



Universidade Federal de Itajubá



**Prof. Guilherme Sousa Bastos
Adriano Henrique Rossette Leite
Audeliano Wolian Li**

Expertinos



Universidade Federal de Itajubá

ROS

Introdução

Dificuldades e Motivação

Dificuldade ao criar uma aplicação multi-robô:

- Cada empresa possui um protocolo de comunicação;
- Sensores de mesma natureza produzem dados em formatos diferentes;
- Como controlar vários robôs em computadores diferentes?
- Para cada aplicação nova deve ser implementada do zero?

How Robotics
Research Keeps...

Re-Inventing the Wheel

First, someone publishes...



...and they write code that barely works but lets them publish...



...a paper with a proof-of-concept robot.



This prompts another lab to try to build on this result...



...but they can't get any details on the software used to make it work...



But inevitably, time runs out...



...and countless sleepless nights are spent writing code from scratch.



So, a grandiose plan is formed to write a new software API...



...and all the code used by previous lab members is a mess.

Definição

ROS (Robot Operating System)

- O que é:
 - Um sistema que disponibiliza bibliotecas e ferramentas para auxiliar na criação de aplicações robóticas.

Características

É caracterizado como um sistema operacional porque fornece:

- abstração de hardware;
- drivers de dispositivos;
- bibliotecas;
- transmissão de mensagens;
- gerenciamento de pacotes;
- e outros.

Características

É suportado por várias plataformas:

- Ubuntu
- Experimental:
 - Debian
 - Windows
 - Ubuntu ARM
 - OS X

Vantagens

Open Source

- Contribuição da comunidade

Multi-linguagem

- C/C++
- Python
- Java (rosjava)
- MATLAB (2014)
- LabVIEW

Modular

- Independência entre aplicações
- Reaproveitamento de programas
- Facilidade na implementação

Histórico

Desenvolvido em 2007

- *Stanford Artificial Intelligence Laboratory*

De 2008 - 2013

- Willow Garage

- Empresa incubadora de cunho tecnológico
- Mais de 20 instituições colaboraram com o desenvolvimento

08/2013 - atual

- *Open Source Robotics Foundation*

ROS Distros

22 janeiro 2010 – ROS 1.0

1 março 2010 – Box Turtle

3 agosto 2010 – C Turtle

2 março 2011 – Diamondback

30 agosto 2011 – Electric Emys

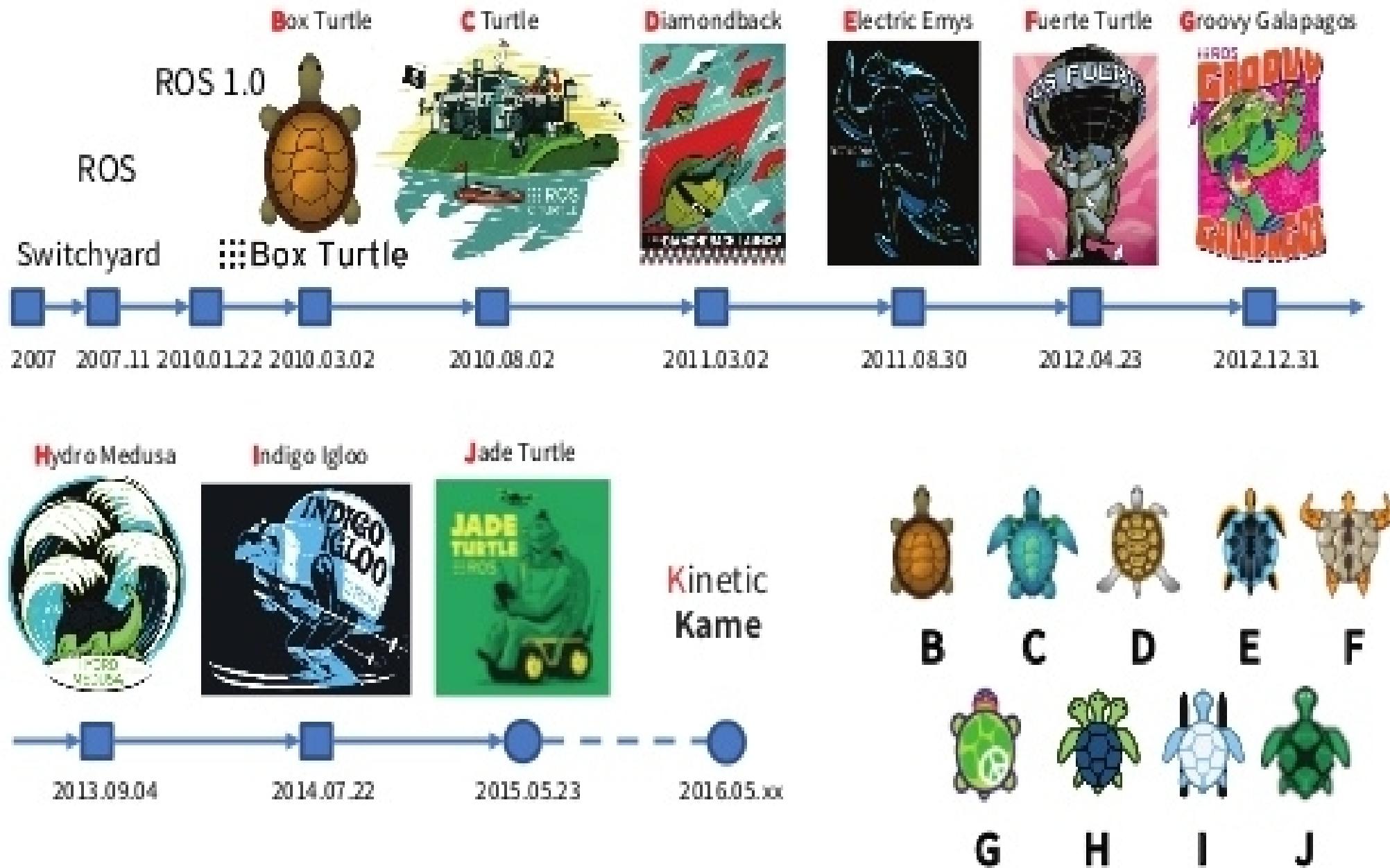
23 abril 2012 – Fuerte

31 dezembro 2012 – Groovy Galapagos

4 setembro 2013 – Hydro Medusa

22 julho 2014 - Indigo Igloo

23 maio 2015 - Jade Turtle



Robôs Suportados pelo ROS



210 Stanley Innovation
V3 Segway



220 Stanley Innovation
V3 Segway



420 Omni Stanley
Innovation V3 Segway



440LE Stanley
Innovation V3 Segway



440SE Stanley
Innovation V3 Segway



ABB Robotics (ROS-
Industrial)



Adept MobileRobots
Pioneer family (P3DX,
P3AT, ...)



Adept MobileRobots
Pioneer LX



Adept MobileRobots
Seekur family (Seekur,
Seekur Jr.)



Aldebaran Nao



Allegro Hand SimLab



AMIGO



AscTec Quadrotor



Barrett Hand



BipedRobin



Bitcraze Crazyflie

Sensores Suportados pelo ROS

Sensores 2D

- laserscan

Sensores 3D

- kinect

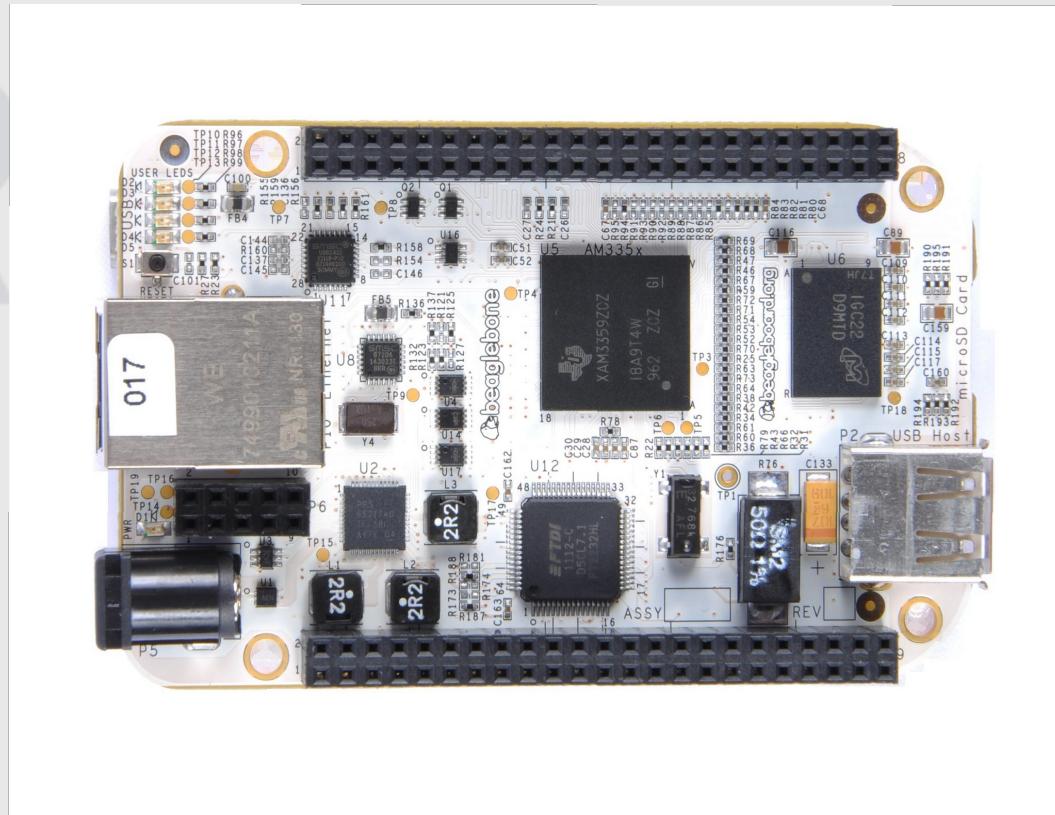
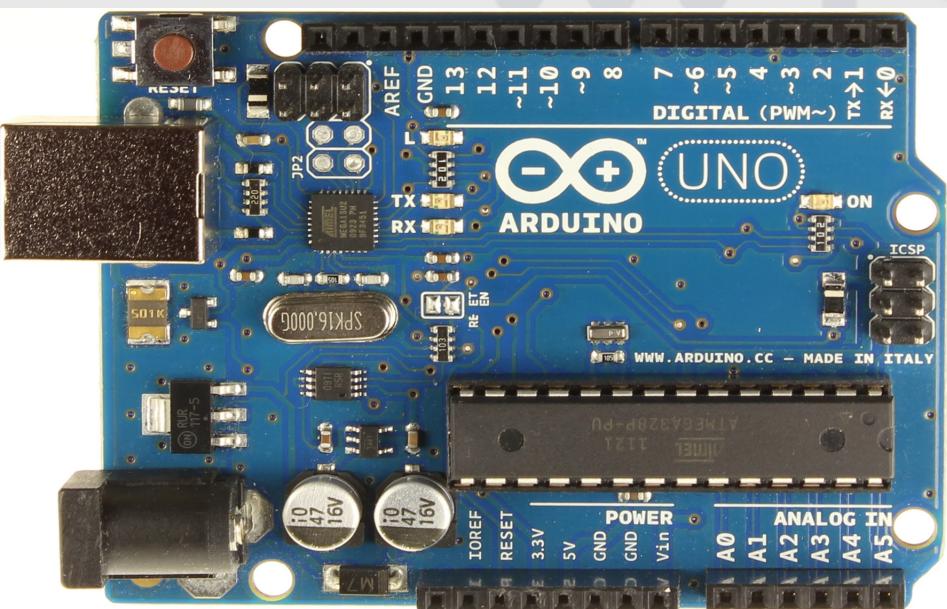
Câmeras

- usb, PTZ, IP, ...

Estimador de posição

- IMU, GPS

Hardware Suportados pelo ROS



Projetos



ROS - an open-source Robot Operating System

Acadêmico

Artigos

Minha biblioteca

A qualquer momento

Desde 2015

[PDF] ROS: an open-source Robot Operating System

[M Quigley, K Conley, B Gerkey... - ICRA workshop on ..., 2009 - pub1.willowgarage.com](#)

Abstract—This paper gives an overview of ROS, an opensource robot operating system. ROS is not an operating system in the traditional sense of process management and scheduling; rather, it provides a structured communications layer above the host operating ...

Citado por 1804 Artigos relacionados Todas as 23 versões Citar Salvar Mais

Projetos

Vídeos:

ROS Three Years

ROS Five Years

Projetos na UNIFEI / LRO



Projetos na UNIFEI / LRO

Foi criado um pacote para gerenciamento dinâmico de banda.

- Tese de mestrado do Ricardo Emerson e orientado pelo professor Dr. Guilherme Bastos Movimentação de um robô usando gestos dos braços com auxílio de um kinect.
- Trabalho de conclusão de curso, artigo será publicado em 10/2015.
- Vídeo: Navegação Robótica por Gestos

Projetos na UNIFEI / LRO

Várias teses de mestrado e iniciações científicas,
tais como:

- robótica cooperativa
- fusão sensorial
- navegação robótica
- processamento de imagens

Vídeo: Autonomous Navigation on Erratic Robot
Using ROS

Projetos na UNIFEI / LRO

Todo ano a equipe Expertinos participa da CBR na categoria Festo Logistics e lá é possível implementar todos os trabalhos e estudos feitos na área da robótica.

Vídeo: RoboCup 2014 Playoffs vs. UVM_Ingenieria

Projetos na UNIFEI / LRO

Este ano a equipe participará da categoria At Home e vários estudos estão sendo feitos nas áreas de:

- navegação
- reconhecimento de voz
- interface gráfica
- visão computacional
- vários outros temas

ROS Industrial

20 de março de 2013.

Utiliza a plataforma ROS.

É um programa de desenvolvimento para criação de URDF (Unified Robot Description Format) para robôs industriais.

ABB, Fanuc, Motoman, outros.

- Vídeo: SwRI's ROS-Industrial Interoperability Demonstration at Automate 2013



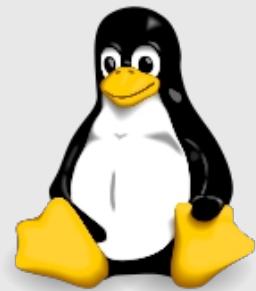


Universidade Federal de Itajubá



Conceitos Básicos Linux

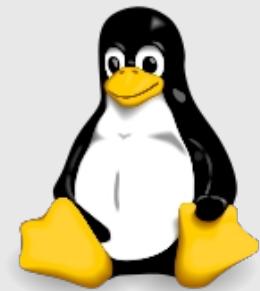
Distros do Ubuntu e Comandos no Terminal



GNU/Linux

Linux é o **núcleo**, o kernel, do sistema operacional que faz a comunicação entre **hardware** e **software**. O conjunto **kernel** e demais programas responsáveis por interagir com este é o que denominamos sistema operacional.

Os principais programas responsáveis por interagir com o kernel foram criados pela fundação **GNU**.



Distribuições (Distros)

É o conjunto de kernel, programas de sistema e aplicativos reunidos em uma única mídia.

Nós utilizaremos neste minicurso (agosto/2015) a distro Ubuntu 12.04 LTS

*LTS: Long Term Support (5 anos de suporte)

O Terminal

No ínicio todos os sistemas operacionais utilizavam apenas uma **interface modo texto**, CLI. Porém, graças a evolução tecnológica temos diversos sistemas operacionais que disponibilizam uma **interface gráfica amigável**.

Contudo, a interface no modo texto no Linux também evoluiu juntamente com as novas tecnologias, possibilitando uma administração mais detalhada, completa e complexa do Linux. [1]

[1] Disponível em: <http://wiki.ubuntu-br.org/Terminal> Acesso em: 11/08/2015

O Terminal

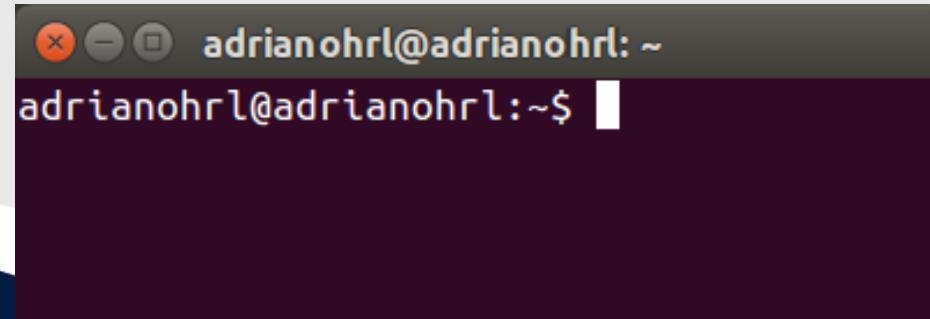
adrianohrl → nome do usuário ativo no terminal;

@ → popularmente, 'pertence' ou 'vinculado';

adrianohrl → nome da estação de trabalho, host;

~ → atual localização de acesso aos arquivos é no diretório pessoal (home) do usuário ativo;

\$ → status do usuário comum; caso seja #, será super usuário (root).



Atalhos Utéis

Ctrl+Alt+T → Abre o terminal em uma nova janela;

Ctrl+Shift+T → Abre outro terminal em uma nova tab;

Ctrl+C → Aborta aplicação em execução no terminal;

Tab → completa comando do prompt;

Double Tab → lista possíveis entradas no prompt;

Ctrl+Shift+C → Copia conteúdo selecionado no terminal;

Ctrl+Shift+V → Cola conteúdo copiado para o terminal;

Botão de Rolagem do Mouse → cola conteúdo do texto selecionado; minimiza janela.

Conceitos de Diretórios

- / → diretório raiz;
- ~ → diretório pessoal (home);
- .. → diretório de nível acima do atual;
- . → diretório atual;

Privilégio Sudo

Sempre que um comando necessitar de **privilégios de administrador**, torna-se necessário o uso do “sudo”. Este ,também, deve ser adicionado na frente de todos os comandos, caso esteja trabalhando em um diretório ou com arquivos que não lhe pertencem (arquivos do sistema, por exemplo).

Para mais detalhes sobre super usuário e privilégio sudo, visite [1].

Comandos Básicos^[1]

pwd → permite saber qual é o diretório atual;

ls *dir* → lista o conteúdo do diretório *dir*;

ls -a *dir* → lista todo o conteúdo do diretório *dir*, inclusive arquivos ocultos;

cd *dir* → navega entre a árvore de diretórios do sistema;

cp *file copy* → copia arquivos e diretórios;

mv *src dst* → move arquivos e diretórios (usado também para renomear arquivos).

Comandos Básicos^[1]

rm *file* → remove arquivos;

rm -r *dir* → remove diretórios;

rm -f *file* → remove arquivos forçadamente;

rm -rf *dir* → remove diretórios forçadamente;

mkdir *dir* → cria diretórios;

mkdir -p *dir* → cria qualquer diretório pai faltante.

Comandos Básicos^[1]

export *ENV*=”...” → define conteúdo de variável de ambiente;

echo \$*ENV* → visualiza conteúdo da variável de ambiente;

export | grep *ENV* → lista todas as variáveis de ambiente, e seus respectivos conteúdos, que possuem a sub-string *ENV* em seu nome;

ifconfig → exibe as configurações das interfaces de redes (é possível identificar o IP address);

ping → testa conectividade com outros equipamentos da rede.

O Arquivo `~/.bashrc`

É um **arquivo oculto** na pasta pessoal que é executando sempre que um novo terminal é aberto.

Adicionamos a eles comandos que usamos sempre que um novo terminal é aberto a fim de facilitar nossa vida.



Universidade Federal de Itajubá

ROS

Instalação

Instalação

<http://wiki.ros.org/ROS/Installation>

Fuerte: Ubuntu 10.04, 11.10, 12.04

Hydro: Ubuntu 12.04, 12.10, 13.04

Indigo: Ubuntu 13.10, 14.04

Jade: Ubuntu 14.04, 14.10, 15.04

Migration guide

VirtualBox



Universidade Federal de Itajubá

ROS

Conceitos

Conceitos

Analogias

Nós (Nodes)

Mestre (Master)

Mensagens (Messages)

Tópicos (Topics)

Serviços (Services)

Pacotes (Packages)

Pilhas vs Meta-pacotes (Stacks vs Meta-packages)

Bags

Analogias

Para facilitar o entendimento dos conceitos serão feitos três analogias:

- robótica;
- editora de revistas;
- youtube.

Nodes

Nós são processos que executam algum tipo de computação, podendo ser considerados como executáveis.

Robótica	Editora	YouTube
Controlador do robô	Editor	Youtuber
Anel de sensores ultrassônicos	Assinante	Público Alvo
Câmera		
Processamento de imagem		
Planejador de trajetórias		

Messages

Os nós se comunicam entre si através de mensagens.
Sendo cada uma com um tipo diferente de informação:

- integer;
- boolean;
- array.

E também um conjunto de tipos primitivos (struct):

- geometry_msgs/Pose2D
- http://docs.ros.org/api/geometry_msgs/html/msg/Pose2D.html

Pode-se também criar sua própria mensagem (será abordado mais a frente).

Messages

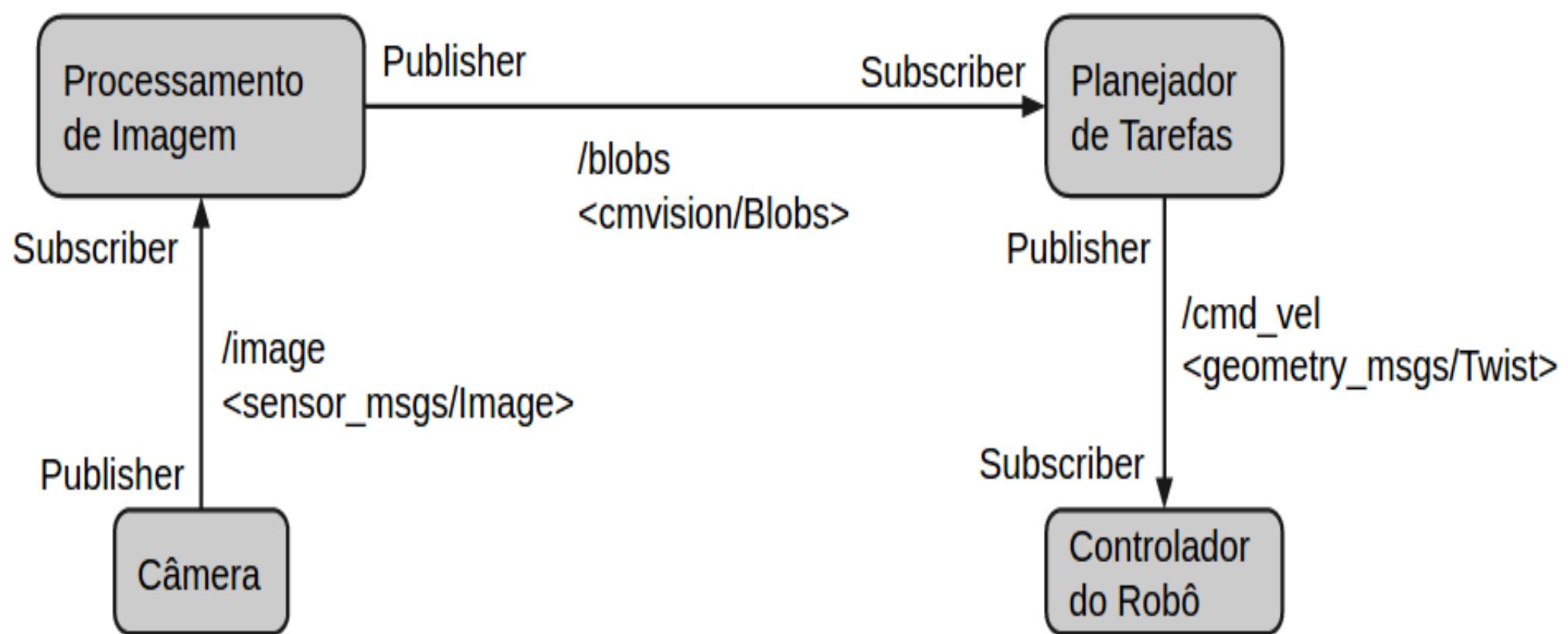
Robótica	Editora	YouTube
geometry_msgs/Pose2D	científica	comédia
geometry_msgs/Twist	entretenimento	documentário
integer	informativo	educativo
sensor_msgs/Image		

Topics

- Os nós trocam informações através dos tópicos, que podem ser considerados como um **barramento de dados**.
- Os nós que colocam dados em um certo tópico são chamados de **publisher**.
- Enquanto os nós que fazem a leitura dos dados presentes no tópico, são os **subscriber**.
- Em um mesmo nó podemos ter tanto publishers quanto subscribers.
- OBS: o tipo do tópico deverá ser igual ao tipo da mensagem.

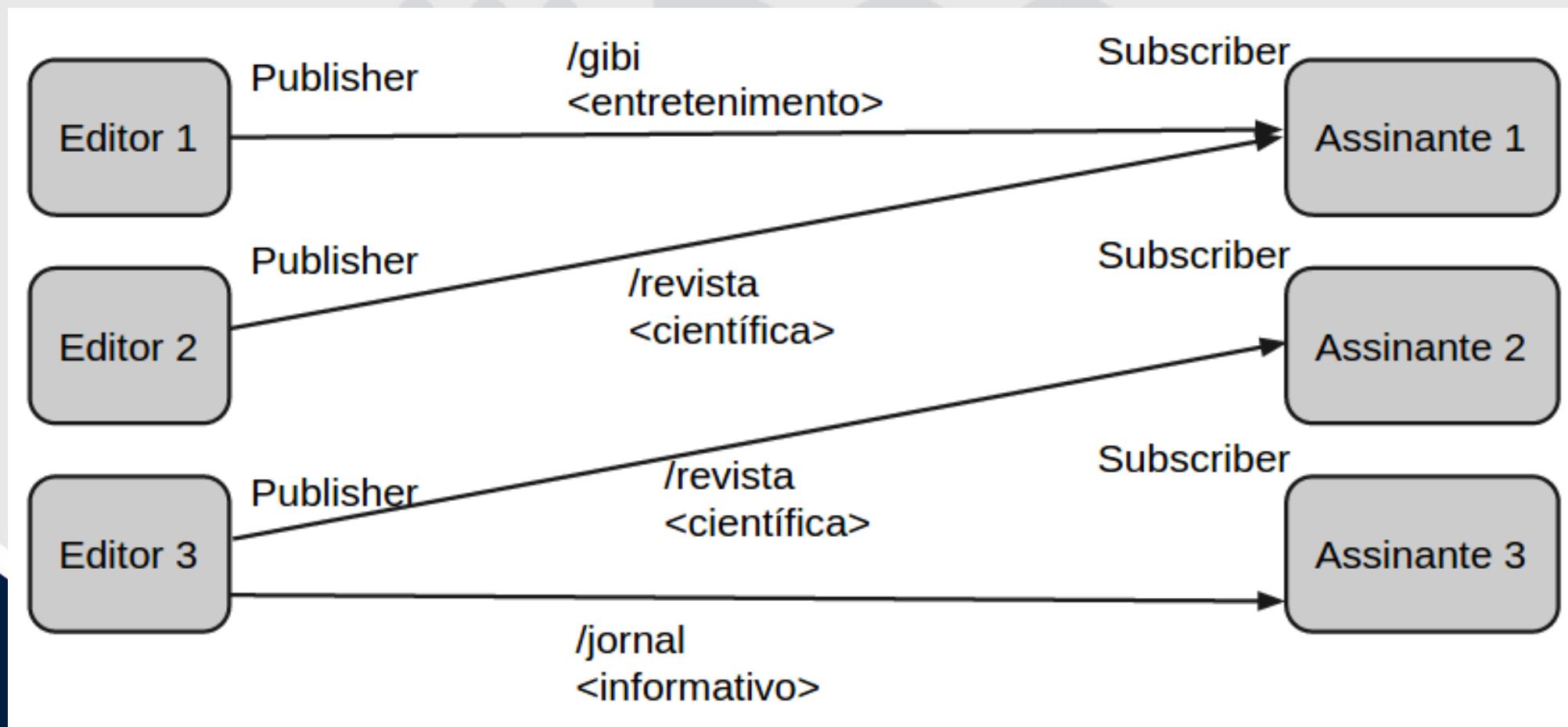
Topics

› Robótica



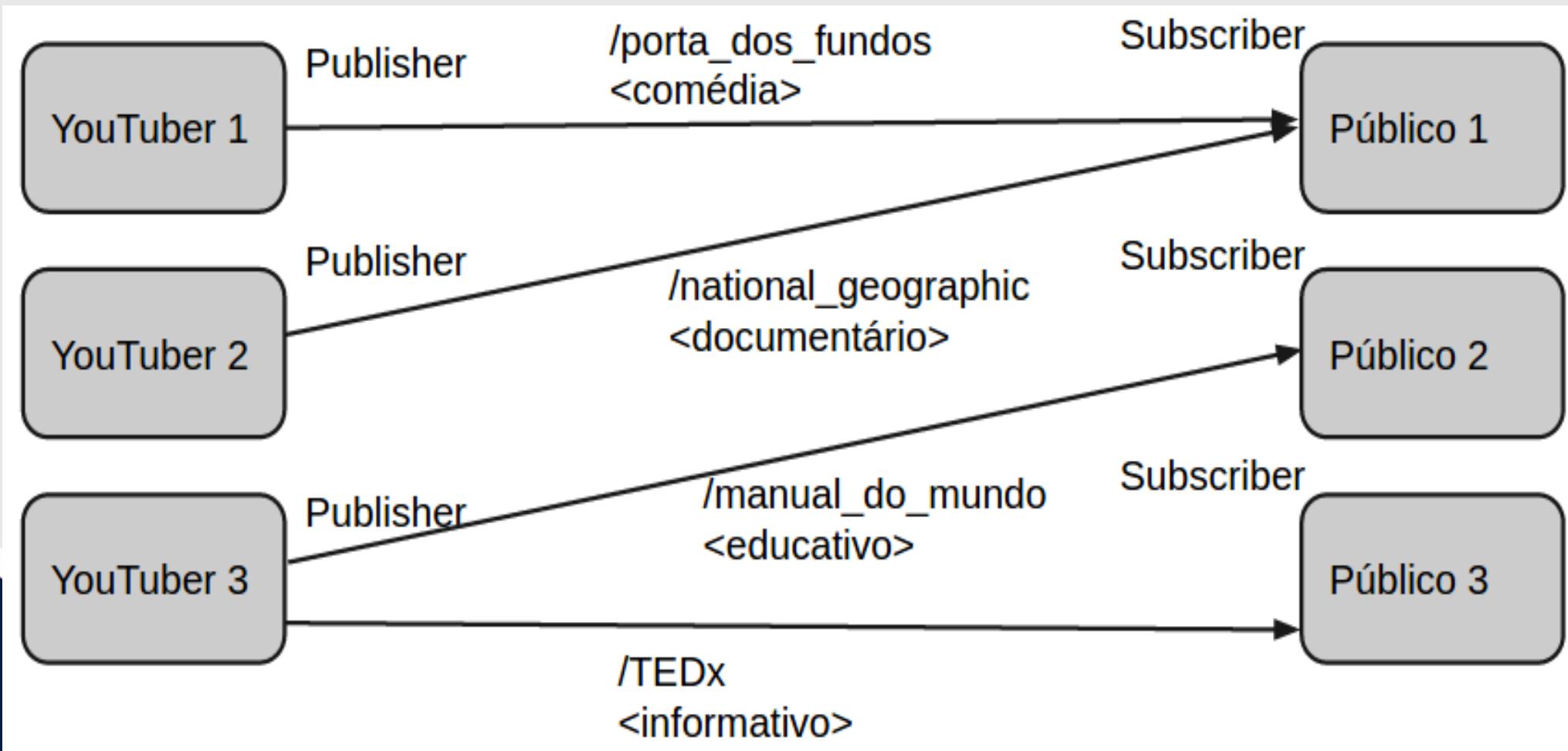
Topics

➤ Editora



Topics

➤ YouTube



Services

Os publishers e os subscribers não foram projetados para requisitar/responder algo de um nó.

Para essa interação tornar-se possível, criou-se os serviços.

Através deles um nó pode requisitar uma tarefa e receber uma resposta, ambas em forma de mensagem.

	Robótica	Editora	YouTube
Serviço	Menor trajetória	Assinatura	Inscrever-se

Master

Sempre deverá ser **inicializado** primeiramente.

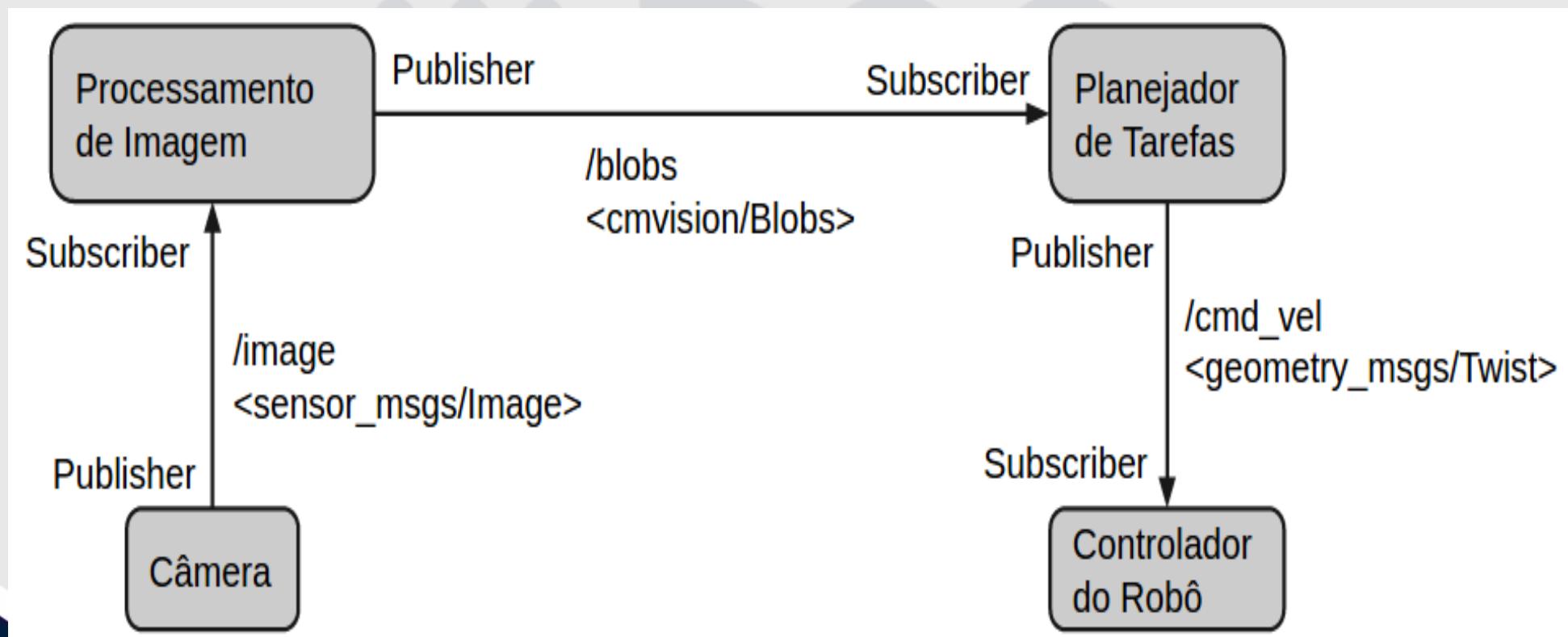
É um nó que provê serviços de registro e consulta de nomes de outros nós, de tópicos e de serviços.

Sempre que um nó é iniciado, ele se registra com o mestre.

O nó que subscreve um tópico faz uma consulta ao mestre e estabelece uma conexão direta com este.

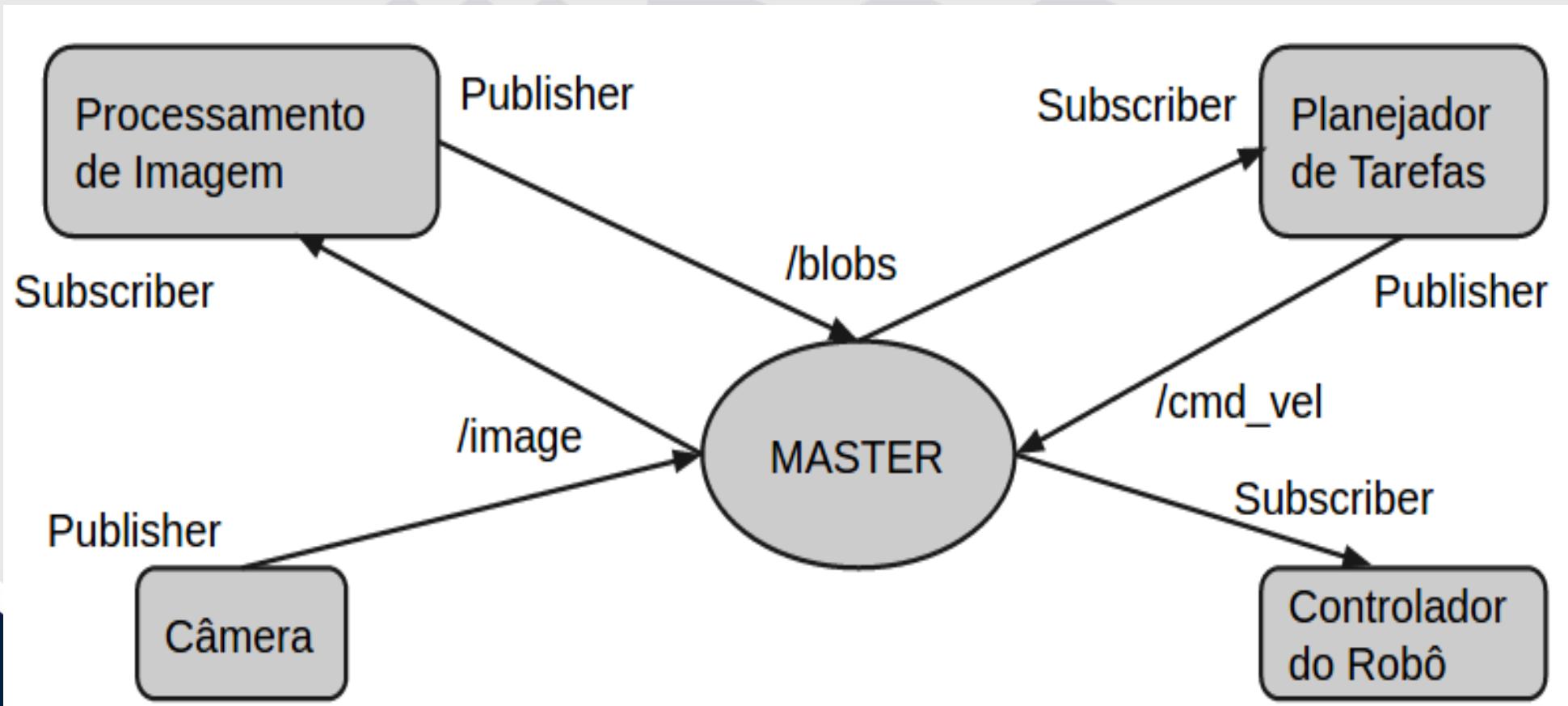
Topics

› Robótica



Master

› Robótica



Packages

Os pacotes são a principal unidade de organização no ROS.

Agrupam arquivos de mesma natureza:

Pacote_modelagem_robótica

- includes..... cabeçalhos (.h)
- src..... códigos-fonte (.cpp)
 - camera_node.cpp
 - processamento_imagem.cpp
- manifest.xml..... informações sobre o pacote
- Cmakelists.txt..... configuração Cmake
- launch, msg, srv etc

Mostrar pacote hokuyo

Stacks vs Meta-Packages

O conceito de stack foi utilizado até o ROS Fuerte (compilador rosbuild) e nas versões posteriores (**catkin**) foi implementado os meta-packages.

A definição básica dos dois se manteve a mesma:

- os meta-packages são pacotes que **agrupam** outros pacotes que estão relacionados entre si.

Mostrar pacote Navigation

Bags

Bags são arquivos para **armazenar** dados de uma aplicação, tais como: nós, **tópicos**, mensagens, etc.

E posteriormente é possível reproduzi-los sem precisar executar no sistema (robô).

É uma ferramenta muito importante para testes de algoritmos, apresentação de projetos, estudo de possíveis problemas, dentre outros.



Universidade Federal de Itajubá

ROS

Tutorial

Pré-requisito

```
$ sudo apt-get install ros-<distro>-ros-tutorials
```

- substituir '<distro>' pelo ros distro utilizado: hydro

Criando o Workspace

```
$ roscd
```

- viki@c3po:/opt/ros/hydro

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

```
$ cd ~/catkin_ws/src
```

```
$ catkin_init_workspace
```

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

```
$ catkin_make
```

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

```
$ roscd
```

- viki@c3po:~/catkin_ws/devel

Criando um Pacote

Criar um pacote chamado beginner_tutorials:

```
$ cd ~/catkin_ws/src
```

```
$ catkin_create_pkg beginner_tutorials std_msgs  
rospy rospy
```

- # catkin_create_pkg <package_name> [depend1]
[depend2] [depend3]

```
$ cd ~/catkin_ws
```

```
$ catkin_make
```

Adicionar o workspace no ambiente ROS:

```
$ . ~/catkin_ws/devel/setup.bash
```

Entendendo sobre os Nós

Sempre inicializar o nó master:

\$ roscore

\$ rosnode list

- */rosout*

\$ rosnode info /rosout

Entendendo sobre os Nós

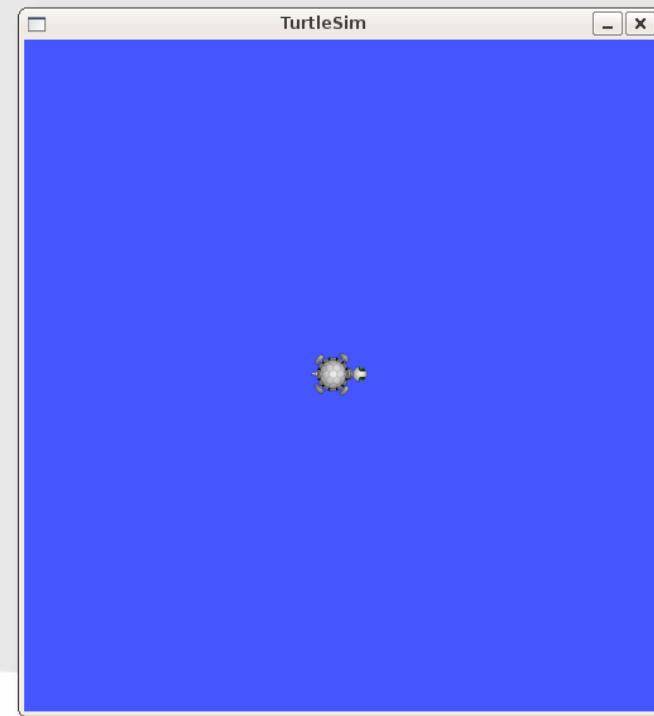
Executar um segundo nó:

```
$ rosrun turtlesim turtlesim_node
```

- rosrun [package_name] [node_name]

```
$ rosnodes list
```

- /rosout
- /turtlesim



Entendendo sobre os Nós

Uma característica poderosa do ROS está na possibilidade de **atribuir nomes** através da linha de comando:

Feche a janela do turtlesim e digite no terminal:

```
$ rosrun turtlesim turtlesim_node __name:=my_turtle
```

```
$ rosnode list
```

- /rosout
- /my_turtle

```
$ rosnode ping my_turtle
```

Entendendo sobre Tópicos

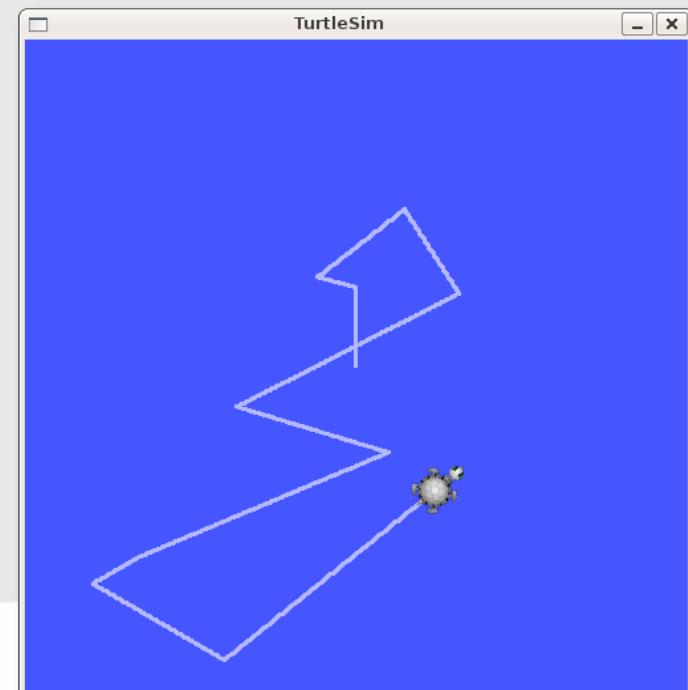
Feche a janela do turtlesim e execute os comandos:

```
$ rosrun turtlesim turtlesim_node  
$ rosrun turtlesim turtle_teleop_key
```

Para visualizar os nós e tópicos de forma gráfica:

```
$ rosrun rqt_graph rqt_graph
```

- caso o rqt não esteja instalado:
 \$ sudo apt-get install ros-hydro-rqt
 \$ sudo apt-get install
 ros-hydro-rqt-common-plugins



Entendendo sobre Tópicos

Para obter informações dos tópicos usa-se o comando **rostopic**:

```
$ rostopic -h
```

```
$ rostopic echo /turtle1/cmd_vel (atualize o  
rqt_graph)
```

```
$ rostopic list
```

```
$ rostopic list -v
```

Entendendo sobre Tópicos

Para obter o tipo da mensagem que algum tópico está publicando:

```
$ rostopic type /turtle1/cmd_vel
```

- rostopic type [tópico]
- geometry_msgs/Twist

```
$ rosmsg show geometry_msgs/Twist
```

Entendendo sobre Tópicos

Para publicar um dado através de um tópico via terminal:

```
rostopic pub [tópico] [tipo_da_mensagem] [args]
```

```
$ rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist --  
'[2.0, 0.0, 0.0]' '[0.0, 0.0, 1.8]'
```

- velocidade linear em x = 2.0
- velocidade angular z = 1.8

```
$ rostopic pub /turtle1/cmd_vel geometry_msgs/Twist -r 1  
-- '[2.0, 0.0, 0.0]' '[0.0, 0.0, 1.8]'
```

```
$ rosrun rqt_graph rqt_graph
```

Entendendo sobre Serviços

Um serviço permite que nós enviem um pedido e recebam uma resposta.

\$ rosservice list

\$ rosservice type [service]

- clear
- spawn

\$ rosservice call [service] [args]

- clear
- spawn 2 2 0.2 ""

Entendendo sobre Roslaunch

roslaunch é usado para inicializar os nós desejados, inclusive o roscore.

- `roslaunch [package] [filename.launch]`

criar uma pasta para os arquivos launch:

- `$ roscd beginner_tutorials`
- `$ mkdir launch`
- `$ cd launch`
- criar um arquivo chamado `turtlemimic.launch`

Entendendo sobre Roslaunch

```
<launch>
  <group ns="turtlesim1">
    <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
  </group>

  <group ns="turtlesim2">
    <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
  </group>

  <node pkg="turtlesim" name="mimic" type="mimic">
    <remap from="input" to="turtlesim1/turtle1"/>
    <remap from="output" to="turtlesim2/turtle1"/>
  </node>
</launch>
```

Entendendo sobre Roslaunch

```
<launch>
    <group ns="turtlesim1">
        <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
    </group>

    <group ns="turtlesim2">
        <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
    </group>

    <node pkg="turtlesim" name="mimic" type="mimic">
        <remap from="input" to="turtlesim1/turtle1"/>
        <remap from="output" to="turtlesim2/turtle1"/>
    </node>
</launch>
```

Entendendo sobre Roslaunch

```
$ rosrun beginner_tutorials turtlemimic.launch  
$ rostopic pub /turtlesim1/turtle1/cmd_vel  
geometry_msgs/Twist -r 1 -- '[2.0, 0.0, 0.0]' '[0.0, 0.0,  
-1.8]'  
$ rqt_graph
```

Criando ROS msg e srv

Arquivos **msg** descrevem o tipo e formato da mensagem a ser usada no ROS.

Os tipos das mensagens podem ser:

- int8, int16, int32, int64
- float32, float64
- string
- time, duration
- outros arquivos de mensagem
- Array de tamanho variado ou fixo

Criando ROS msg e srv

Arquivos **srv** são semelhantes aos arquivos msg, porém com uma diferença: possuem duas partes - o pedido e a resposta, separados entre si através de '---'.

int64 A

int64 B

int64 Sum

A e B são os pedidos e Sum é a resposta

Criando ROS msg e srv

Para **criar uma msg**

```
$ cd ~/catkin_ws/src/beginner_tutorials  
$ mkdir msg  
$ echo "int64 num" > msg/Num.msg
```

Abra o **package.xml** e **descomente** as linhas:

```
<build_depend>message_generation</build_depend>  
<run_depend>message_runtime</run_depend>
```

Criando ROS msg e srv

Modificações a serem feitas no **CMakeLists.txt**

Adicionar message_generation em find_package(...)

- `find_package(catkin REQUIRED COMPONENTS
roscpp
rospy
std_msgs
message_generation
)`
- `catkin_package(
...
CATKIN_DEPENDS message_runtime ...
...)`

Em add_message_files(...) editar colocando o arquivo .msg criado:

- add_message_files(
 FILES
 Num.msg
)

Descomentar:

- generate_messages(
 DEPENDENCIES
 std_msgs
)

\$ catkin_make

\$ rosmsg show beginner_tutorials/Num

Criando ROS msg e srv

Para criar um srv

```
$ cd ~/catkin_ws/src/beginner_tutorials
```

```
$ mkdir srv
```

Iremos copiar um arquivo .srv de outro pacote para demonstrar o exemplo:

```
$ roscp rospy_tutorials AddTwoInts.srv  
srv/AddTwoInts.srv
```

Criando ROS msg e srv

Em add_service_files(...) editar colocando o arquivo .srv criado:

- add_service_files(
FILES
AddTwoInts.srv
)

\$ catkin_make

\$ rossrv show beginner_tutorials/AddTwoInts

Navegando no wiki ros

- <http://wiki.ros.org>
- <http://answers.ros.org>
- <http://www.ros.org/browse/list.php>
 - [http://www.ros.org/browse/list.php?
package_type=package&distro=hydro](http://www.ros.org/browse/list.php?package_type=package&distro=hydro)

dynamic_bandwidth_manager

Ricardo Emerson Julio, Guilherme
Sousa Bastos

dynamic_bandwidth_manager is a ros-based dynamic bandwidth
management system for controlling the rat...



Universidade Federal de Itajubá

ROS

Aria e Rosaria

Aria e Rosaria

Os robôs Pioneer 3DX e AmigoBot são robôs desenvolvidos pela empresa **Adept MobileRobot** que disponibiliza uma biblioteca **open source**, chamada **Aria**, para facilitar na criação de novas aplicações.

Aria e Rosaria

Ao trabalhar com o Rosaria, o programador desfruta da facilidade dos dois mundos:

- Através do **Aria**, a **conexão** dos robôs é feita com apenas uma linha de código e existem várias **classes** já criadas disponíveis nesta biblioteca.
- Através do **Rosaria**, é possível visualizar as informações em cada tópico e enviar comandos através das ferramentas do **ROS**. Além de todas as funções que o ROS oferece.

<http://www.ai.rug.nl/vakinformatie/pas/content/Aria-manual/classes.html>

Aria e Rosaria

Abra o simulador **Mobilesim**

Escolha o modelo do robô e clique em '**no map**'

No terminal:

`$roscore`

`$rosrun rosaria RosAria`

`$rostopic list`

`$rostopic pub /RosAria/cmd_vel geometry_msgs/Twist
'[1.0, 0.0, 0.0]' '[0.0, 0.0, 2.0]'`

`$rosservice list`

`$rosservice call /disable_motors`



Universidade Federal de Itajubá

ROS

Visão Geral

Visão Geral

- Imaginemos a seguinte situação:
- Várias **editoras publicam** jornais, artigos, livros, revistas e e-books para seus **assinantes**, cada uma em sua especialidade.
- Os assinantes se **inscrevem** para receber assuntos que julgam ser interessantes para si; geralmente, pensando no conteúdo da mensagem passada através do canal (jornal, artigo, livro, revista, e-book, etc).

Visão Geral

- As editoras podem também servir alguns serviços para seus clientes; tais como, cadastro de um novo assinante, envio de sugestões de leitura, orçamento de um plano ou de um material específico.
- Vamos analisá-la agora...

Visão Geral

Falemos que:

As editoras e os assinantes terão seus processos próprios (**nós**). Exemplo:

- a editora possui um processo de passar o conteúdo (mensagem) para o jornal (canal/tópico), impressão gráfica, ela também possui uma ouvidoria que oferece alguns serviços e trabalha em paralelo com o processo de impressão. A princípio, estes dois processos (nós/aplicações) não estão diretamente relacionados, entretanto, ambos estão relacionados a editora.

Visão Geral

- o assinante, por sua vez, pode ter processos como leitura, ouvir música, busca de novo acervo, etc. Estes podem ser paralelos ou não. Consideraremos que eles não são executados em paralelo.

Visão Geral

- Para que tudo fique organizado, teremos um pacote que define editoras e um outro pacote que define assinantes.
- Modelando este sistema, podemos obter:

2 pacotes (editora e assinante);

4 nós (assinante, assinante culto, impressão e ouvidoria);

5 tópicos (gibi, artigo, jornal, revista e livro);

3 mensagem (texto, texto ilustrado e dados do cliente);

2 serviço (colheta acervo e cadastro de assinante);

Vejamos, agora, o conceito por trás de cada um destes termos e, ao final, vejamos como podemos organizar o projeto.

Visão Geral

Detalhando os nós:

- impressao_node (Node)
 - jornal<editora_pkg/Texto> (Topic Publisher)
 - revista<editora_pkg/TextoIlustrado> (Topic Publisher)
 - livro<editora_pkg/Texto> (Topic Publisher)
 - gibi<editora_pkg/TextoIlustrado> (Topic Publisher)
 - artigo<editora_pkg/Texto> (Topic Publisher)
- ouvidoria_node (Node)
 - add_assinante<editora_pkg/CadastraAssinante> (Service Server)
 - get_acervo<editora_pkg/ColhetaAcervo> (Service Server)

Visão Geral

leitura_stck (Stack)

- editora_pkg (Package)
 - impressao_node (Node)
 - ouvidoria_node (Node)
 - editora_pkg/Texto (Message)
 - editora_pkg/Textollustrado (Message)
 - editora_pkg/CadastraAssinante (Service)
 - editora_pkg/ColhetaAcervo (Service)
- assinante_pkg (Package)
 - assinante_node (Node)
 - assinante_culto_node (Node)
 - assinante_pkg/Dados (Message)

- assinante_node (Node)
 - revista<editora_pkg/TextoIlustrado> (Topic Subscriber)
 - gibi<editora_pkg/TextoIlustrado> (Topic Subscriber)
 - add_assinante<editora_pkg/CadastraAssinante> (Service Cliente)
- assinante_culto_node (Node)
 - livro<editora_pkg/Texto> (Topic Subscriber)
 - jornal<editora_pkg/Texto> (Topic Subscriber)
 - artigo<editora_pkg/Texto> (Topic Subscriber)
 - add_assinante<editora_pkg/CadastraAssinante> (Service Cliente)



Universidade Federal de Itajubá



Conceitos Básicos C++

Falando um pouco sobre arquivos de extensões .h e .cpp

Cabeçalho (Header)

Arquivos de extensão .h que definem a classe que o implementa, isto é, inclui todas as **bibliotecas** necessárias, define **herança**, **constantes**, **construtores**, **destrutores**, **atributos**, **métodos** e outros, todos com suas **sobrecargas** e **níveis de acesso**.

Para melhor organização, ficam na pasta **include**. Lembre-se de respeitar os níveis do namespace, **cada nível tem uma pasta**.

Código-Fonte (Source)

Arquivos de extensão **.cpp** que implementam classes definidas por arquivos de extensão **.h** (ou não) e, também, implementam aplicações.

Contém **regras de negócios** e **cálculos**.

Nos casos de implementar uma classe definida por um **.h**, este **inclui apenas** seu **.h**, preferencialmente.

Erros comuns de acontecer

- #include ...
- using namespace ...
- void my_ns::MyClass::setVal(float val){}
- ; {} [] () :,
- my_ns::MyClass::MyClass() : myObj(?) {}
- conversões entre tipos

roscpp

É uma biblioteca que **implementa** o ROS, é a biblioteca **mais utilizada** por clientes e é projetada para ser a biblioteca **de maior desempenho** do ROS.

A comunidade ROS definiu um **estilo padrão** de programação em C++ [1]. Para publicar seus pacotes na comunidade ROS, **é desejável que seu código esteja dentro deste padrão**.

[1] Disponível em: <http://wiki.ros.org/CppStyleGuide> Acesso em: 11/08/2015

ros::init()

É um método que possui várias sobrecargas e que deve ser **chamado antes** de usar qualquer outra parte do sistema ROS.

Defini o nome do nó, o qual deve ser **único** e **não pode ter barra** (/) como caracter inicial, ou seja, sua nomenclatura é do **tipo base** obrigatoriamente.

ros::NodeHandle

Nó é um termo do ROS para um **executável** que se conecta com a rede do ROS.

NodeHandle é o principal **ponto de acesso** para se **comunicar** com o sistema ROS.

O primeiro NodeHandle construído inicializará completamente o nó e o último a ser destruído finalizará o nó.

`ros::ok()`

Retorna um booleano, que será **false**:

- quando a aplicação for interrompida pelo usuário (**Ctrl+C**) ou pelo ROS, por existir outro nó com o mesmo nome;
- quando **ros::shutdown()** for chamado em outra parte da aplicação;
- ou quando todos os **ros::NodeHandles** forem **destruidos**.

Callbacks

São métodos, **funções**, ou rotinas que são **invocadas** na aplicação.

ros::spinOnce()

Este método é importante em nós que possuem algum tipo de **callback**, pois ele é responsável por invocá-las.

ros::spin()

Basicamente, este método implementa `ros::ok()` e `ros::spinOnce()`, entretanto, possui uma limitação. As callbacks são chamadas na **mesma thread** que a aplicação.

Este método executa um loop que invoca as callbacks do nó enquanto Ctrl+C não é pressionado ou não é interrompido pela master.

ros::Publisher^[1]

ros::NodeHandle possui um método que retorna um objeto ros::Publisher chamado **advertise()** que o define e o cria. Você diz ao ROS, através deste método, que você quer **publicar** em um **tópico** específico. Ele invoca uma chamada para o nó mestre do ROS, que mantém um registro de quem está **publicando** e de quem está **subscrevendo** nos tópicos.

...

ros::Publisher^[1]

Após a chamada do método `advertise()`, o nó mestre irá **notificar** todos nós que estão **tentando subscrever** neste tópico. Então, eles poderão negociar uma **conexão peer-to-peer** com este nó.

Publisher possui um método chamado **publish()** que permite **publicar mensagens do tipo especificado** em `advertise<>()`, momento em este é construido.

ros::Publisher^[1]

Outro método importante de Publisher é chamado **shutdown()**, o qual é responsável por **destruir o objeto** e, se estiver for o último a ser destruído, o tópico será automaticamente **unadvertised**.

Para mais informações sobre ros::Publisher, vide:

<http://wiki.ros.org/roscpp/Overview/Publishers%20and%20Subscribers>

ros::Subscriber^[1]

ros::NodeHandle possui um método que retorna um objeto ros::Subscriber chamado **subscribe()** que diz ao ROS que você quer **receber** mensagens de um **tópico** específico. Ele invoca uma chamada para o nó mestre do ROS, que mantém um registro de quem está **publicando** e de quem está **subscrevendo** nos tópicos.

...

ros::Subscriber^[1]

As **mensagens** são passadas através de uma função **callback** definida na criação do objeto ros::Subscriber. A callback é chamada sempre que chega uma nova mensagem.

Quando todas as cópias do objeto estiverem **fora de escopo**, esta callback **perde automaticamente sua subscrição** neste tópico.

ros::ServiceServer

ros::NodeHandle possui um método que retorna um objeto ros::ServiceServer chamado **advertiseService()** anuncia ao ROS o tipo de serviço que pode ser **prestashop** pelo nó.

Uma **callback** que retorna um booleano e possui parâmetros de entrada do tipo **requisito** (Request) e **resposta** (Response) do serviço, deve ser definida.

ros::ServiceClient

ros::NodeHandle possui um método que retorna um objeto ros::ServiceClient chamado **serviceClient()** que cria um cliente de um **serviço** específico e que permiti **solicitá-lo** à qualquer instante através de seu método **call()**.

IMPORTANTE: solicitação de serviços, **travam** o nó!!! Cuidado com isso.

ros::ServiceClient

Quando o método `call()` é executado, seu retorno será **verdadeiro** se a **resposta** do serviço for **válida**. Caso contrário, seu retorno será falso.

ros::Duration, ros::Time e ros::Rate

ros::Rate permite especificar a frequência que se deseja rodar o loop. Através do seu método **sleep()**, este objeto controla o tempo necessário de pausa para que a frequência estabelecida permaneça constante.

ros::Duration trata durações, enquanto ros::Time trata tempo. ros::Time::**now()** retorna time stamp atual.

Erros comuns de acontecer

- **Tabulação** entre tipo e nome do atributo (arquivos .msg, .srv e .action);
- Sempre **compilar** após qualquer alteração de código (**exceto** em arquivos **launch**);
- Alterar arquivos **CmakeList.txt** e **package.xml** sempre que necessário.



Universidade Federal de Itajubá



Nomenclatura no ROS

Estrutura hierárquica de nomes para nós,
parâmetros, tópicos e serviços no ROS

Nomenclatura

Estes nomes são muito **poderosos** no ROS e em sua central já que este gerencia **sistemas grandes e complicados**. Sabendo disso, é muito importante entender como esses nomes trabalham e como podemos manipulá-los. [1]

Exemplos:

- /unifei/name
- /unifei/labs/lro/map
- /unifei/robots/p3_dx/identifier

Nomenclatura

Promeve **encapsulamento** para os **recursos** (nós, parâmetros, tópicos e serviços).

Cada recurso é definido dentro de um **namespace**, que é compartilhado com outros vários recursos.

Em geral, recursos podem **criar** recursos dentro do seu namespace e **acessar** recursos dentro ou acima do seu namespace.

Nomenclatura

Promeve **encapsulamento** para os **recursos** (nós, parâmetros, tópicos e serviços), evitando com que recursos sejam acessados acidentalmente, e **simplifica** a programação.

Cada recurso é definido dentro de um **namespace**, que é compartilhado com outros vários recursos.

Em geral, recursos podem **criar** recursos dentro do seu namespace, **acessar** recursos dentro ou acima do seu namespace e se **conectar** com recursos de qualquer namespace.

Tipos de Nomes^[1]

- base
- relativo/nome
- /global/nome
- ~privado/nome



Node	Relative (default)	Global	Private
/node1	bar -> /bar	/bar -> /bar	~bar -> /node1/bar
/wg/node2	bar -> /wg/bar	/bar -> /bar	~bar -> /wg/node2/bar
/wg/node3	foo/bar -> /wg/foo/bar	/foo/bar -> /foo/bar	~foo/bar -> /wg/node3/foo/bar

[1] Disponível em: <http://wiki.ros.org/Names> Acesso em: 11/08/2015