ROS - Parte 1

Programação Robótica
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.

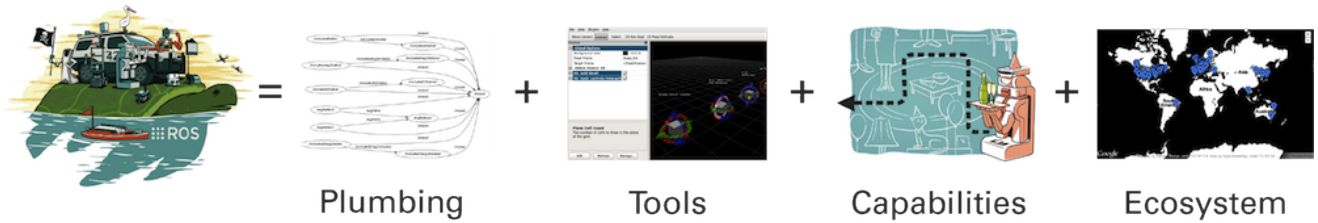
Exemplo - 1

- 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.

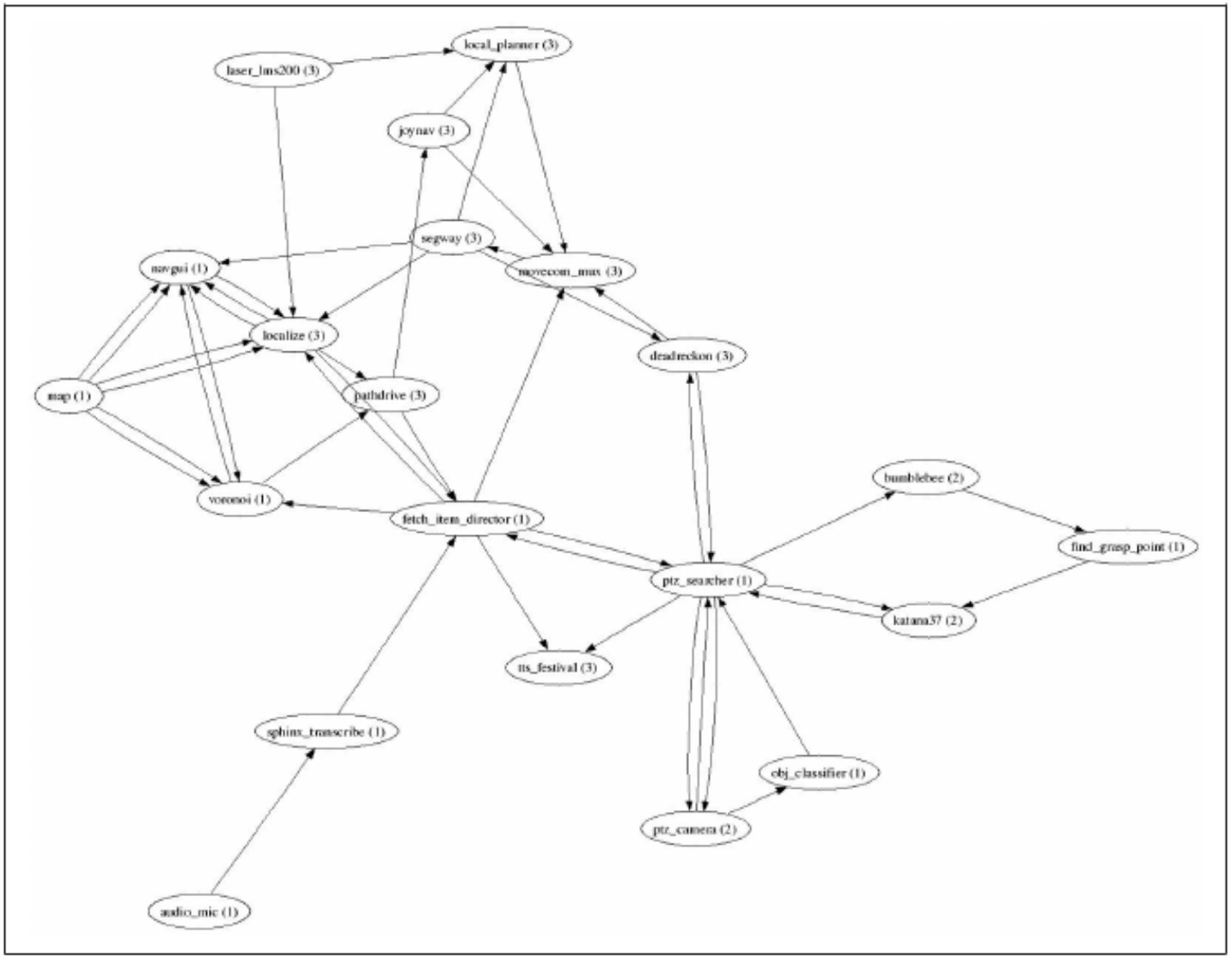
Conceitos chaves

- Os sistemas ROS são compostos por um grande número de programas independentes que estão constantemente se comunicando.
  - ROS Graph
  - roscore
  - catkin
  - Workspaces
  - ROS packages
  - rosrun
  - roslaunch

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



Exemplo de gráfico

roscore - 1

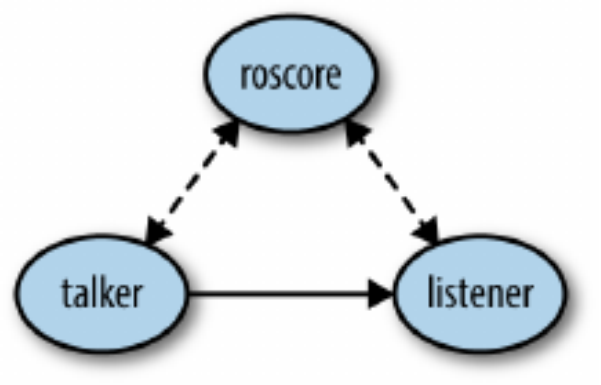
- O roscore é um serviço que fornece informações de conexão aos nós para que eles possam transmitir mensagens entre si.
- Cada nó se conecta ao roscore na inicialização para registrar detalhes dos fluxos de mensagens que publica e dos fluxos nos quais deseja assinar.
- Quando um novo nó aparece, o roscore 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.

roscore - 3

- 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.

ROS Packages - 2

- 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

- `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]
```

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`

Exemplo 1

```
#! /usr/bin/python

# Importação da biblioteca ROS para Python
import rospy

# Criação do nó com o nome hello_world
rospy.init_node('hello_world')

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.

Exemplo 2

```
#! /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()
```