros

catkin_create_pkg mybot_description

catkin_create_pkg mybot_control

cd ~/catkin_ws

catkin_make
```

Criando o mundo

- Vamos criar o arquivo `.world` que define o mundo da simulação:

```
roscd mybot_gazebo

mkdir launch worlds

cd worlds

gedit mybot.world
```

Criando o mundo

```
<?xml version="1.0"?>
<sdf version="1.4">
    <world name="myworld">
        <include>
            <uri>model://sun</uri>
        </include>
        <include>
            <uri>model://ground_plane</uri>
        </include>
    </world>
</sdf>
```

Criando o mundo

- Esse é um arquivo `.world` bem básico, que apenas define o nome do mundo e inclui dois modelos: um piso no chão e o sol para fornecer iluminação.
- Vamos criar um arquivo `.launch` que inicializará o Gazebo como um nó do ROS e carrega o mundo de simulação que criamos.

```
roscd mybot_gazebo/launch

gedit mybot_world.launch
```

Criando o mundo

- Inserir no arquivo:

```
<launch>
    <include file="$(find gazebo_ros)/launch/empty_world.launch">
        <arg name="world_name"
            value="$(find mybot_gazebo)/worlds/mybot.world" />
        <arg name="gui" value="true" />
    </include>
</launch>
```

Criando o mundo

- Agora vamos executar o arquivo `.launch`:

```
roslaunch mybot_gazebo mybot_world.launch
```

Criando o Modelo

- Agora vamos criar o modelo do robô diferencial.
- Como estamos trabalhando com o ROS, vamos usar a linguagem URDF para definir o nosso robô.

```
roscd mybot_description

mkdir urdf

cd urdf

gedit mybot.urdf
```

Criando o Modelo

- Um modelo de um robô é formado por links e joints.
- Links são as partes do robô: o corpo, cada roda, eventuais sensores, etc.
- Joints são as ligações entre os links. Existem diversos tipos de joints, e cada tipo define como o link é capaz de se mover em relação ao outro link ao qual está ligado.
- Usamos as tags xml para definir os diversos elementos que formam o robô.

Criando o Modelo

- O modelo mais básico de um robô é:

```
<?xml version="1.0"?>

<robot name="mybot">

</robot>
```

Criando o Modelo

- Vamos adicionar um link para ser o corpo do robô.
- Será um bloco retangular de 0,4m x 0,2m x 0,1m, com massa de 50 gramas.
- Esse link se chamará “chassis”.
- Adicione o código a seguir dentro da tag `<robot>`.

```
<link name='chassis'>
    <collision>
      <origin xyz="0 0 0.1" rpy="0 0 0"/>
      <geometry>
        <box size="0.4 0.2 0.1"/>
      </geometry>
    </collision>
    <visual>
      <origin xyz="0 0 0.1" rpy="0 0 0"/>
      <geometry>
        <box size="0.4 0.2 0.1"/>
      </geometry>
    </visual>
    <inertial>
      <origin xyz="0 0 0.1" rpy="0 0 0"/>
      <mass value="50"/>
      <inertia ixx="0.208" ixy="0" ixz="0" iyy="0.708" iyz="0" izz="0.708"/>
    </inertial>
</link>
```

Criando o Modelo

- Cada link é formado por três elementos.
- O elemento `<collision>`

```
<collision>
    <origin xyz="0 0 0.1" rpy="0 0 0"/>
    <geometry>
      <box size="0.4 0.2 0.1"/>
    </geometry>
</collision>
```

É utilizado para detecção de colisões.

Criando o Modelo

- O elemento `<visual>`

```
<visual>
    <origin xyz="0 0 0.1" rpy="0 0 0"/>
    <geometry>
      <box size="0.4 0.2 0.1"/>
    </geometry>
</visual>
```

É usado pelo motor gráfico do Gazebo para desenhar o objeto na tela.

Criando o Modelo

- O elemento `<inertial>`

```
<inertial>
    <origin xyz="0 0 0.1" rpy="0 0 0"/>
    <mass value="50"/>
    <inertia ixx="0.208" ixy="0" ixz="0" iyy="0.708" iyz="0" izz="0.708"/>
</inertial>
```

  É usado pelo simulador de física do Gazebo.

- A tag `<inertia>` representa a matriz de inércia do link. O cálculo dessa matriz é muito complexo, portanto estamos usando um formato padrão.

Criando o Modelo

- No nosso caso, esses três elementos são iguais: representam um bloco retangular com as dimensões que definimos.
- No caso de objetos muito complexos, o cálculo de colisões exigiria muito processamento do computador. Nesse caso, é possível definir um elemento `<visual>` com toda a complexidade necessária, porém um elemento `<collision>` mais simplificado para aliviar o processamento.

Criando um Modelo

- Se abrirmos o modelo no Gazebo neste ponto, aparecerá um bloco branco no centro do mundo de simulação.
- O formato URDF não suporta a definição de cores, nem de outras características que são exclusivas da simulação.
- Nesse caso, podemos definir essas características usando a tag `<gazebo>`. O ROS ignora essa tag, porém o Gazebo processa a tag quando coverte o URDF para SDF.

Criando um Modelo

- Adicione o seguinte após o link “chassis”:

```
<gazebo reference="chassis">
    <material>Gazebo/Orange</material>
</gazebo>
```

- Agora o bloco ficará laranja!

Criando um Modelo

- Por fim, o Gazebo não aceita que o link base do robô possua inércia.
- Vamos criar um link falso, sem inércia, e ligá-lo ao chassis através de uma junta fixa. Adicione, antes do link chassis:

```
<link name="footprint"/>

<joint name="base_joint" type="fixed">
  <parent link="footprint"/>
  <child link="chassis"/>
</joint>
```

Criando um Modelo

- Agora vamos modificar nosso arquivo launch para carregar o robô dentro do mundo de simulação.
- Adicione o seguinte no arquivo launch:

```
<node name="mybot_spawn" pkg="gazebo_ros" type="spawn_model"
  args="-file $(find mybot_description)/urdf/mybot.urdf -urdf -model mybot"
/>
```

Criando um Modelo

- Execute o arquivo launch:

```
roslaunch mybot_gazebo mybot_world.launch
```

- Deverá aparecer um bloco laranja!

Criando um Modelo

- Vamos agora adicionar a caster wheel.
- Vamos representar a caster wheel como uma esfera de raio 0.05m e massa 5 gramas, fixa no corpo do robô, que se arrasta pelo chão com pouco atrito. Essa é uma simplificação que funciona de forma bem próxima à caster wheel real.
- Novamente, vamos adicionar os elementos `<collision>`, `<vision>` e `<inertial>`.
- Vamos também adicionar um elemento `<gazebo>` com informações que são específicas da simulação.

```
<joint name="fixed" type="fixed">
  <parent link="chassis"/>
  <child link="caster_wheel"/>
</joint>
```

```
<link name="caster_wheel">
  <collision>
    <origin xyz="-0.15 0 0.05" rpy="0 0 0"/>
    <geometry>
      <sphere radius="0.05"/>
    </geometry>
  </collision>
  <visual>
    <origin xyz="-0.15 0 0.05" rpy="0 0 0"/>
    <geometry>
      <sphere radius="0.05"/>
    </geometry>
  </visual>
  <inertial>
    <origin xyz="-0.15 0 0.05" rpy="0 0 0"/>
    <mass value="5"/>
    <inertia ixx="0.005" ixy="0" ixz="0" iyy="0.005" iyz="0" izz="0.005" />
  </inertial>
</link>
```

```
<gazebo reference="caster_wheel">
  <mu1>0.0</mu1>
  <mu2>0.0</mu2>
  <material>Gazebo/Red</material>
</gazebo>
```

Criando um Modelo

- Agora vamos adicionar as duas rodas.
- As rodas serão representadas por cilindros com 0.1m de raio e 0.05m de altura. Cada uma tem massa de 5 gramas.
- As rodas são presas no corpo do robô através de joints do tipo “continuous”. Esse tipo de joint representa uma rotação contínua ao redor de um determinado eixo. Escolhendo o eixo Y (direita - esquerda), criamos um movimento de roda.

```
<joint name="right_wheel_hinge" type="continuous">
  <parent link="chassis"/>
  <child link="right_wheel"/>
  <origin xyz="0 -0.125 0.1" rpy="0 0 0" />
  <axis xyz="0 1 0" rpy="0 0 0" />
  <limit effort="100" velocity="100"/>
  <joint_properties damping="0.0" friction="0.0"/>
</joint>
```

```
<link name="right_wheel">
    <collision>
        <origin xyz="0 0 0" rpy="0 1.57 1.57" />
        <geometry>
          <cylinder length="0.05" radius="0.1"/>
        </geometry>
    </collision>
    <visual>
        <origin xyz="0 0 0" rpy="0 1.57 1.57" />
        <geometry>
          <cylinder length="0.05" radius="0.1"/>
        </geometry>
    </visual>
    <inertial>
        <origin xyz="0 0 0" rpy="0 1.57 1.57" />
        <mass value="5"/>
        <inertia ixx="0.0135" ixy="0" ixz="0" iyy="0.0135" iyz="0" izz="0.025" />
    </inertial>
</link>
```

```
<gazebo reference="right_wheel">
      <mu1 value="1.0"/>
      <mu2 value="1.0"/>
      <material>Gazebo/Black</material>
</gazebo>
```

```
<joint name="left_wheel_hinge" type="continuous">
      <parent link="chassis"/>
      <child link="left_wheel"/>
    <origin xyz="0 0.125 0.1" rpy="0 0 0" />
      <axis xyz="0 1 0" rpy="0 0 0" />
      <limit effort="100" velocity="100"/>
      <joint_properties damping="0.0" friction="0.0"/>
</joint>
```

```
<link name="left_wheel">
    <collision>
        <origin xyz="0 0 0" rpy="0 1.57 1.57" />
        <geometry>
          <cylinder length="0.05" radius="0.1"/>
        </geometry>
    </collision>
    <visual>
        <origin xyz="0 0 0" rpy="0 1.57 1.57" />
        <geometry>
          <cylinder length="0.05" radius="0.1"/>
        </geometry>
    </visual>
    <inertial>
        <origin xyz="0 0 0" rpy="0 1.57 1.57" />
        <mass value="5"/>
        <inertia ixx="0.0135" ixy="0" ixz="0" iyy="0.0135" iyz="0" izz="0.025" />
    </inertial>
</link>
```

```
<gazebo reference="left_wheel">
      <mu1 value="1.0"/>
      <mu2 value="1.0"/>
      <material>Gazebo/Black</material>
</gazebo>
```

Criando um Modelo

- Execute novamente o arquivo launch.

```
roslaunch mybot_gazebo mybot_world.launch
```

- Agora nosso robô está completo!

Controlando o Robô

- Para controlar o robô, vamos utilizar um plugin fornecido pelo Gazebo.
- Vamos adicionar mais um código no arquivo URDF, instruindo o Gazebo a carregar o plugin e passando as configurações necessárias.
- Adicione no final do arquivo URDF o código a seguir.

```
<gazebo>
    <plugin name="differential_drive_controller"
            filename="libgazebo_ros_diff_drive.so">
        <alwaysOn>true</alwaysOn>
        <updateRate>100</updateRate>
        <leftJoint>left_wheel_hinge</leftJoint>
        <rightJoint>right_wheel_hinge</rightJoint>
        <wheelSeparation>0.25</wheelSeparation>
        <wheelDiameter>0.2</wheelDiameter>
        <torque>20</torque>
        <commandTopic>mybot/cmd_vel</commandTopic>
        <odometryTopic>mybot/odom_diffdrive</odometryTopic>
        <odometryFrame>odom</odometryFrame>
        <robotBaseFrame>footprint</robotBaseFrame>
      </plugin>
</gazebo>
```

Controlando o Robô

- Esse plugin subscreve no tópico `/mybot/cmd_vel` e aguarda mensagens de comando de velocidade de forma similar à tartaruga do turtlesim.
- E assim como o turtlesim, ele publica a posição do robô em um tópico chamado `/mybot/odom_diffdrive`.

Adicionando Sensores

Adicionando sensores

- Para que o robô possa sentir o ambiente ao seu redor, precisamos adicionar sensores a ele.
- Os sensores publicam sua informação em tópicos do ROS.
- Existe um tipo de mensagem específica para cada tipo de sensor, por exemplo `sensor_msgs/Image` para câmeras, `sensor_msgs/LaserScan` para lasers, `sensor_msgs/NavSatFix` para GPS, etc.
- No Gazebo os sensores são implementados através de plugins.

Adicionando sensores

- Para adicionar um sensor ao nosso modelo, é necessário incluir três novos elementos:
- Um link, que representa o corpo físico do sensor;
- Uma joint, ligando o sensor ao corpo do robô;
- Um plugin, que implementa o funcionamento do sensor.
- O Gazebo fornece plugins para diversos tipos de sensores, como lasers, lidars, câmeras, IMU, entre outros.

Laser

- Vamos adicionar um laser ao nosso robô;
- Esse é um tipo de sensor de distância que emite diversos feixes de luz e mede quanto tempo a luz demora para ir até um obstáculo e voltar. Assim ele é capaz de calcular qual a distância até aquele ponto.
- Vamos adicionar um sensor que emite 8 lasers espalhados em todas as direções.

```
<joint name="laser_joint" type="fixed">
    <origin xyz="0.15 0 0.2" rpy="0 0 0"/>
    <parent link="chassis"/>
    <child link="laser_link"/>
</joint>
```

```
<link name="laser_link">
    <collision>
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <geometry>
            <box size="0.1 0.1 0.1"/>
        </geometry>
    </collision>
    <visual>
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <geometry>
            <box size="0.1 0.1 0.1"/>
        </geometry>
    </visual>
    <inertial>
        <mass value="1e-5" />
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <inertia ixx="1e-6" ixy="0" ixz="0" iyy="1e-6" iyz="0" izz="1e-6" />
    </inertial>
</link>
```

```
<gazebo reference="laser_link">
    <sensor type="ray" name="laser">
        <pose>0 0 0 0 0 0</pose>
        <visualize>true</visualize>
        <update_rate>40</update_rate>
        <ray>
            <scan>
                <horizontal>
                    <samples>8</samples>
                    <resolution>1</resolution>
                    <min_angle>-3.14159</min_angle>
                    <max_angle>3.14159</max_angle>
                </horizontal>
            </scan>
            <range>
                <min>0.10</min>
                <max>10.0</max>
                <resolution>0.01</resolution>
            </range>
        </ray>
```

```
        <plugin name="laser_controller" filename="libgazebo_ros_laser.so">
            <topicName>mybot/scan</topicName>
            <frameName>base_link</frameName>
        </plugin>
    </sensor>
</gazebo>
```

Laser

- Esse sensor que acabamos de adicionar publica mensagens do tipo `sensor_msgs/LaserScan` no tópico `mybot/scan` (o nome do tópico pode ser configurado no código do sensor).
- Essa mensagem possui um campo chamado `ranges`, que é um array contendo as medidas de cada um dos lasers.
- Vamos agora escrever um nó que é capaz de ler essas mensagens.
- Na pasta `src` do pacote `mybot_control`, crie um arquivo chamado `scansub.cpp`.

```
#include <ros/ros.h>
#include <sensor_msgs/LaserScan.h>
#include <sstream>

void laserScanCallback(const sensor_msgs::LaserScan& msg)
{
    std::ostringstream oss;
    oss << "Ranges = [ ";

    for (int i = 0; i < msg.ranges.size(); i++)
    {
        oss << msg.ranges[i] << " ";
    }

    oss << " ];";

    ROS_INFO_STREAM( oss.str() );
}
```

```
int main(int argc, char** argv)
{

    ros::init(argc, argv, "laser_scan_sub");
    ros::NodeHandle nh;

    ros::Subscriber scan_sub = nh.subscribe("mybot/scan",
        1000, &laserScanCallback);

    ros::spin();
}
```

Laser

- Esse programa é muito parecido com o subpose, porém ao invés de usarmos mensagens do tipo `turtlesim/Pose` estamos usando mensagens do tipo `sensor_msgs/LarserScan`.
- Dentro da função `laserScanCallback`, o objeto `msg` contém a mensagem que foi recebida. O array `msg.ranges` contém as medidas de cada um dos sensores.
- Fazemos um for para ler cada uma das medidas e concatenar em uma string para imprimir na tela.

Laser

- Para compilar o programa:
- Adicionar as dependências no arquivo `package.xml`

```
<build_depend>roscpp</build_depend>
<build_depend>sensor_msgs</build_depend>

<run_depend>roscpp</run_depend>
<run_depend>sensor_msgs</run_depend>
```

Laser

- Adicionar também as dependências no arquivo `CMakeLists.txt`:

```
find_package(catkin REQUIRED COMPONENTS
    roscpp
    sensor_msgs
)
```

- Adicionar o novo executável no arquivo `CMakeLists.txt`:

```
add_executable(scansub src/scansub.cpp)
target_link_libraries(scansub ${catkin_LIBRARIES})
```

Laser

- Compilar o pacote

```
cd ~/catkin_ws

catkin_make
```

- Executar:

```
rosrun mybot_control scansub
```

Câmera

- Vamos agora adicionar uma câmera.
- Adicione o seguinte código ao modelo:

```
<joint name="camera_joint" type="fixed">
    <origin xyz="0.15 0 0.175" rpy="0 0 0"/>
    <parent link="chassis"/>
    <child link="camera"/>
</joint>
```

```
<link name="camera">
    <collision>
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <geometry>
            <box size="0.05 0.05 0.05"/>
        </geometry>
    </collision>
    <visual>
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <geometry>
            <box size="0.05 0.05 0.05"/>
        </geometry>
    </visual>
    <inertial>
        <mass value="0.1" />
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <inertia ixx="1e-6" ixy="0" ixz="0" iyy="1e-6" iyz="0" izz="1e-6" />
    </inertial>
</link>
```

```
<gazebo reference="camera">

      <material>Gazebo/Blue</material>

      <sensor type="camera" name="camera1">
        <update_rate>30.0</update_rate>
        <camera name="head">
          <horizontal_fov>1.3962634</horizontal_fov>
          <image>
            <width>800</width>
            <height>800</height>
            <format>R8G8B8</format>
          </image>
          <clip>
            <near>0.02</near>
            <far>300</far>
          </clip>
        </camera>
```

```
        <plugin name="camera_controller" filename="libgazebo_ros_camera.so">
          <alwaysOn>true</alwaysOn>
          <updateRate>0.0</updateRate>
          <cameraName>mybot/camera</cameraName>
          <imageTopicName>image_raw</imageTopicName>
          <cameraInfoTopicName>camera_info</cameraInfoTopicName>
          <frameName>camera_link</frameName>
          <hackBaseline>0.07</hackBaseline>
          <distortionK1>0.0</distortionK1>
          <distortionK2>0.0</distortionK2>
          <distortionK3>0.0</distortionK3>
          <distortionT1>0.0</distortionT1>
          <distortionT2>0.0</distortionT2>
        </plugin>
    </sensor>
</gazebo>
```

Câmera

- A câmera publica mensagens do tipo `sensor_msgs/Image` no tópico `mybot/camera/image_raw`.
- Para visualizar as imagens, vamos utilizar o nó `image_view` do pacote `image_view`:

```
rosrun image_view image_view image:=/mybot/camera/image_raw
```

Simulador de Quadrotores

Simulador de Quadrotores

- Agora nós vamos usar o pacote viscap_gazebo para simular quadrotores dentro do Gazebo.
- Esse pacote contém os arquivos necessários para executar a simulação: o modelo do quadrotor, o plugin para o controle, alguns cenários com os quais o robô pode interagir.
- O quadrotor dessa simulação contém vários sensores que utilizaremos para fazê-lo navegar pelo mundo da simulação, como câmeras, sensores de distância e lasers.

Instalação

- As instruções para a instalação encontram-se na página https://github.com/viscap/viscap_gazebo

Descrição do pacote

O pacote viscap_gazebo contém recursos organizados em diversas pastas, das quais as principais são:

- `worlds` e `models`: Contém os arquivos .world que descrevem os cenários de simulação e modelos de objetos presentes nesses cenários;
- `urdf`, `meshes` e `plugins`: Contém os arquivos que constituem o modelo do quadrotor, incluindo o urdf, o modelo visual e o plugin de controle;
- `launch`: Contém os arquivos launch que usaremos para iniciar os diferentes cenários de simulação e carregar drones dentro do Gazebo;

Vamos dar uma olhada mais a fundo no modelo do quadrotor.

Modelo do quadrotor

- A pasta urdf contém os arquivos que formam o modelo urdf do quadrotor.
- A extensão utilizada nesses arquivos é .xacro. Isso significa que os arquivos utilizam o xacro, um programa do ROS que processa esses arquivos e gera urdf.
- O xacro incrementa o urdf com diversas funcionalidades úteis como criação de variáveis e macros para evitar repetição de código, inclusão de arquivos externos, operações matemáticas, entre outras.

Modelo do quadrotor

- O arquivo principal é o `quadrotor.urdf.xacro`.
- Todo arquivo xacro deve começar com o seguinte código:

```
<?xml version="1.0"?>

<robot name="quadrotor_full" xmlns:xacro="http://wiki.ros.org/xacro">
</robot>
```

- Onde “`quadrotor_full`” é o nome que escolhemos para nosso quadrotor, mas poderia ser qualquer outro nome.
- Dentro da tag `robot` colocamos o código que descreve o robô.

Modelo do quadrotor

```
<xacro:property name="ID" value="$(arg id)" />
<xacro:property name="robot_name" value="quadrotor_$(arg id)" />
<xacro:property name="M_PI" value="3.1415926535897931" />
```

- Essas linhas estão definindo “propriedades” que funcionam com variáveis. São valores que poderemos reutilizar ao longo do restante do código.
- As propriedades definidas são o ID do quadrotor, o nome dele (que depende do ID) e o valor de pi.
- O código “`$(arg id)`” indica que o ID pode ser passado como parâmetro a partir de um arquivo externo (repare que a notação é igual à dos arquivos launch).

Modelo do quadrotor

- Para organização e reutilização do código, cada parte do quadrotor está definida em um arquivo diferente.
- O arquivo `quadrotor.urdf.xacro` inclui esses outros arquivos através de linhas como essa:

```
<xacro:include filename=
    "$(find viscap_gazebo)/urdf/quadrotor_base.urdf.xacro" />
```

Modelo do quadrotor

- As diferentes partes do quadrotor estão definidas na forma de macros.
- Macros são trechos de código que podem ser reutilizados facilmente. Basta “chamar” o macro e o xacro incluirá aquele trecho de código automaticamente.
- É possível passar parâmetros para o macro, modificando seu comportamento conforme desejado.

Modelo do quadrotor

- Exemplo:

```
<xacro:sonar_sensor name="sonar" parent="base_link"
sim_name="${robot_name}" ros_topic="sonar_height" update_rate="10"
min_range="0.01" max_range="3.0" field_of_view="${40*M_PI/180}"
ray_count="3">
    <origin xyz="-0.15 0.0 0.0" rpy="0 ${90*M_PI/180} 0"/>
</xacro:sonar_sensor>
```

- Nessa linha é chamado o macro `sonar_sensor`, que adiciona um sonar (sensor de distância) ao quadrotor. Observe que vários parâmetros são passados, como campo de visão e distância máxima.

Modelo do quadrotor

- Observe o valor do parâmetro `field_of_view`:

```
field_of_view="${40*M_PI/180}"
```

- Esse é um exemplo de expressão matemática que será calculada automaticamente pelo xacro. Repare também que foi utilizada a variável que guarda o valor de pi.

Modelo do quadrotor

- O arquivo `quadrotor_base.urdf.xacro` contém a macro que cria o frame do quadrotor

```
<xacro:macro name="quadrotor_base_macro">
    <link name="base_link">

      …

    </link>
  </xacro:macro>
```

Modelo do quadrotor

- Por simplicidade, o quadrotor foi modelado como um corpo rígido composto por apenas um link, cujo bloco inercial tem a seguinte forma:

```
<inertial>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <mass value="1.477" />
    <inertia ixx="0.01152" ixy="0.0" ixz="0.0" iyy="0.01152"
        iyz="0.0" izz="0.0218" />
</inertial>
```

- A tag `inertia` representa a **matriz de inércia** do quadrotor, que foi estudada em aulas anteriores.

Modelo do quadrotor

- A geometria dos blocos `visual` e `collision` é definida em arquivos externos que possuem um modelo detalhado do quadrotor.

```
<visual>
    ...
    <mesh
filename="package://viscap_gazebo/meshes/quadrotor/quadrotor_4.dae"/>
</visual>

<collision>
    ...
    <mesh
filename="package://viscap_gazebo/meshes/quadrotor/quadrotor_4.stl"/>
</collision>
```

Modelo do quadrotor

- O arquivo `quadrotor.urdf.xacro` também executa o macro que carrega o plugin `quadrotor_simple_controller`, que implementa um controlador PID básico que recebe mensagens de comando de velocidade.
- Por fim, são executados os macros que adicionam os sensores embarcados do quadrotor.

Arquivos launch

- A pasta `launch` possui alguns arquivos que usaremos para iniciar a simulação. Executar o comando:

```
roslaunch viscap_gazebo start_simulation.launch
```

Sensores

- O quadrotor que estamos simulando está equipado com diversos sensores que são comuns em drones reais.
- O Gazebo gera medidas simuladas para cada um desses sensores e então publica em tópicos para que possamos acessar a partir de nossa aplicação em ROS.

Sensores

IMU (Inertial Measurement Unit)

- A IMU utiliza uma combinação de acelerômetros, giroscópios e magnetômetros para estimar a orientação, velocidade angular e aceleração linear do quadrotor.
- Publica mensagens do tipo `sensor_msgs/Imu` no tópico `/quadrotor_1/raw_imu`.
- Os outros tópicos que possuem o prefixo “`raw_imu`” contém informações sobre os parâmetros do sensor.

Sensores

GPS:

- O GPS tenta estimar a posição global do quadrotor, retornando suas coordenadas de longitude, latitude e altitude.
- Publica mensagens do tipo `sensor_msgs/NavSatFix` no tópico `/quadrotor_1/fix`.
- Também é capaz de estimar a velocidade linear do quadrotor, publicando essa medida como mensagens do tipo `geometry_msgs/Vector3Stamped` no tópico `/quadrotor_1/fix_velocity`.
- Novamente, outros tópicos que contém o prefixo “fix” contém dados sobre parâmetros do sensor.

Sensores

Bússola:

- Estima o valor do campo magnético terrestre e publica essa informação como mensagens do tipo `geometry_msgs/Vector3Stamped` no tópico `/quadrotor_1/magnetic`.
- Outros tópicos com o prefixo “magnetic” contém parâmetros do sensor.

Sensores

Sonar:

- O sonar é um sensor de distância que aponta para baixo. Sua função é fornecer uma estimativa precisa da altitude do quadrotor (ou seja, a distância até o chão).
- Publica as medidas como mensagens do tipo `sensor_msgs/Range` no tópico `/quadrotor_1/sonar_height`.
- Outros tópicos com o prefixo “sonar_height” contém parâmetros do sensor.

Sensores

Câmeras:

- O quadrotor possui duas câmeras, uma apontada para frente (front) e uma apontada para baixo (bottom).
- Publicam as imagens obtidas nos tópicos `/quadrotor_1/front/image_raw` e `/quadrotor_1/bottom/image_raw`, em mensagens do tipo `sensor_msgs/Image`.
- Outros tópicos que contenham “front” ou “bottom” no tópico contém parâmetros das câmeras e dados que o ROS utiliza para comprimir as imagens.

Sensores

- Para visualizar as imagens produzidas pelas câmeras, podemos usar o image_view:

```
rosrun image_view image_view image:=/quadrotor_1/front/image_raw

rosrun image_view image_view image:=/quadrotor_1/bottom/image_raw
```

Sensores

Laser scanner:

- Montado no topo do quadrotor, esse sensor emite feixes de laser para diversas direções ao redor do drone. Cada um desses lasers, ao incidir sobre algum objeto, é capaz de calcular a distância entre o quadrotor e esse objeto. No fim, temos um array de valores, um para cada laser, representando a distância entre o quadrotor e a superfície sobre a qual esse laser está incidindo.
- É utilizado principalmente para detectar obstáculos ao redor do quadrotor e *landmarks* para criar mapas.

Sensores

Laser scanner:

- O laser publica suas medidas no tópico `/quadrotor_1/scan`, em mensgens do tipo `sensor_msgs/LaserScan`.

Sensores

- Há mais um tópico ao qual temos acesso na simulação, chamado `/quadrotor_1/ground_truth/state`.
- Esse tópico nos fornece a pose (posição + orientação) exatada do quadrotor, utilizando mensagens do tipo `nav_msgs/Odometry`.
- Só é possível obter esse valor exato em simulação. Na vida real esses dados não estão disponíveis, e precisamos estimar a pose do robô através das medidas dos outros sensores.
- Em simulação, porém, esses dados são úteis para testar a performance de nossos algoritmos.

Controle

- Como já foi dito, estamos utilizando o plugin `quadrotor_simple_controller`, que implementa um controlador PID básico para controlar a velocidade do quadrotor.
- Dessa forma, podemos enviar comandos para mover o drone publicando mensagens do tipo `geometry_msgs/Twist` no tópico `/quadrotor_1/cmd_vel`.
- Note que é bem parecido com o exemplo da tartaruga no `turtlesim`, porém agora temos acesso a movimentos em 3 dimensões.

Programando o quadrotor

Programando o quadrotor

- Vamos agora criar dois programas simples para acessar os sensores do quadrotor.
- Nesses programas vamos aprender como ler as medidas da câmera frontal e do laser.
- Essas medidas podem ser utilizadas como entrada para algoritmos de navegação que movem o drone através de ambientes com obstáculos.

Laser

- Vamos escrever um nó que é capaz de ler as mensgens publicadas pelo laser.
- Na pasta `src` do pacote `simuladores`, crie um arquivo chamado `scansub.cpp`.

```
#include <ros/ros.h>
#include <sensor_msgs/LaserScan.h>
#include <sstream>

void laserScanCallback(const sensor_msgs::LaserScan& msg)
{
    std::ostringstream oss;
    oss << "Ranges = [ ";

    for (int i = 0; i < msg.ranges.size(); i++)
    {
        oss << msg.ranges[i] << " ";
    }

    oss << " ];";

    ROS_INFO_STREAM( oss.str() );
}
```

```
int main(int argc, char** argv)
{

    ros::init(argc, argv, "laser_scan_sub");
    ros::NodeHandle nh;

    ros::Subscriber scan_sub = nh.subscribe("/quadrotor_1/scan",
        1000, &laserScanCallback);

    ros::spin();
}
```

Laser

- Esse programa é muito parecido com o subpose, porém ao invés de usarmos mensagens do tipo `turtlesim/Pose` estamos usando mensagens do tipo `sensor_msgs/LarserScan`.
- Dentro da função `laserScanCallback`, o objeto `msg` contém a mensagem que foi recebida. O array `msg.ranges` contém as medidas de cada um dos sensores.
- Utilizamos uma estrutura `for` para ler cada uma das medidas e concatenar em uma string para imprimir na tela.

Laser

- Para compilar o programa:
- Adicionar as dependências no arquivo `package.xml`

```
<build_depend>roscpp</build_depend>
<build_depend>sensor_msgs</build_depend>

<run_depend>roscpp</run_depend>
<run_depend>sensor_msgs</run_depend>
```

Laser

- Adicionar também as dependências no arquivo `CMakeLists.txt`:

```
find_package(catkin REQUIRED COMPONENTS
    roscpp
    sensor_msgs
)
```

- Adicionar o novo executável no arquivo `CMakeLists.txt`:

```
add_executable(scansub src/scansub.cpp)
target_link_libraries(scansub ${catkin_LIBRARIES})
```

Laser

- Compilar o pacote

```
cd ~/catkin_ws

catkin_make
```

- Executar:

```
rosrun simuladores scansub
```

Câmera

- Agora, vamos escrever um nó que lê as imagens publicadas pela câmera frontal do quadrotor.
- Para isso será necessário utilizar o pacote `image_transport`. Esse é um pacote do ROS que contém diversas funções e estruturas para subscrever e publicar imagens. Ele também permite comprimir as imagens para tornar a transmissão mais rápida.
- Usaremos também algumas funções do opencv para visualizar as imagens. O pacote `cv_bridge` será utilizado para converter entre os formatos de imagem do opencv e do ROS.
- Na pasta `src` do pacote `simuladores`, crie um arquivo chamado `camerasub.cpp`.

```
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <opencv2/highgui/highgui.hpp>
#include <cv_bridge/cv_bridge.h>

void imageCallback(const sensor_msgs::ImageConstPtr& msg)
{
  try
  {
    cv::imshow("view", cv_bridge::toCvShare(msg, "bgr8")->image);
    cv::waitKey(30);
  }
  catch (cv_bridge::Exception& e)
  {
    ROS_ERROR("Could not convert from '%s' to 'bgr8'.",
msg->encoding.c_str());
  }
}
```

```
int main(int argc, char **argv)
{
  ros::init(argc, argv, "camerasub");
  ros::NodeHandle nh;
  cv::namedWindow("view");
  cv::startWindowThread();
  image_transport::ImageTransport it(nh);
  image_transport::Subscriber sub = it.subscribe(
    "/quadrotor_1/front/image_raw", 1, imageCallback);
  ros::spin();
  cv::destroyWindow("view");
}
```

Câmera

- Primeiro, incluímos os pacotes necessários.

```
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <opencv2/highgui/highgui.hpp>
#include <cv_bridge/cv_bridge.h>
```

Câmera

- Criamos uma função callback para processar as imagens recebidas:

```
void imageCallback(const sensor_msgs::ImageConstPtr& msg)
```

- Dentro da função callback, utilizaremos a função `cv::imshow` do opencv para mostrar a imagem em uma janela:

```
try
{
  cv::imshow("view", cv_bridge::toCvShare(msg, "bgr8")->image);
  cv::waitKey(30);
}
```

Câmera

- O código:

```
cv_bridge::toCvShare(msg, "bgr8")->image
```

Tenta converter a mensgem recebida (no formato do ROS) para o formato que o opencv utiliza.

- Caso a conversão não funcione, uma mensagem de erro é mostrada:

```
catch (cv_bridge::Exception& e){
    ROS_ERROR("Could not convert from '%s' to 'bgr8'.",
        msg->encoding.c_str());
}
```

Câmera

- A função main inicia ROS e cria uma janela na qual será mostrada a imagem

```
ros::init(argc, argv, "camerasub");
ros::NodeHandle nh;
cv::namedWindow("view");
cv::startWindowThread();
```

Câmera

- Para subscrever ao tópico da câmera um NodeHandle e um Subscriber convencionais não funcionariam. O pacote `image_transport` disponibiliza objetos análogos, mas que são preparados para funcionar com imagens. Note que o Subscriber que definimos é do pacote `image_transport`, não do `ros`:

```
image_transport::ImageTransport it(nh);
image_transport::Subscriber sub = it.subscribe(
  "/quadrotor_1/front/image_raw", 1, imageCallback);
```

Câmera

- Para finalizar, o controle é passado para o ROS:

```
ros::spin();
```

- Quando o programa for encerrado, é necessário destruir a janela que foi criada pelo opencv:

```
cv::destroyWindow("view");
```

Câmera

- Para compilar o programa:
- Adicionar as dependências no arquivo `package.xml`

```
<build_depend>image_transport</build_depend>
<build_depend>cv_bridge</build_depend>

<run_depend>image_transport</run_depend>
<run_depend>cv_bridge</run_depend>
```

Câmera

- Adicionar também as dependências no arquivo `CMakeLists.txt`:

```
find_package(catkin REQUIRED COMPONENTS
    image_transport
    cv_bridge
)
```

- Adicionar o novo executável no arquivo `CMakeLists.txt`:

```
add_executable(camerasub src/camerasub.cpp)
target_link_libraries(camerasub ${catkin_LIBRARIES})
```

Câmera

- Compilar o pacote

```
cd ~/catkin_ws

catkin_make
```

- Executar:

```
rosrun simuladores camerasub
```

Exercício 1

- Crie um nó que mova o quadrotor para uma posição (X, Y, Z) específica.
- Escreva um arquivo launch que permita abrir a simulação e o nó de controle com apenas um comando no terminal.

Exercício 2

- Usando o mapa do IMAV 2017, escreva um nó que mova o quadrotor através do mapa desviando dos obstáculos.
- Para detectar os obstáculos, utilize as medidas do laser.

Serviços

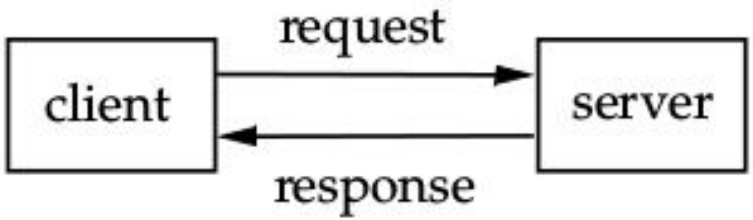
Serviços

Anteriormente nós estudamos como o ROS implementa a comunicação entre programas através da troca de mensagens. Essa é a principal forma de comunicação no ROS. Outra forma de comunicação são os **Serviços**. Os serviços se diferenciam das mensagens em dois pontos:

- Serviços são **bi-direcionais**. Um nó envia informação para outro e aguarda uma resposta em troca.
- Serviços implementam comunicação **um-para-um**. Cada chamada de serviço é iniciada por um nó e a resposta volta para aquele mesmo nó.

Serviços

- A idéia é que um nó **cliente** envia um conjunto de dados chamado de **requisição** para um nó **servidor** e aguarda um retorno.
- O servidor, tendo recebido aqueles dados, toma uma ação e então envia alguns dados de volta para o cliente, chamados de **resposta**.



Serviços

- O conteúdo específico da requisição e da resposta é determinado por um tipo de dados de serviço.
- Assim como o tipo de mensagem, o tipo de dados de serviço é um conjunto de campos nomeados.
- A diferença é que nos serviços o conjunto de campos é dividido em duas partes, uma representando a requisição e outra representando a resposta.

Acessando Serviços pela Linha de Comando

- Vamos ver como encontrar e chamar serviços pela linha de comando com um exemplo.
- Inicie o `turtlesim`.

```
roscore
```

Em outro terminal:

```
rosrun turtlesim turtlesim_node
```

Acessando Serviços pela Linha de Comando

- Para listar todos os serviços disponíveis, executar em outro terminal:

```
rosservice list
```

- Vão aparecer vários serviços oferecidos pelo `turtlesim_node`.

Acessando Serviços pela Linha de Comando

- Para listar os serviços oferecidos por um nó específico, executar o comando:

```
rosnode info <nome-do-nó>
```

- Esse comando já foi visto antes e lista várias informações sobre o nó, entre as quais os serviços que ele oferece.

Acessando Serviços pela Linha de Comando

- Para descobrir qual nó está oferecendo um serviço específico, usar o comando:

```
rosservice node <nome-do-serviço>
```

Acessando Serviços pela Linha de Comando

- Para saber qual o tipo de dados de um serviço, usar o comando:

  rosservice info <nome-do-serviço>

- `Por exemplo:`

  rosservice info /spawn

Acessando Serviços pela Linha de Comando

- Para saber mais informações sobre o tipo de dados de um serviço, usar o comando:

```
rossrv show <tipo-de-dados-do-serviço>
```

- Exemplo:

```
rossrv show turtlesim/Spawn
```

- Os traços (---) separam os campos da requisição e da resposta.

Acessando Serviços pela Linha de Comando

- Note que é possível existir serviços em que a resposta é vazia. Por exemplo:

```
rosservice info /reset

rossrv show std_srvs/Empty
```

- Isso é análogo a uma função em C++ que não retorna nada (void).

Acessando Serviços pela Linha de Comando

- Para chamar um serviço pela linha de comando, basta utilizar a seguinte sintaxe:

```
rosservice call <nome-do-serviço> <conteúdo-da-requisição>
```

- Por exemplo:

```
rosservice call /spawn 3 3 0 Mickey
```

- O efeito desse comando é criar uma tartaruga chamada “Mickey” na posição (x, y) = (3, 3) virada para a direção theta = 0.

Acessando Serviços pela Linha de Comando

- O servidor retornará a resposta. Nesse caso a resposta é o nome da tartaruga criada.
- O servidor também é capaz de retornar uma mensagem de erro caso algum problema ocorra.
- No exemplo, ocorreria um erro caso tentássemos criar uma tartaruga com um nome que já está sendo usado.
- É possível ver esse erro executando o último comando uma segunda vez.

Programa cliente

- Vamos agora ver como escrever um programa cliente, ou seja, capaz de chamar serviços oferecidos por outros nós.
- Na pasta `src` do pacote `simuladores`, criar o arquivo `spawn_turtle.cpp`.
- Vamos escrever um programa que cria uma tartaruga dentro do turtlesim.

```
1  // This program spawns a new turtlesim turtle by calling
2  // the appropriate service.
3  #include <ros/ros.h>
4
5  // The srv class for the service.
6  #include <turtlesim/Spawn.h>
7
8  int main(int argc, char **argv) {
9    ros::init(argc, argv, "spawn_turtle");
10   ros::NodeHandle nh;
11
12   // Create a client object for the spawn service.  This
13   // needs to know the data type of the service and its
14   // name.
15   ros::ServiceClient spawnClient
16     = nh.serviceClient<turtlesim::Spawn>("spawn");
```

```
18    // Create the request and response objects.
19    turtlesim::Spawn::Request req;
20    turtlesim::Spawn::Response resp;
21
22    // Fill in the request data members.
23    req.x = 2;
24    req.y = 3;
25    req.theta = M_PI / 2;
26    req.name = "Leo";
27
28    // Actually call the service.  This won't return until
29    // the service is complete.
30    bool success = spawnClient.call(req, resp);
31
32    // Check for success and use the response.
33    if(success) {
34      ROS_INFO_STREAM("Spawned␣a␣turtle␣named␣"
35        << resp.name);
36    } else {
37      ROS_ERROR_STREAM("Failed␣to␣spawn.");
38    }
39
40  }
```

Programa cliente

- Assim como no caso das mensagens, temos que incluir os arquivos que definem os tipos de dados de serviços que vamos utilizar:

```
#include <turtlesim/Spawn.h>
```

- Criamos um objeto cliente e inicializamos com a função `serviceClient` do `nodeHandle`:

```
ros::ServiceClient client
    = node-handle.serviceClient<tipo-de-dados>(nome-do-serviço);
```

Programa cliente

- Agora criamos os objetos para guardar os dados da requisição e da resposta:

```
turtlesim::Spawn::request req;
turtlesim::Spawn::response resp;
```

- Esses dois objetos contém os campos relativos à requisição e à resposta respectivamente.

Programa cliente

- Para efetivamente chamar o serviço:

```
bool success = cliente.call(requisição, resposta);
```

- Esse método faz todo o trabalho de localizar o serviço desejado, enviar a requisição para o servidor e armazenar a resposta recebida.
- O método retorna um boolean verdadeiro ou falso indicando se o serviço foi executado com sucesso ou não.

Programa cliente

- Vamos compilar e executar o programa. Adicionar ao final do arquivo `CMakeLists.txt`:

```
add_executable(spawn_turtle src/spawn_turtle.cpp)
target_link_libraries(spawn_turtle ${catkin_LIBRARIES})
```

- Executar no terminal:

```
cd <caminho-para-workspace>
catkin_make
rosrun simuladores spawn_turtle
```

Programa servidor

- Agora vamos criar um programa que age como servidor, ou seja, oferece um serviço que pode ser chamado por outros nós.
- Na pasta `src` do pacote `simuladores`, criar um arquivo chamado `pubvel_toggle.cpp`.
- O objetivo do nosso programa é alternar entre girar a tartaruga ou movê-la para frente cada vez que o serviço for chamado.

```
1  // This program toggles between rotatation and translation
2  // commands, based on calls to a service.
3  #include <ros/ros.h>
4  #include <std_srvs/Empty.h>
5  #include <geometry_msgs/Twist.h>
6
7  bool forward = true;
8  bool toggleForward(
9    std_srvs::Empty::Request &req,
10   std_srvs::Empty::Response &resp
11 ) {
12   forward = !forward;
13   ROS_INFO_STREAM("Now␣sending␣" << (forward ?
14     "forward" : "rotate") << "␣commands.");
15   return true;
16 }
17
18 int main(int argc, char **argv) {
19   ros::init(argc, argv, "pubvel_toggle");
20   ros::NodeHandle nh;
21
```

```
22    // Register our service with the master.
23    ros::ServiceServer server = nh.advertiseService(
24      "toggle_forward", &toggleForward);
25
26    // Publish commands, using the latest value for forward,
27    // until the node shuts down.
28    ros::Publisher pub = nh.advertise<geometry_msgs::Twist>(
29      "turtle1/cmd_vel", 1000);
30    ros::Rate rate(2);
31    while(ros::ok()) {
32      geometry_msgs::Twist msg;
33      msg.linear.x = forward ? 1.0 : 0.0;
34      msg.angular.z = forward ? 0.0 : 1.0;
35      pub.publish(msg);
36      ros::spinOnce();
37      rate.sleep();
38    }
39  }
```

Programa servidor

- Assim como no caso dos nós subscribers, no servidor precisamos definir uma função callback. Essa função será chamada automaticamente pelo ROS sempre que o serviço for chamado.

```
Bool nome_da_função(
    nome_do_pacote::tipo_do_serviço::Request &req,
    nome_do_pacote::tipo_do_serviço::Response &req )
{
    …
}
```

Programa servidor

- Os dois parâmetros que a função recebe são os dados contidos na requisição e um objeto para armazenar os dados que serão enviados de volta na resposta.
- O trabalho da função é processar os dados recebidos na requisição e preencher o objeto resposta.
- No nosso exemplo ambas requisição e resposta são vazias. O objetivo é apenas variar o tipo de mensagens de comando que serão publicadas para a tartaruga cada vez que o serviço for chamado.
- A função callback deve retornar verdadeiro para indicar sucesso ou false para indicar falha na execução.

Programa servidor

- Para criar um objeto servidor:

```
ros::ServiceServer server = node_handle.advertiseService(
    nome_do_serviço,
    ponteiro_para_função_callback
)
```

- No fim, assim como no subscriber, devemos passar o controle para o ROS através de um `ros::spin()` ou `ros::spinOnce()`.

Programa servidor

- Vamos compilar e executar o programa. Adicionar ao final do arquivo `CMakeLists.txt`:

```
add_executable(pubvel_toggle src/pubvel_toggle.cpp)
target_link_libraries(pubvel_toggle ${catkin_LIBRARIES})
```

- Executar no terminal:

```
cd <caminho-para-workspace>
catkin_make
rosrun simuladores pubvel_toggle
```

Programa servidor

- Para testar o programa, chamar o serviço no terminal:

```
rosservice call /toggle_forward
```

Mavros

Mavros

- Quando trabalhamos com quadrotores reais existe mais um fator que precisamos levar em conta: a inteface entre o ROS e a placa controladora.
- Existem vários modelos de de placas controladoras. A mais popular é a Pixhawk.
- A Pixhawk se comunica com outros dispositivos através de um protocolo chamado MAVLink.
- Com esse protocolo ela é capaz de enviar mensagens contendo dados de vôo e receber mensagens com comandos e novos valores para parâmetros.

Mavros

- Para podermos interagir com a Pixhawk usando o ROS, precisamos de uma forma de converter as mensagens MAVLink em mensagens do ROS e vice-versa.
- O mavros é responsável por fazer esse trabalho.
- Para instalar o mavros:

```
sudo apt-get install ros-distribuição-mavros
sudo apt-get install ros-distribuição-mavros-msgs
```

Iniciando o Mavros

- O primeiro passo é conectar a Pixhawk ao computador. Em geral isso é feito através de um dispositivo chamado *telemetry module*, que cria uma comunicação sem fio através de ondas de rádio.

Iniciando o Mavros

- Quando a comunicação for estabelecida o sistema definirá um endereço representando o dispositivo. É necessário sabermos esse endereço para podermos iniciar o mavros.
- O endereço, e como obtê-lo, dependerá do sistema.
- No Ubuntu isso é bem simples. Após conectar o telemetry module, executar o comando:

```
ls /dev/serial/by-id
```

- Copiar o nome que foi retornado. O endereço completo é:

```
/dev/serial/by-id/<texto_copiado>
```

Iniciando o Mavros

- Para iniciar o mavros é só executar o nó `mavros_node`.
- Com o roscore já rodando, executar o comando:

```
rosrun mavros mavros_node fcu_url:=endereço_da_pixhawk
```

Iniciando o Mavros

- Para facilitar esse processo é possível criar um arquivo launch configurado para iniciar o `mavros_node` com todos os parâmetros pré-definidos.
- Na pasta `launch` do pacote `simuladores`, criar um arquivo chamado `start_mavros.launch` com o seguinte texto:

```
<launch>
    <arg name="pluginlists_yaml"
         default="$(find mavros)/launch/apm_pluginlists.yaml" />
    <arg name="config_yaml"
         default="$(find mavros)/launch/apm_config.yaml" />

    <node pkg="mavros" type="mavros_node" name="mavros"
          required="true" clear_params="true" >
        <param name="fcu_url" value="endereço_da_pixhawk" />
        <param name="gcs_url" value="" />
        <param name="target_system_id" value="1" />
        <param name="target_component_id" value="1" />

        <rosparam command="load" file="$(arg pluginlists_yaml)" />
        <rosparam command="load" file="$(arg config_yaml)" />
    </node>
</launch>
```

Iniciando o Mavros

- Para executar esse arquivo launch:

```
roslaunch simuladores start_mavros.launch
```

Tópicos do Mavros

- Uma vez que o nó tenha sido iniciado e a comunicação tenha sido estabelecida, o mavros vai criar vários tópicos.
- De forma parecida com o simulador, cada tópico contém mensagens de um determinado sensor ou espera algum tipo de comando.
- Também serão criados diversos serviços, que podem ser acessados para fazer tarefas específicas como decolar, pousar, trocar o modo de vôo, etc.

Tópicos do Mavros

IMU (Inertial Measurement Unit)

- O mavros publica mensagens do tipo `sensor_msgs/Imu` no tópico `/mavros/imu/data`.

Tópicos do Mavros

GPS:

- O mavros publica mensagens do tipo `sensor_msgs/NavSatFix` no tópico `/mavros/global_position/global`.
- Também publica a velocidade estimada pelo GPS como mensagens do tipo `geometry_msgs/Vector3Stamped` no tópico `/mavros/global_position/gp_vel`.

Tópicos do Mavros

Bússola:

- O mavros publica a medida do campo gravitacional da Terra como mensagens do tipo `sensor_msgs/MagneticField`no tópico `/mavros/imu/mag`.

Tópicos do Mavros

Barômetro:

- O barômetro mede a pressão atmosférica, utilizada para estimar a altitude do quadrotor.
- O mavros publica as medidas como mensagens do tipo `sensor_msgs/FluidPressure` no tópico `/mavros/imu/atm_pressure`.
- A temperatura também é estimada e publicada como mensagens do tipo `sensor_msgs/Temperature` no tópico `/mavros/imu/temperature`.

Tópicos do Mavros

Estimador de estados:

- A Pixhawk utiliza um filtro de Kalman internamente para fundir as medidas dos sensores e estimar a posição, orientação e velocidade do quadrotor em um sistema de coordenadas cartesianas cuja origem é no ponto de onde o quadrotor decolou.
- A pose (posição + orientação) estimada é publicada pelo mavros no tópico `/mavros/local_position/odom`como mensagens do tipo `nav_msgs/Odometry`.
- A velocidade é publicada no tópico `/mavros/local_position/velocity` como mensagens de tipo `geometry_msgs/TwistStamped`

Controle

- Há duas formas de controlar a Pixhawk através do mavros.
- A primeira é simular um piloto humano utilizando um radio controle. O tópico `/mavros/rc/override` aceita mensagens do tipo `mavros_msgs/OverrideRCIn`. Essa mensagem contém campos que representam os comandos gerados por um radio controle manual.
- A segunda forma é utilizando mensagens de comando de velocidade, as mesmas usadas no simulador. O mavros aceita mensagens do tipo `geometry_msgs/Twist` no tópico `/mavros/setpoint_velocity/cmd_vel`.