Robot Operating System

Universidade Federal de Pernambuco
Adrien Durand-Petiteville
adrien.durandpetiteville@ufpe.br

Robot Operating System

- O Robot Operating System (ROS) é uma estrutura para escrever software de robô.
- É uma coleção de **ferramentas**, **bibliotecas** e **convenções** que visam simplificar a tarefa de criar comportamento robótico **complexo** e **robusto** em uma ampla **variedade** de plataformas robóticas.

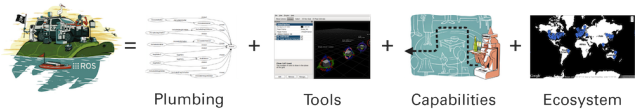
Exemplo - 1

- Considere uma tarefa simples de ”buscar um item”, na qual um robô é instruído a recuperar um grampeador.
- Primeiro, o robô deve entender a solicitação, verbalmente ou através de alguma outra modalidade, como interface da web, email ou até SMS.
- Em seguida, o robô deve iniciar algum tipo de planejador para coordenar a pesquisa do item, o que provavelmente exigirá a navegação por várias salas de um edifício, talvez incluindo elevadores e portas.
- Ao chegar a uma sala, o robô deve procurar em mesas repletas de objetos de tamanho semelhante (já que todos os objetos portáteis têm aproximadamente o mesmo tamanho) e encontrar um grampeador.
- O robô deve então refazer suas etapas e entregar o grampeador no local desejado.
- Cada um desses subproblemas pode ter números arbitrários de fatores complicadores.

Exemplo - 2

- ROS foi construído desde o início para incentivar o desenvolvimento de software de robótica colaborativa.
  - Uma organização pode ter especialistas em mapeamento de ambientes internos e contribuir com um sistema complexo e fácil de usar para a produção de mapas internos.
  - Outro grupo pode ter experiência no uso de mapas para navegar de maneira robusta em ambientes internos.
  - Ainda outro grupo pode ter descoberto uma abordagem de visão computacional específica que funciona bem para reconhecer pequenos objetos desorganizados.
- ROS inclui muitos recursos projetados especificamente para simplificar esse tipo de colaboração em larga escala.

Filosofia - 1



- Peer To Peer
  - Os sistemas ROS consistem em vários pequenos programas de computador que se conectam e trocam mensagens continuamente.
  - Essas mensagens viajam diretamente de um programa para outro; não há serviço de roteamento central.
- Baseado em ferramentas
  - Sistemas de software complexos podem ser criados a partir de muitos programas pequenos e genéricos.
  - As tarefas são executadas por programas separados:
    - Navegar na árvore do código-fonte
    - Visualizar as interconexões do sistema
    - Plotar graficamente fluxos de dados
    - Gerar documentação
    - Registrar dados

Filosofia - 2

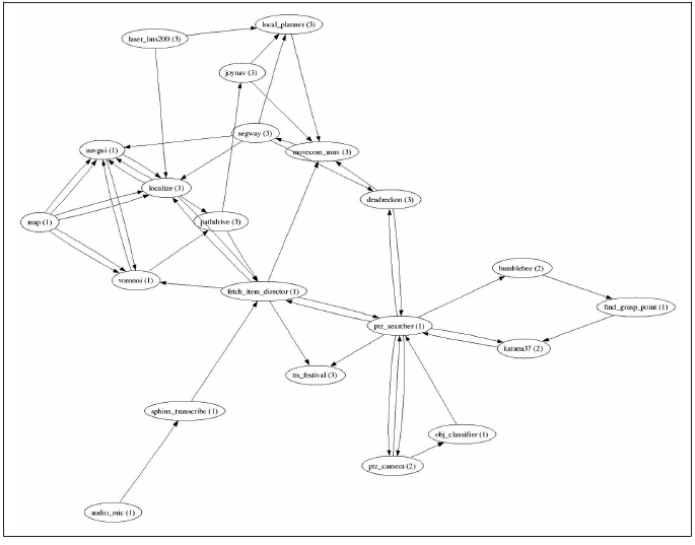
- Multilíngue
  - Muitas tarefas de software são mais fáceis de realizar em linguagens de script de "alta produtividade", como Python ou Ruby.
  - No entanto, há momentos em que os requisitos de desempenho determinam o uso de linguagens mais rápidas, como C++.
  - Os módulos do software ROS podem ser escritos em qualquer idioma para o qual uma biblioteca cliente foi gravada.
  - No momento existem bibliotecas cliente para C++, Python, LISP, Java, JavaScript, MATLAB, Ruby, Haskell, R, Julia...
- Fino
  - As convenções de ROS incentivam os colaboradores a criar bibliotecas independentes e, em seguida, agrupar essas bibliotecas para que possam enviar e receber mensagens para e de outros módulos do ROS.
  - Essa camada extra destina-se a permitir a reutilização de software fora do ROS para outras aplicações
- Fonte livre e aberta
  - O núcleo do ROS é liberado sob a licença permissiva BSD, que permite o uso comercial e não comercial.

Conceitos Chaves

ROS Graph - 1

- Um sistema ROS é composto de muitos programas diferentes, executando simultaneamente e se comunicando, transmitindo mensagens.
- É conveniente usar um gráfico matemático para representar essa coleção de programas e mensagens:
  - Os programas são os nós do gráfico.
  - Os programas que se comunicam entre si são conectados por arestas.
  - Um nó representa um módulo de software que está enviando ou recebendo mensagens.
  - Uma aresta representa um fluxo de mensagens entre dois nós.
- O maior benefício de uma arquitetura baseada em gráficos é a capacidade de prototipar sistemas complexos com pouca ou nenhuma "cola"de software necessária para a experimentação.

ROS Graph - 2


local_planner (3)
laser_lms200 (3)
joynav (3)
segway (3)
navgui (1)
wavefront_stm (3)
localize (3)
deadreckon (3)
map (1)
pathdrive (3)
voronoi (1)
fetch_item_director (1)
bumblebee (2)
find_grasp_point (1)
ptz_searcher (1)
katana37 (2)
tts_festival (3)
sphinx_transcribe (1)
obj_classifier (1)
ptz_camera (2)
audio_mic (1)

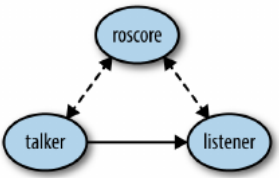
roscore - 1

- O roscore é um serviço que fornece informações de conexão aos nós para que eles possam transmitir mensagens entre si.
- Cada nó se conecta ao roscore na inicialização para registrar detalhes dos fluxos de mensagens que publica e dos fluxos nos quais deseja assinar.
- Quando um novo nó aparece, o roscore fornece as informações necessárias para formar uma conexão direta ponto a ponto com outros nós que publicam e assinam os mesmos tópicos de mensagem.
- Todo sistema ROS precisa de um roscore em execução, pois sem ele, os nós não podem encontrar outros nós.

roscore - 2

- Quando um nó ROS é iniciado, ele espera que seu processo tenha uma variável de ambiente chamada ROS_MASTER_URI.
- Com o conhecimento da localização do roscore na rede, os nós se registram na inicialização com o roscore e consultam o roscore para encontrar outros nós e fluxos de dados por nome.
- Cada nó do ROS informa ao roscore quais mensagens ele fornece e quais gostaria de assinar.
- O roscore fornece os endereços dos produtores e consumidores de mensagens relevantes.

- Visualizados em forma de gráfico, cada nó no gráfico pode periodicamente chamar os serviços fornecidos pela roscore para encontrar seus pares (linhas tracejadas).



- O roscore também fornece um servidor de parâmetros, usado extensivamente pelos nós do ROS para configuração.
- O servidor de parâmetros permite que os nós armazenem e recuperem estruturas de dados arbitrárias, como descrições de robôs, parâmetros para algoritmos e assim por diante.

catkin

- catkin é o sistema de criação do ROS: o conjunto de ferramentas que o ROS usa para gerar programas executáveis, bibliotecas, scripts e interfaces que outro código pode usar.
- O catkin inclui um conjunto de macros do CMake e scripts Python personalizados para fornecer funcionalidade extra sobre o fluxo de trabalho normal do CMake.
- Existem dois arquivos, *CMakeLists.txt* e *package.xml*, aos quais vocês precisam adicionar algumas informações específicas para que as coisas funcionem corretamente.

Workspaces - 1

- Antes de começar a escrever qualquer código ROS, é necessário configurar uma área de trabalho para a permanência desse código.
- Uma área de trabalho é simplesmente um conjunto de diretórios nos quais vive um conjunto relacionado de códigos ROS.
- Vocês podem ter vários espaços de trabalho do ROS, mas só pode trabalhar em um deles a qualquer momento.
- A maneira mais simples de pensar sobre isso é que vocês só podem ver o código que vive no seu espaço de trabalho atual.

Workspaces - 2

- Como criar um espaço de trabalho de catkin e inicializá-lo:

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/src
$ catkin_init_workspace
```

- Isso cria um diretório da área de trabalho chamado catkin_ws (embora você possa chamá-lo como quiser), com um diretório src dentro para o seu código.
- O comando `catkin_init_workspace` cria um arquivo *CMakeLists.txt* no diretório src.

Workspaces - 3

- Como criar alguns outros arquivos da área de trabalho:

```
$ cd ~/catkin_ws
$ catkin_make
```

- A execução de catkin_make gerará muita saída.
- Quando terminar, você terminará com dois novos diretórios: *build* e *devel*.
  - *build* é onde o catkin armazena os resultados de alguns de seus trabalhos, como bibliotecas e programas executáveis (C++).
  - O *devel* contém vários arquivos e diretórios, sendo os mais interessantes os arquivos de instalação. A execução dessas configurações configura o sistema para usar esse espaço de trabalho e o código contido nele.

```
$ source devel/setup.bash
```

- Para cada novo terminal, você precisa originar o arquivo setup.bash para o espaço de trabalho com o qual deseja trabalhar.

ROS Packages - 1

- O software ROS é organizado em pacotes, cada um dos quais contém alguma combinação de código, dados e documentação.
- O ecossistema ROS inclui milhares de pacotes publicamente disponíveis em repositórios abertos.
- Os pacotes ficam dentro das áreas de trabalho, no diretório src.
- Cada diretório do pacote deve incluir um arquivo *CMakeLists.txt* e um arquivo *package.xml* que descreve o conteúdo do pacote e como o catkin deve interagir com ele.

- Como criar um novo pacote:

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg my_awesome_code rospy
```

- `catkin_create_pkg` cria o novo pacote chamado my_awesome_code, que depende do pacote rospy.
- Se o novo pacote depender de outros pacotes existentes, liste-os na linha de comando.
- O comando `catkin_create_pkg` cria um diretório com o mesmo nome que o novo pacote com um arquivo `CMakeLists.txt`, um arquivo `package.xml` e um diretório src.
- Depois de criar um pacote, vocês podem colocar seus nós Python no diretório src.

Example of package.xml

Ferramentas do sistema de arquivos

- **rospack** permite que você obtenha informações sobre pacotes

```
$ rospack find [package_name]
```

- **roscd** permite que você mude diretamente para o diretório de um pacote

```
$ roscd <package>[/subdir]
```

- **rosls** permite que você ls diretamente em um pacote por nome

```
$ rosls <package>[/subdir]
```

- `rosrun` procura em um pacote o programa solicitado e passa a ele todos os parâmetros fornecidos na linha de comando.
- A sintaxe é a seguinte:

```
$ rosrun PACKAGE EXECUTABLE [ARGS]
```

Exemplo - 1

- Por exemplo, queremos executar o programa **talker** do pacote **rospy_tutorials** localizado em */opt/ros/melodic/share/rospy_tutorials*.
- Em terminal 1, inicie uma instância do roscore

```
$ roscore
```

- Em terminal 2, inicie uma instância do ROS graph

```
$ rqt_graph
```

- Em terminal 3, execute:

```
$ rosrun rospy_tutorials talker
```

- Atualize o ROS graph.

Exemplo - 2

- Em terminal 4, execute:

```
$ rostopic list
$ rostopic echo [completar]
$ rostopic info [completar]
```

- Em terminal 5, execute:

```
$ rosrun rospy_tutorials listener
```

- Atualize o ROS graph.
- Em terminal 4, execute:

```
$ rostopic info [completar]
```

Node

Criação de um nó

- Os sistemas ROS consistem em vários nós independentes que compõem um gráfico.
- Criação de um nó:
  - `cd workspace/src/package/src/`: mover para o diretório do package
  - Criar um arquivo `arquivoNo.py`
  - `sudo chmod +x arquivoNo.py`: autorizar a execução do arquivo
- **Exemplo**: `01_node.py`

```
1 #! /usr/bin/python3
2
3 # Importação do modulo ROS para Python
4 import rospy
5
6 # Criação do nó com o nome hello_world
7 rospy.init_node('hello_world')
8
9 print("Hello world")
```

Criação de um nó- 2

```
$ #!/usr/bin/python
```

- É conhecido como o shebang. Ele permite que o sistema operacional saiba que esse é um arquivo Python e que deve ser passado para o intérprete Python.

```
$ import rospy
```

- Aparece em todos os nós do ROS Python e importa toda a funcionalidade básica

```
$ rospy.init_node('hello_world')
```

- Inicialize o nó.

```python
#! /usr/bin/python

import rospy

rospy.init_node('hello_world')

# Definição da frequência do laço while
rate = rospy.Rate(2)

count = 0
# Em loop até a detecção de Ctrl+c
while not rospy.is_shutdown():
    print("Hello world number {}".format(count))
    count += 1

    # Esperar pelo fim do tempo do laço
    rate.sleep()
```

Topics

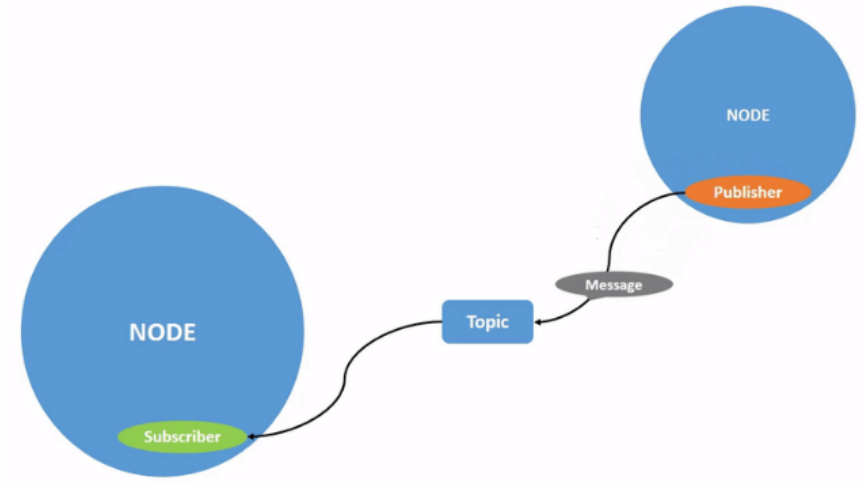
Topics

- Nós se comunicam, trocando informações e dados.
- A maneira mais comum de fazer isso é através de tópicos (topics).
- **Um tópico é um nome para um fluxo de mensagens com um tipo definido.**
- Por exemplo, os dados de uma câmera podem ser enviados sobre um tópico chamado *image_topics*, com um tipo de mensagem *Image*.

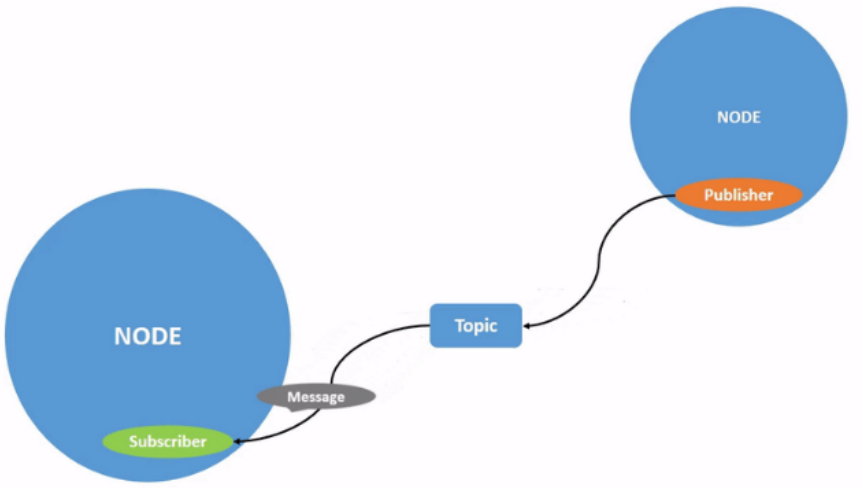
publish/subscribe

- Os tópicos implementam um mecanismo de comunicação de publicação / assinatura (publish/subscribe).
- Antes de os nós começarem a transmitir dados sobre tópicos, eles devem primeiro anunciar, ou *advertise*, o nome do tópico e os tipos de mensagens que serão enviadas.
- Em seguida, eles podem começar a enviar, ou *publish*, os dados reais sobre o tópico.
- Os nós que desejam receber mensagens em um tópico podem se inscrever, ou *subscribe*, nesse tópico, solicitando o `roscore`.
- Após a assinatura, todas as mensagens no tópico são entregues no nó que fez a solicitação.

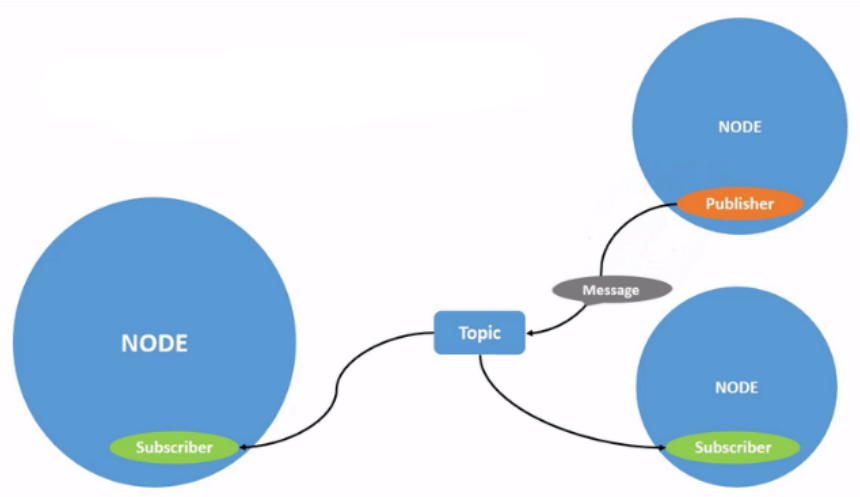
Publisher



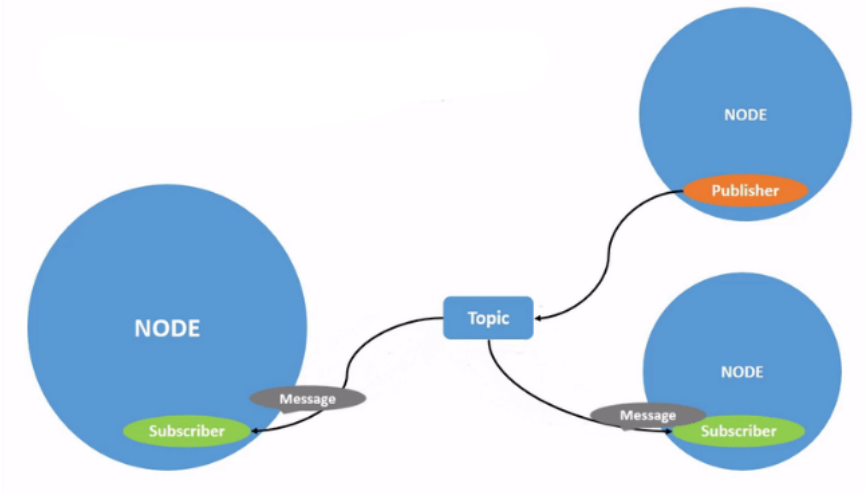
Subscriber



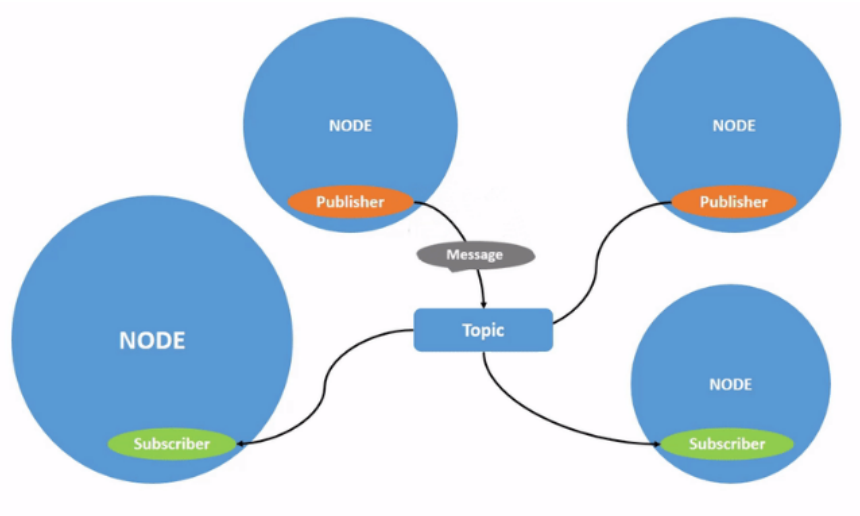
1 publisher - 2 subscribers



1 publisher - 2 subscribers



2 publishers - 2 subscribers



Publisher

```
1 #! /usr/bin/python
2
3 import rospy
4 # Importação do modulo de mensagens padrões
5 from std_msgs.msg import Int32
6
7 # Criação do nó com o nome simple_publisher
8 rospy.init_node('simple_publisher')
9
10 # Criação do publisher no topic counter de um mensagem de tipo
      Int32
11 pub = rospy.Publisher('counter', Int32, queue_size=1)
```

```
1  rate = rospy.Rate(2)
2
3  count = 0
4
5  while not rospy.is_shutdown():
6      # Publicação da mensagem no topico
7      pub.publish(count)
8
9      count += 1
10     rate.sleep()
```

Publicar um tópico - 1

```
$ from std_msgs.msg import Int32
```

- Importa a definição da mensagem que vamos enviar sobre o tópico.
- Aqui, usamos um número inteiro de 32 bits, definido no pacote de mensagens padrão do ROS, std_msgs.
- Para que a importação funcione conforme o esperado, precisamos importar do `<nome do pacote>.msg`, pois é aqui que as definições do pacote são armazenadas.
- Como estamos usando uma mensagem de outro pacote, precisamos informar o sistema de criação do ROS sobre isso adicionando uma dependência ao nosso arquivo package.xml.

```
$ <depend package="std_msgs"/>
```

- Sem essa dependência, o ROS não saberá onde encontrar a definição da mensagem.

Publicar um tópico - 2

```
$ pub = rospy.Publisher('counter', Int32)
```

- Anuncia o nó com um publicador.
- Fornece um nome ao tópico (counter) e especifica o tipo de mensagem que será enviada por ele (Int32).
- Neste ponto, o tópico é anunciado e está disponível para outros nós se inscreverem.

Publicar um tópico - 3

```
$ count = 0
$ while not rospy.is_shutdown():
$     pub.publish(count)
$     count += 1
$     rate.sleep()
```

- Primeiro, definimos a taxa, em hertz, na qual queremos publicar.
- A função `is_shutdown()` retornará True se o está pronto para ser desligado e False caso contrário, para que possamos usá-lo para determinar se é hora de sair do loop `while`.
- Dentro do loop `while`, publicamos o valor atual do contador, aumentamos seu valor em 1 e depois dormimos um pouco.
- A chamada para `rate.sleep()` durará o suficiente para garantir que executemos o corpo do loop `while` em aproximadamente 2 Hz.

Verificar se tudo funciona como esperado

```
$ rostopic -h
```

```
$ rostopic list
```

```
$  rosrun package file.py
```

```
$ rostopic list
```

Verificar se tudo funciona como esperado - 2

```
$  rostopic echo topic -n 5
```

```
$ rostopic hz topic
```

```
$ rostopic info topic
```

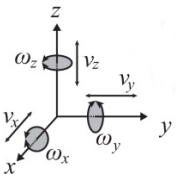
```
$ rostopic find std_msgs/Int32
```

Tipo de mensagem

ROS type	Serialization	C++ type	Python type	Notes
`bool`	Unsigned 8-bit integer	`uint8_t`	`bool`	
`int8`	Signed 8-bit integer	`int8_t`	`int`	
`uint8`	Unsigned 8-bit integer	`uint8_t`	`int`	`uint8[]` is treated as a `string` in Python
`int16`	Signed 16-bit integer	`int16_t`	`int`	
`uint16`	Unsigned 16-bit integer	`uint16_t`	`int`	
`int32`	Signed 32-bit integer	`int32_t`	`int`	
`uint32`	Unsigned 32-bit integer	`uint32_t`	`int`	
`int64`	Signed 64-bit integer	`int64_t`	`long`	
`uint64`	Unsigned 64-bit integer	`uint64_t`	`long`	
`float32`	32-bit IEEE float	`float`	`float`	
`float64`	64-bit IEEE float	`double`	`float`	
`string`	ASCII string	`std::string`	`string`	ROS does not support Unicode strings; use a UTF-8 encoding
`time`	secs/nsecs unsigned 32-bit ints	`ros::Time`	`rospy.Time`	`duration`

Controle de base movel

A mensagem **TWIST**



- Para controlar um corpo com 6 graus de liberdade usamos a mensagem padrão **TWIST**
  `http://docs.ros.org/en/melodic/api/geometry_msgs/html/msg/Twist.html`
- **TWIST** é uma estrutura com 6 campos:
  - v_x: TWIST.linear.x
  - v_y: TWIST.linear.y
  - v_z: TWIST.linear.z
  - w_x: TWIST.angular.x
  - w_y: TWIST.angular.y
  - w_z: TWIST.angular.z

Controlar o robô TurtleBot

- No terminal 1

```
$ roscore
```

- No terminal 2

```
$ rosrun turtlesim turtlesim_node
```

- No terminal 3 (opcional)

```
$ rosrun turtlesim turtle_teleop_key
```

```
1  #! /usr/bin/python
2
3  import rospy
4  # Importação da biblioteca de mensagens de geometria
5  from geometry_msgs.msg import Twist
6
7  rospy.init_node('simple_publisher')
8
9  # Criação do publisher no topic /turtle/cmd_vel de uma
       mensagem de tipo Twist
10 pub = rospy.Publisher('/turtle1/cmd_vel', Twist, queue_size=1)
11
12 cmd = Twist()
```

```
rate = rospy.Rate(1)

count = 0
while not rospy.is_shutdown():

    if count % 2 == 0:
        cmd.linear.x = 1
        cmd.angular.z = 0
    else:
        cmd.linear.x = 0
        cmd.angular.z = 1

    pub.publish(cmd)
    count += 1
    rate.sleep()
```

Controlar o robô TIAGo

- No workspace do Tiago

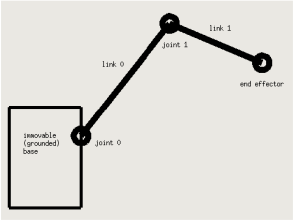
```
$ roslaunch tiago_gazebo tiago_gazebo.launch public_sim:=true
end_effector:=pal-gripper
```

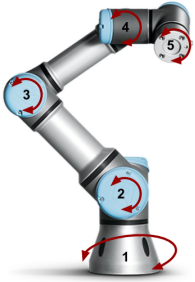
```
1  #! /usr/bin/python3
2
3  import rospy
4  from geometry_msgs.msg import Twist
5
6
7  rospy.init_node('simple_publisher')
8
9  # Criação do publisher no topic /mobile_base_controller/
       cmd_vel de uma mensagem de tipo Twist
10 pub = rospy.Publisher('/mobile_base_controller/cmd_vel', Twist
       , queue_size=1)
11
12 cmd = Twist()
```

```
rate = rospy.Rate(1)

count = 0
while not rospy.is_shutdown():

    if count % 2 == 0:
        cmd.linear.x = 1
        cmd.angular.z = 0
    else:
        cmd.linear.x = 0
        cmd.angular.z = 1

    pub.publish(cmd)
    count += 1
    rate.sleep()
```

Controle de braço robótico

Manipuladores





- Manipuladores robóticos são um conjunto de juntas mantidas juntas por uma estrutura de algum tipo.
  - As juntas rotativas giram em torno de um eixo de rotação.
  - As juntas prismáticas movem-se linearmente ao longo de um eixo de movimento.
- Um link é uma seção de um braço de robô conectada por uma junta.

A mensagem **JointTrajectory**

- Para controlar um conjunto de juntas usamos a mensagem padrão **JointTrajectory** `http://docs.ros.org/en/melodic/api/trajectory_msgs/html/msg/JointTrajectory.html`
- **JointTrajectory** é uma estrutura com 3 campos:
  - `std_msgs/Header` **header**
  - `string[]` **joint_names**
  - `trajectory_msgs/JointTrajectoryPoint[]` **points**
- O nome das juntas deve corresponder ao nome no controlador (usar rostopic list & echo para encontrar)

A mensagem **std_msgs/Header**

- Para controlar um conjunto de juntas usamos a mensagem padrão **Header**
  `http://docs.ros.org/en/melodic/api/std_msgs/html/msg/Header.html`
- **Header** é uma estrutura com 3 campos:
  - `uint32` **seq**
  - `time` **stamp**
  - `string` **frame_id**

```
# Standard metadata for higher-level stamped data types.
# This is generally used to communicate timestamped data
# in a particular coordinate frame.
#
# sequence ID: consecutively increasing ID
uint32 seq
#Two-integer timestamp that is expressed as:
# * stamp.sec: seconds (stamp_secs) since epoch (in Python the variable is called 'secs')
# * stamp.nsec: nanoseconds since stamp_secs (in Python the variable is called 'nsecs')
# time-handling sugar is provided by the client library
time stamp
#Frame this data is associated with
string frame_id
```

A mensagem **trajectory_msgs/JointTrajectoryPoint[]**

- Para criar uma trajetória usamos a mensagem padrão **JointTrajectoryPoint[]** `http://docs.ros.org/en/melodic/api/trajectory_msgs/html/msg/JointTrajectoryPoint.html`
- **JointTrajectoryPoint[]** é uma estrutura com 5 campos:
  - `float64[]` **positions**
  - `float64[]` **velocities**
  - `float64[]` **accelerations**
  - `float64[]` **effort**
  - `duration` **time_from_start**
- Apenas um dos campos pode ser usado por vez (**positions**, **velocities**, **acceleration** ou **effort**).
- O campo **time_from_start** é obrigatório.

```
#! /usr/bin/python

import rospy
# Importação da biblioteca de mensagens de trajetoria
from trajectory_msgs.msg import JointTrajectory
from trajectory_msgs.msg import JointTrajectoryPoint

rospy.init_node('move_arm')

# Criação do publisher no topic /arm_controller/command
# de uma mensagem de tipo JointTrajectory
pub = rospy.Publisher('/arm_controller/command',
    JointTrajectory, queue_size=1)
```

```
cmd = JointTrajectory()

cmd.joint_names.append("arm_1_joint")
cmd.joint_names.append("arm_2_joint")
cmd.joint_names.append("arm_3_joint")
cmd.joint_names.append("arm_4_joint")
cmd.joint_names.append("arm_5_joint")
cmd.joint_names.append("arm_6_joint")
cmd.joint_names.append("arm_7_joint")

point = JointTrajectoryPoint()
point.positions = [0] * 7
point.time_from_start = rospy.Duration(1)

cmd.points.append(point)
```

```
rate = rospy.Rate(1)

angle = 0.1

while not rospy.is_shutdown():

    cmd.points[0].positions[1] = angle
    cmd.points[0].time_from_start = rospy.Duration(1)

    pub.publish(cmd)
    angle += 0.1
    rate.sleep()
```

Subscriber

```
1 #! /usr/bin/python3
2
3 import rospy
4 from std_msgs.msg import Int32
5
6
7 # Define the function called by the subscriber
8 def callback(msg):
9     print (msg.data)
10
11 rospy.init_node('simple_subscriber')
12
13 # Define the new subsriber
14 rospy.Subscriber('counter', Int32, callback)
15
16 # Equivalent to an infinite while to not close the program
17 rospy.spin()
```

Inscrevendo-se em um tópico - 1

- A primeira parte interessante desse código é o callback (retorno de chamada) que lida com as mensagens quando elas chegam.

```
$ def callback(msg):
$     print msg.data
```

- O ROS é um sistema orientado a eventos e utiliza muito as funções de callback.
- Depois que um nó se inscreve em um tópico, toda vez que uma mensagem chega, a função de callback associada é chamada, com a mensagem como parâmetro.

Inscrevendo-se em um tópico - 2

- Após inicializar o nó, como antes, assinamos o tópico do contador:

```
$ sub = rospy.Subscriber('counter', Int32, callback)
```

- Fornecemos o nome do tópico, o tipo de mensagem do tópico e o nome da função de callback.
- Se o tópico não existir, ou se o tipo estiver errado, não haverá mensagens de erro: o nó simplesmente aguardará até que as mensagens comecem a ser publicadas no tópico.
- Depois que a assinatura é feita, damos controle ao ROS chamando `rospy.spin()`.
- Esta função retornará apenas quando o nó estiver pronto para desligar.

Verificar se tudo funciona como esperado

- Primeiro, verifique se o nó do editor ainda está em execução e se ainda está publicando mensagens no tópico do contador.
- Em outro terminal, inicie o nó do assinante.

```
$ rosrun package file.py
```

```
$ rostopic info topic
```

- Também podemos publicar mensagens em um tópico na linha de comando usando o rostopic pub.

```
$ rostopic pub topic messageType valor
$ rostopic pub counter std_msgs/Int32 1000000
```

Problema com frequência de publicação

- O tempo de execução da função callback pode ser incompatível com a frequência de publicação

```
#! /usr/bin/python3

import rospy
import time
from std_msgs.msg import Int32

# Define the function called by the subscriber
def callback(msg):
    print msg.data
    time.sleep(1) # Simulate some processing

rospy.init_node('simple_subscriber')

# queue_size is used to read the last message
rospy.Subscriber('counter', Int32, callback, queue_size=1)

rospy.spin()
```

Subscriber com Programação Orientada a Objetos

- Precisamos de usar a programação orientada a objetos para implementar corretamente um subscriber

```
1 #! /usr/bin/python3
2
3 import rospy
4 from std_msgs.msg import Int32
```

```python
# Definition of the class
class mySub():

    def __init__(self):
        # Define the subscriber
        self.sub = rospy.Subscriber('counter', Int32, self.callback, queue_size=1)

        # Message variables
        self.counterValue = 0

    # Definition of the function called by the subscriber
    def callback(self, msg):
        self.counterValue = msg.data

    def printMsg(self):
        print(self.counterValue)
```

```
# Main program
if __name__ == '__main__':
    # Define the node
    rospy.init_node('simpleSubPOO')
    # Creaete an object of class mySub and run the init
    function
    subObj = mySub()
    # While ROS is running
    while not rospy.is_shutdown():
        # Call printMsg method of mySub class
        subObj.printMsg()
```

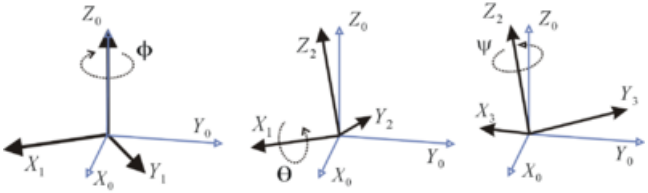
Odometria

Orientação de um corpo rígido

- Existem várias soluções para descrever a orientação de um corpo rígido:
  - Quaternion
  - Ângulos de Euler
  - Matriz de orientação
  - ...
- ROS dá orientação como um quatérnion
- Precisamos fazer a conversão Quaternion $\rightarrow$ Ângulos de Euler

Ângulos de Euler

- Esse método usa dois sistemas de coordenadas:
  - Um sistema inercial fixo
  - Um sistema que gira junto ao corpo em rotação
- Para especificar a orientação do corpo girante em relação ao sistema inercial (fixo) faz-se uso de três ângulos independentes:
  - Uma rotação ϕ em torno do eixo Z_0 que dá a base 1
  - Uma rotação θ em torno do eixo X_1 que dá a base 2
  - Uma rotação ψ em torno do eixo Z_2 que dá a base 3



A mensagem **nav_msgs/Odometry**

- Para compartilhar os dados de odometria usamos a mensagem padrão **Odometry** `http://docs.ros.org/en/noetic/api/nav_msgs/html/msg/Odometry.html`
- **Odometry** é uma estrutura com 4 campos e usamos principalmente os seguintes:
  - `geometry_msgs/PoseWithCovariance` **pose**
  - `geometry_msgs/TwistWithCovariance` **twist**
- **pose**: pose (posição e orientação) do corpo rígido
- **twist**: velocidade do corpo rígido

```
1 #! /usr/bin/python3
2
3 import rospy
4 from nav_msgs.msg import Odometry
5
6 # Definition of the class
7 class mySub():
8
9     def __init__(self):
10         # Define the subscriber
11         self.sub = rospy.Subscriber('/mobile_base_controller/
    odom', Odometry, self.callback, queue_size=1)
12
13     # Definition of the function called by the subscriber
14     def callback(self, msg):
15         #self.counterValue = msg.data
16         print(msg.pose)
```

```
# Main program
if __name__ == '__main__':
    # Define the node
    rospy.init_node('TIAGo_odem_node')
    # Create an object of class mySub and run the init
    function
    subObj = mySub()
    # While ROS is running
    rospy.spin()
```

TIAGo Odometry com angulos de Euler

- ROS usa os quaternions para representar a orientatção
- Pode ser necessário convertê-la em ângulos (Euler por exmplo)

```
#! /usr/bin/python3

import rospy
from nav_msgs.msg import Odometry
from tf.transformations import euler_from_quaternion,
    quaternion_from_euler

class mySub():
     def __init__(self):
         self.sub = rospy.Subscriber('/mobile_base_controller/
    odom', Odometry, self.callback, queue_size=1)

     def callback(self, msg):
         qtn = msg.pose.pose.orientation
         qtn_list = [qtn.x, qtn.y, qtn.z, qtn.w]
         (roll, pitch, yaw) = euler_from_quaternion (qtn_list)
         print(roll, pitch, yaw)
```

```
# Main program
if __name__ == '__main__':
    # Define the node
    rospy.init_node('TIAGo_odem_node')
    # Create an object of class mySub and run the init
    function
    subObj = mySub()
    # While ROS is running
    rospy.spin()
```

Laser

A mensagem **sensor_msgs/LaserScan**

- Para compartilhar os dados de um laser usamos a mensagem padrão **LaserScan**
  `http://docs.ros.org/en/noetic/api/sensor_msgs/html/msg/LaserScan.html`
- **LaserScan** é uma estrutura com 10 campos e usamos principalmente os seguintes:
  - `float32` **angle_min**: ângulo mínimo
  - `float32` **angle_max**: ângulo maximo
  - `float32` **angle_increment**: ângulo entre cada raio do laser
  - `float32[]` **ranges**: arranjo com as distâncias medidas por cada raio
- O número de raio é: (angle_max − angle_min)/angle_increment

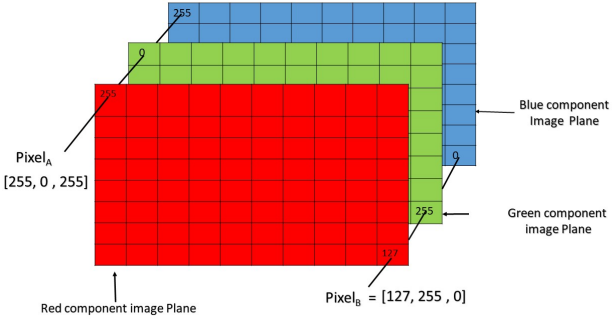
```
1 #! /usr/bin/python3
2
3 import rospy
4 from sensor_msgs.msg import LaserScan
5
6 class mySub():
7
8     def __init__(self):
9         self.sub = rospy.Subscriber('/scan_raw', LaserScan,
    self.callback, queue_size=1)
10
11     # Definition of the function called by the subscriber
12     def callback(self, msg):
13         #self.counterValue = msg.data
14         print(msg.ranges)
```

```
# Main program
if __name__ == '__main__':
    # Define the node
    rospy.init_node('TIAGo_laser_node')
    # Creaete an object of class mySub and run the init function
    subObj = mySub()
    # While ROS is running
    rospy.spin()
```

Imagem

Imagem digital

- Uma imagem digital é uma matrizes com 3 dimensões.
  - Primeira dimensão: largura da imagem
  - Segunda dimensão altura da imagem
  - Terceira dimensão: cores (vermelho, verde e azul)
- Cada pixel há um valor entre 0 e 255 (uint8) ou entre 0 e 1



Pixel of an RGB image are formed from the corresponding pixel of the three component images

- Nós sempre convertemos uma mensagem de imagem para um tipo opencv.

```
#!/usr/bin/env python3
import rospy
import cv2
from cv_bridge import CvBridge, CvBridgeError
from sensor_msgs.msg import Image

class myCamera():

    def __init__(self):
        print('init camera')
        # Bridge to convert ROS message to openCV
        self.bridge = CvBridge()

        # Subscriber to the camera image
        self.image_sub = rospy.Subscriber("/xtion/rgb/
    image_color",Image,self.imageCallBack)
```

```
1     def callback_SubscribeCamera(self, msg):
2         print('callback camera')
3         try:
4             self.cv_image = self.bridge.imgmsg_to_cv2(msg, "
      bgr8")
5         except CvBridgeError as e:
6             print(e)
7
8         # cv_image[linha][coluna][bgr] bgr-> 0:blue, 1:green,
      2:red
9         print(self.cv_image[0][0])
10        print(self.cv_image[0][0][0])
11
12        # Display the image
13        cv2.imshow("raw", self.cv_image)
14        cv2.waitKey(3)
```

Definir e usar um novo tipo de mensagem

Definir uma nova mensagem

- As mensagens ROS são definidas por arquivos especiais de definição de mensagens no diretório *msg* de um pacote.
- Esses arquivos são compilados em implementações específicas do idioma que podem ser usadas no seu código.
- Você precisa executar `catkin_make` se quiser definir seus próprios tipos de mensagens.
- Os arquivos de definição de mensagem geralmente são bastante simples e curtos.
- Cada linha especifica um tipo e um nome de campo.
- Os tipos podem ser tipos primitivos ROS internos, tipos de mensagens de outros pacotes, matrizes de tipos ou o tipo especial `Header`.

Definir uma nova mensagem - Exemplo - 1

- O arquivo *Complex.msg* está no diretório *msg* do pacote básico.
- Ele define dois valores, real e imaginário, ambos com o mesmo tipo `float32`.

```
$ float32 real
$ float32 imaginary
```

Definir uma nova mensagem - Exemplo - 2

- Para que o ROS gere o código de mensagem específico do idioma, precisamos informar o sistema de compilação sobre as novas definições de mensagem.
- Podemos fazer isso adicionando estas linhas ao nosso arquivo *package.xml* do **pacote**:

```
$ <build_depend>message_generation</build_depend>
$ <exec_depend>message_runtime</exec_depend>
```

Definir uma nova mensagem - Exemplo - 3

- Precisamos fazer algumas alterações no arquivo *CMakeLists.txt* do **pacote**.
- Primeiro, precisamos adicionar `message_generation` ao final da chamada `find_package()`, para que o `catkin` saiba procurar o pacote `message_generation`

```
$ find_package(catkin REQUIRED COMPONENTS
$     roscpp
$     rospy
$     std_msgs
$     message_generation
$ )
```

Definir uma nova mensagem - Exemplo - 4

- Dizemos a `catkin` quais arquivos de mensagem queremos compilar adicionando-os à chamada `add_message_files()`:

```
$ add_message_files(
$     FILES
$     Complex.msg
$ )
```

Definir uma nova mensagem - Exemplo - 5

- Finalmente, ainda no arquivo `CMakeLists.txt`, precisamos garantir que a chamada `generate_messages()` não seja comentada e contenha todas as dependências necessárias às nossas mensagens.

```
$ generate_messages(
$     DEPENDENCIES
$     std_msgs
$ )
```

Definir uma nova mensagem - Exemplo - 6

- Agora que dissemos tudo o que precisamos saber sobre nossas mensagens, estamos prontos para compilá-las.
- Vá para a raiz do seu espaço de trabalho `catkin` e execute `catkin_make`.
- Isso gerará um tipo de mensagem com o mesmo nome que o arquivo de definição de mensagem, com a extensão .msg removida.

```python
#!/usr/bin/env python3

import rospy
from pack.msg import Complex
from random import random

rospy.init_node("message_publisher")

pub = rospy.Publisher("complex", Complex, queue_size = 1)
rate = rospy.Rate(2)

while not rospy.is_shutdown():
    msg = Complex()
    msg.real = random()
    msg.imaginary = random()
    pub.publish(msg)
    rate.sleep()
```

Usar a nova mensagem - 1

- A importação do seu novo tipo de mensagem funciona como incluir um tipo de mensagem ROS padrão.
- O nome da biblioteca é o nome do pacote.

```
$ from pack.msg import Complex
```

- Depois de criar a instância, você pode preencher os valores para os campos individuais.
- Quaisquer campos aos quais não seja atribuído um valor explicitamente devem ser considerados como tendo um valor indefinido.

```
$ msg = Complex()
$ msg.real = random()
$ msg.imaginary = random()
```

```python
#!/usr/bin/env python

import rospy
from pack.msg import Complex

def callback(msg):
    print("Real: {}".format(msg.real))
    print("Imaginary: {}".format(msg.imaginary))

rospy.init_node("message subscriber")
sub = rospy.Subscriber("complex", Complex, callback)
rospy.spin()
```

Usar a nova mensagem - 2

- O comando `rosmsg` permite examinar o conteúdo de um tipo de mensagem.
- Se uma mensagem contiver outras mensagens, elas serão exibidas recursivamente por `rosmsg`.

```
$ >>rosmsg show Complex
$ [basics/Complex]:
$ float32 real
$ float32 imaginary
```

- `rosmsg list`: mostra todas as mensagens disponíveis no ROS.
- `rosmsg packages`: lista todos os pacotes que definem mensagens.
- `rosmsg package`: lista as mensagens definidas em um pacote específico

Serviços

Serviços

- Os serviços são outra maneira de transmitir dados entre nós no ROS.
- Serviços são apenas chamadas de procedimento remoto síncronas; eles permitem que um nó chame uma função que é executada em outro nó.
- Definimos as entradas e saídas dessa função de maneira semelhante à maneira como definimos novos tipos de mensagens.
- O servidor (que fornece o serviço) especifica um callback para lidar com a solicitação de serviço e anuncia o serviço.
- O cliente (que chama o serviço) acessa esse serviço por meio de um proxy local.
- As chamadas de serviço são adequadas para coisas que só precisam ser feitas ocasionalmente.

Definir um serviço - 1

- A primeira etapa na criação de um novo serviço é definir as entradas e saídas da chamada de serviço.
- Isso é feito em um arquivo de definição de serviço, que possui uma estrutura semelhante aos arquivos de definição de mensagem
- No entanto, uma chamada de serviço possui entradas e saídas.
- Exemplo: um serviço adiciona dois inteiros.
  - A entrada para a chamada de serviço deve ser dois números inteiros.
  - A saída deve ser um número inteiro.

Definir um serviço - 2

AddTwoInts.srv

```
$ int64 A
$ int64 B
$ ---
$ int64 Sum
```

- O arquivo que contém essa definição é chamado *AddTwoIntst.srv* e está tradicionalmente em um diretório chamado *srv* no diretório principal do pacote.
- As entradas para a chamada de serviço vêm primeiro.
- Três traços (- - -) marcam o final das entradas e o início de a definição de saída.
- Finalmente, as saídas para a chamada de serviço vêm.

Definir um serviço - 3

- Precisamos executar `catkin_make`a partir do diretório principal do pacote para criar as definições de código e classe que realmente usaremos ao interagir com o serviço.
- Temos que fazer uma adição ao arquivo ***package.xml*** **do diretório principal do pacote** para refletir as dependências tanto no rospy quanto no sistema de mensagens.

```
$ <build_depend>rospy</build_depend>
$ <exec_depend>rospy</exec_depend>
$
$ <build_depend>message_generation</build_depend>
$ <exec_depend>message_runtime</exec_depend>
```

Definir um serviço - 4

- Precisamos fazer algumas alterações no arquivo ***CMakeLists.txt*** **no diretório principal do pacote**.
- Primeiro, precisamos adicionar `message_generation` ao final da chamada `find_package()`, para que o `catkin` saiba procurar o pacote `message_generation`

```
$ find_package(catkin REQUIRED COMPONENTS
$     roscpp
$     rospy
$     message_generation)
```

Definir um serviço - 5

- Precisamos informar quais arquivos de definição de serviço queremos compilar, usando a chamada `add_service_files()`.

```
$ add_service_files(
$     FILES
$     AddTwoInts.srv)
```

Definir um serviço - 6

- Devemos garantir que as dependências para o arquivo de definição de serviço sejam declaradas, usando a chamada `generate_messages()`.

```
$ generate_messages(
$     DEPENDENCIES
$     std_msgs
$ )
```

- A execução de `catkin_make` gerará três classes: `AddTwoInts`, `AddTwoIntsRequest` e `AddTwoIntsResponse`.
- Essas classes serão usadas para interagir com o serviço.
- Podemos verificar se a definição de chamada de serviço é o que esperamos usando o comando `rossrv`.

Implementar um serviço - 1

- Serviços são um mecanismo baseado em callback.
- O provedor especifica um callback que será executado quando a chamada de serviço for feita e aguarda a chegada de solicitações.

```python
#! /usr/bin/python3

import rospy

# Import the service classes: servive definition, response message
from pkg_name.srv import AddTwoInts, AddTwoIntsResponse

# Definition of the function called by the serice
def callback_AddTwoInts(req):

    # Create a response variable
    res = AddTwoIntsResponse()
    # Compute the sum
    res = req.A + req.B
    # Return the response variable
    return res
```

```
1  if __name__ == "__main__":
2
3      # Start the node
4      rospy.init_node("AddTwoInts_server_node")
5      # Start the service server
6      my_service = rospy.Service('add_two_ints_service_name',
       AddTwoInts, callback_AddTwoInts)
7
8      print("Ready to add two ints.")
9
10     # Wait to be closed by the user
11     rospy.spin()
```

Implementar um serviço - 2

- Primeiro precisamos importar o código gerado pelo `catkin`.

```
$ from pkg_name.srv import AddTwoInts,AddTwoIntsResponse
```

- Precisamos importar o `AddTwoInts` e o `AddTwoIntsResponse`.
- Ambos são gerados em um módulo Python com o **mesmo nome do pacote**, com uma extensão *.srv*.

Implementar um serviço - 3

- A função callback
  - usa um único argumento do tipo `AddTwoIntsRequest`
  - retorna um único argumento do tipo `AddTwoIntsResponse`

```
$ def callback_AddTwoInts(AddTwoIntsRequest):
$     return AddTwoIntsResponse
```

Implementar um serviço - 4

- Na função MAIN
- Depois de inicializar o nó, anunciamos o serviço, fornecendo:
  - um nome (add_two_ints_service_name);
  - um tipo (`AddTwoInts`);
  - um handler (h_AddTwoInts).

```
$ my_service = rospy.Service('add_two_ints_service_name', AddTwoInts,
handle_AddTwoInts)
```

- Fazemos uma chamada para `rospy.spin()`, que fornece o controle do nó para o ROS e sai quando o nó está pronto para desligar.

Verificar se tudo funciona como esperado

```
$ rosrun pkg_name simpleServer.py
```

```
$ rosservice list
```

```
$ rosservice info AddTwoInts
```

```
1  #! /usr/bin/python3
2
3  import rospy
4  from aula_offline.srv import *
5
6  if __name__ == "__main__":
7
8      # Wait for the server to be started
9      rospy.wait_for_service('add_two_ints_service_name')
10     print("add_two_ints service is active")
```

```
1     try:
2         # Connect to the server
3         h_AddTwoInts = rospy.ServiceProxy('
    add_two_ints_service_name', AddTwoInts)
4
5         # Create and fill the resquest
6         request = AddTwoIntsRequest()
7         print('Service call')
8         request.A = 2
9         request.B = 3
10        # Call the service
11        response = h_AddTwoInts(request)
12        print(response.Sum)
13
14    # If the connection fails (DEBUG)
15    except rospy.ServiceException as e:
16        print("Service call failed: %s"%e)
```

Usar um serviço - 1

- Primeiro, esperamos que o serviço seja anunciado pelo servidor.

```
$ rospy.wait_for_service('add_two_ints_service_name')
```

- Se tentarmos usar o serviço antes de ser anunciado, a chamada falhará.
- Essa é uma grande diferença entre tópicos e serviços.

Usar um serviço - 2

- Depois que o serviço é anunciado, podemos configurar um proxy local para ele.

```
$ h_AddTwoInts = rospy.ServiceProxy('add_two_ints_service_name',
AddTwoInts)
```

- Precisamos especificar o nome do serviço (add_two_ints_service_name) e o tipo (AddTwoInts).
- Isso nos permitirá usar o h_AddTwoInts como uma função local que, quando chamada, realmente fará a chamada de serviço para nós.

```
$ response = h_AddTwoInts(request)
```

Comunicação assíncrona vs síncrona

- Publisher / Subscriber
  - many-to-many
  - One-way transport
  - Assíncrono
  - Code will not block
- Servicio
  - One to one
  - Two-way transport
  - Síncrono
  - Code will block

RQt

- RQt é uma estrutura de interface gráfica do usuário que implementa várias ferramentas e interfaces na forma de plug-ins.
- Pode-se executar todas as ferramentas GUI existentes como janelas encaixáveis no RQt
- Você pode executar qualquer ferramenta / plug-in rqt facilmente:

```
$ >>rqt
```

- Vantagens:
  - Procedimentos comuns padronizados para GUI
  - Vários widgets encaixáveis em uma única janela
  - Suporte multiplataforma e multilíngue

RVIZ

RVIZ

- RVIZ é uma ferramenta de visualização 3D para aplicativos ROS.
- Ele fornece uma visão do modelo do seu robô, captura informações do sensor dos sensores do robô e reproduz os dados capturados.
- Ele pode exibir dados de câmeras, lasers e dispositivos 3D e 2D, incluindo imagens e nuvens de pontos.
- Para iniciá-lo, digite (com roscore rodando):

```
$ >>rviz rviz
```

RVIZ - 2

- Informe ao rviz qual frame fixo queremos usar.
  No grupo 'Displays', no item 'Global Options', clique no rótulo do quadro ao lado de 'Fixed Frame'. Digite ou selecione 'base_link'.
- Visualizando o modelo do robô
  Clique em 'Adicionar' e vá até 'rviz >RobotModel' e clique em 'OK'.
- Visualizando Informações do Sensor
  - Clique em "Add"e adicione um "Item". Selecione um topico no campo "Topic"
  - Clique em "Add"e adicione um "Topic.
- Para salvar a configuração como padrão, clique em "File >Save Config".
  Na próxima vez que você executar o rviz, ele carregará essa configuração.

Mensagens de visualização

- `visualization_msgs` é um conjunto de mensagens usado por pacotes de nível superior, como rviz, que lidam com dados específicos de visualização `http://wiki.ros.org/visualization_msgs`
- A principal mensagem em `visualization_msgs` é `visualization_msgs/Marker` `http://docs.ros.org/en/api/visualization_msgs/html/msg/Marker.html`
- A mensagem `Marker` é usada para enviar "marcadores"de visualização como caixas, esferas, setas, linhas, etc. para um ambiente de visualização como o rviz

Exemplo

```
$ marker = Marker()
$ marker.header.frame_id = "base_link"
$ marker.type = marker.SPHERE
$ marker.action = marker.ADD
$ marker.scale.x = 0.05
$ marker.scale.y = 0.05
$ marker.scale.z = 0.05
$ marker.color.a = 1.0
$ marker.color.r = 1.0
$ marker.color.g = 0.0
$ marker.color.b = 1.0
$ marker.pose.orientation.w = 1.0
$ marker.pose.position.x = x
$ marker.pose.position.y = y
$ marker.pose.position.z = z
```

Rosbag

Gravação de todos os tópicos publicados

- É possível gravar dados de um sistema ROS em execução em um arquivo .bag, e depois reproduzir os dados para produzir um comportamento semelhante em um sistema em execução
- Os tópicos publicados são os únicos tipos de mensagem que poderiam potencialmente ser gravados no arquivo de registro de dados, já que somente as mensagens publicadas são gravados.
- Para gravar todos os tópicos, abrir um terminal e digitar:

```
$ >>mkdir bagfiles
$ >>cd bagfiles
$ >>rosbag record -a
```

- O usuário escolhe o nome e a localização do diretório.
- -a, indica que todos os tópicos publicados devem ser acumulados em um rosbag.

Examinando e tocando o arquivo rosbag

- Para ver o que está gravado no arquivo rosbag, usamos o comando `info`:

```
$ >>rosbag info <your bagfile>
```

- Reproduzir o rosbag para reproduzir o comportamento no sistema em execução:

```
$ >>rosbag play <your bagfile>
```

Gravação de um subconjunto dos dados

- Pode haver centenas de tópicos sendo publicados, com alguns tópicos, como fluxos de imagens de câmeras, potencialmente publicando enormes quantidades de dados.
- Em tal sistema, muitas vezes é impraticável gravar rosbag que consistem em todos os tópicos.
- O comando rosbag suporta a gravação de apenas determinados tópicos em um rosbag, permitindo aos usuários registrar apenas os tópicos de seu interesse.

```
$ >>rosbag record -O subset /turtle1/cmd_vel /turtle1/pose
```

Servidor de parâmetros

O que é o servidor de parâmetros?

- Um servidor de parâmetros é um dicionário compartilhado que pode ser acessado por meio de APIs de rede
- Os nós usam esse servidor para armazenar e recuperar parâmetros em tempo de execução
- Como não foi projetado para alto desempenho, é melhor usado para dados estáticos, como parâmetros de configuração
- Ele pode ser visualizado globalmente para que as ferramentas possam inspecionar facilmente o estado de configuração do sistema e modificá-lo, se necessário

Tipos de parâmetros e nomes

- Tipos de parâmetros
  - 32-bit integers
  - booleans
  - strings
  - doubles
  - iso8601 dates
  - lists
  - base64-encoded binary data
- Os parâmetros são nomeados usando a convenção de nomenclatura normal do ROS

```
$ /camera/left/name:  leftcamera

$ /camera/left/exposure:  1

$ /camera/right/name:  rightcamera

$ /camera/right/exposure:  1.1
```

rosparam

- `rosparam` permite a configuração e obtenção de parâmetros, bem como carregar e despejar o estado do Parameter Server em um arquivo

```
$ rosparam set # definir parâmetro
$ rosparam get # obter parâmetro
$ rosparam load # carregar parâmetros do arquivo
$ rosparam dump # despejar parâmetros para arquivo
$ rosparam delete # excluir parâmetro
$ rosparam list # listar nomes de parâmetros
```

Formato YAML

- YAML é uma linguagem leve que suporta todos os tipos de parâmetros
- Os parâmetros podem ser definidos em arquivos codificados em YAML

```
$ string:  'foo'
$ integer:  1234
$ float:  1234.5
$ boolean:  true
$ list:  [1.0, mixed list]
$ dictionary:  a:  b, c:  d
```

Os parâmetros em um nó

- Obter os parâmetros:
  `rospy.get_param(param_name)`

```
$ name = rospy.get_param("/name")
$ default_param = rospy.get_param('default_param', 'default_value')
```

- Configurar os parâmetros:
  `rospy.set_param(param_name, param_value)`

```
$ rospy.set_param('a_string', 'baz')
$ rospy.set_param('list_of_floats', [1., 2., 3., 4.])
$ rospy.set_param('bool_True', True)
$ rospy.set_param('gains', 'p': 1, 'i': 2, 'd': 3)
```

Os parâmetros em um nó - 2

- Existência do parâmetro:
  `rospy.has_param(param_name)`

```
$ if rospy.has_param('to_delete'):
$     rospy.delete_param('to_delete')
```

- Excluindo parâmetros:
  `rospy.delete_param(param_name)`

```
$ try:
$     rospy.delete_param('to_delete')
$ except KeyError:
$     print("value not set")
```

roslaunch

O que é o `roslaunch`?

- `roslaunch` é uma ferramenta para:
  - Iniciar facilmente vários nós ROS, localmente e remotamente via SSH
  - Definir parâmetros no Parameter Server
  - Redefinir nomes de nó e dos tópicos (remaping)
- `roslaunch` recebe um ou mais arquivos de configuração XML (com a extensão .launch) que especificam os parâmetros a serem definidos e os nós a serem executados, bem como as máquinas nas quais eles devem ser executados
- O pacote `roslaunch` contém as ferramentas roslaunch, que lêem o formato roslaunch .launch/XML. Ele também contém uma variedade de outras ferramentas de suporte para ajudá-lo a usar esses arquivos
- Um `roslaunch` iniciará automaticamente o `roscore` se detectar que ainda não está em execução

Executar `roslaunch`

- Muitos pacotes ROS vêm com arquivos `roslaunch`, localizados em um diretório launch do pacote
- Comando para executar um arquivo `roslaunch`:

```
$ roslaunch <package-name> <launch-filename> [args]
```

- Exemplo

```
$ roslaunch roslaunch example.launch
```

- O `roslaunch` instancia automaticamente um `roscore` se não existir quando o `roslaunch` é chamado.
- O `roslaunch` fecha todos os seus nós quando Ctrl-C é pressionado no console que contém o `roslaunch`.

- Exemplo: talker_listener.launch

```
$ <launch>
$     <node name="talker"  pkg="rospy_tutorials"
$           type="talker.py"  output="screen"/>
$     <node name="listener"  pkg="rospy_tutorials"
$           type="listener.py"  output="screen"/>
$ </launch>
```

- Cada tag `<node>` inclui atributos:
  - Nome do nó no gráfico ROS.
  - O pacote em que pode ser encontrado.
  - O tipo de nó, que é simplesmente o nome do arquivo do programa executável.
  - Saídas para o console atual.

roslaunch com parâmetros

```
$ <launch>
$     <rosparam file="path/file.yaml" command="load"/>
$     <node name="talker"  pkg="rospy_tutorials"
$           type="talker.py"  output="screen"/>
$     <node name="listener"  pkg="rospy_tutorials"
$           type="listener.py"  output="screen"/>
$ </launch>
```

`roslaunch` com remapeamento de tópico

```
$ <launch>
$     <rosparam file="path/file.yaml" command="load"/>
$     <node name="talker"  pkg="rospy_tutorials"
$           type="talker.py"  output="screen"/>
$     <node name="listener"  pkg="rospy_tutorials"
$           type="listener.py"  output="screen"/>
$           <remap from="topic1" to="topic2"/>
$     </node>
$ </launch>
```