Material Teórico

Treinamento de ROS I

Sumário

Introdução

Oi! Bem vindo ao treinamento de ROS! Se você é do Grupo SEMEAR, com certeza esse treinamento será útil durante os projetos! Se você não é do Grupo SEMEAR, espero que te ajude de alguma forma! Aproveite!

Meu nome é Lucas Toschi, sou aluno de Engenharia Elétrica com ênfase em Eletrônica (019) e escrevi esse treinamento. Antes de qualquer outra coisa, é importante dizer que ele é um treinamento básico que ainda precisa de melhorias, explicações e detalhes. Além disso, eu fui escrevendo e testando, mas talvez existam alguns problemas ainda. Se você encontrar algum erro ou algum problema, por favor, me envie uma mensagem pelo whatsapp - **(18) 99659-0040**. Tentarei checar e corrigi-lo o quanto antes, para evitar que outros tenham problemas futuros. Além disso, pretendo publicá-lo em breve na plataforma Medium do SEMEAR, para que mais pessoas tenham acesso.

Os requisitos deste treinamento são: alguma experiência com terminal Linux e também alguma experiência em C++. Ah, eu sei que ele parece grande, mas não tem tanta coisa assim. Eu tentei deixar o mais mastigado possível, porque espero que seja útil para pessoas com diferentes *backgrounds*. Me inspirei no modo de escrever de um professor da SEL - espero que lhe agrade!

Bom, pegue um café e mãos à obra!

O que é

Segundo a documentação oficial, Robot Operating System (ROS) é um pseudo sistema operacional para robôs. Ele conta com serviços que você espera de um sistema operacional, como abstração de hardware, controle de dispositivos baixo-nível, mensagens entre processos e gerenciamento de pacotes.

Considerando a minha experiência, ROS tem algumas vantagens interessantes:

- a capacidade de utilizar *multi-threading* com facilidade (códigos executando em paralelo)
- a flexibilidade de trabalhar com mais de uma linguagem simultaneamente
- a possibilidade de modularização das funcionalidades de um robô.

Tudo isso permite realizar códigos que sejam capazes de tratar informações de sensores e simultaneamente atuar motores de acordo com as informações obtidas. Além disso, você pode escrever códigos que gerenciam partes diferentes do robô, modularizando o sistema. Assim, caso aconteça algum problema, você pode mexer apenas em alguma parte, sem comprometer o todo. Para finalizar, temos a capacidade de utilizar tanto C++ (consideravelmente mais rápido na execução, mas mais demorado para desenvolver) quanto Python (demorado na execução, mas muito prático para desenvolver), trazendo flexibilidade e facilidade para o projeto.

Como opinião pessoal, eu realmente gostei de utilizar ROS para desenvolver o código do projeto da IEEE Open em 2021 na RMA (Núcleo de Robótica Móvel Autônoma). Ficou bem mais claro, limpo e fácil de mexer do que em outros projetos que participei. Realmente aconselho o aprendizado, pode te ajudar muito! :)

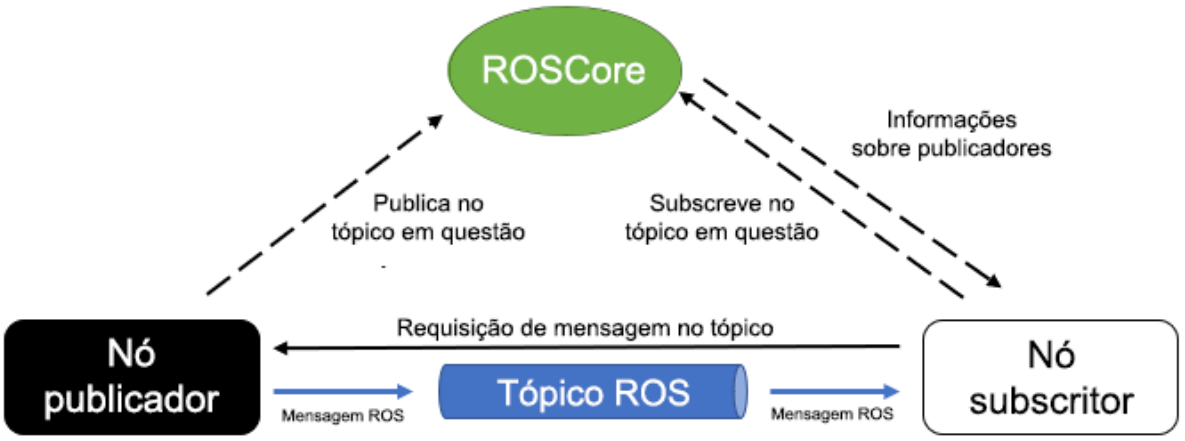
Sobre a instalação (não está descrito aqui)

Neste tutorial, o foco não é se ater a detalhes de instalação. De fato, é um processo que pode variar bastante de sistema para sistema. Inclusive, desenvolvi este documento usando Ubuntu 20, com ROS Noetic.

Recomendo **MUITO** a utilização deste tutorial aqui, oficial do ROS. Ele está em inglês, mas acredito ser o mais completo. Posteriormente, posso traduzi-lo e colocá-lo aqui também. Caso tenha alguma dúvida ou problema, entre em contato comigo!

Entendendo o paradigma

Quando você começa a programar um sistema específico, precisa entender como segmentá-lo. Entender como você pode quebrar um grande problema em pequenos pedaços é, afinal, uma das magias (e requisitos!) da engenharia. Bom, dado isso, vamos entender algumas estruturas básicas do ROS I.



Seu projeto tem códigos, correto? Claro, pô, ele precisa processar informações e produzir resultados. Isso, em ROS, é chamado de Nó (*em inglês, Node*). Ou seja, um Nó é um código que é executado de forma independente. Ou seja, ele roda independentemente de outros processos. Idealmente, ele não é influenciado por outros fatores e pode ser executado de forma única ou contínua (com *loops* sem condição de parada, por exemplo).

Vale ressaltar que ele disputa por recursos com os demais processos que estão sendo executados no dispositivo, como qualquer outro aplicativo do seu computador. Geralmente, o sistema operacional é responsável por equilibrar a distribuição de memória e processamento. Se muitos Nós estão sendo executados simultaneamente (ou se você não fechou o joguinho online), fica complicado - seu dispositivo pode *"arriar as quatro pernas"* porque ele não aguenta executar tudo isso de uma vez.

Se temos os nós como processos independentes, como as partes do robô se comunicam? Poxa, preciso que o código que trata as informações dos sensores as envie para um outro código, responsável por analisá-las e tomar decisões. Para isso, temos canais de comunicação entre Nós, chamados de Tópicos.

Como analogia, podemos pensar no Tópico como um cj7b arteiro. Só que, no mundo do Toschi, esse carteiro é um pouco irresponsável (ou talvez até fofoqueiro?). Ele literalmente GRITA para todo mundo quais encomendas ele tá entregando - pode ser um carta, pode ser um pacote da lojinha online, enfim. Pensando assim, um Tópico é um

canal de comunicação "aberto". Ele é criado por um Nó (o termo correto é "publicado") e qualquer outro Nó rodando simultaneamente pode obter as informações (ao se "inscrever" no Tópico).

Vale ressaltar que um Tópico pode carregar estruturas específicas de informação (para os amigos do C, é como uma "struct" - na analogia, a carta ou o pacote). A essa estrutura, damos o nome de Mensagem. Existem Mensagens prontas, feitas pelos desenvolvedores do ROS - mas, como nem tudo na vida é doce, eventualmente (e dependendo do que você faz, BEM eventualmente) você vai precisar desenvolver sua própria mensagem. Não é difícil e nem vamos aprender isso agora, mas é importante você entender o conceito e saber que isso é uma possibilidade. E ah! O Tópico só pode carregar UM ÚNICO tipo de mensagem, para não virar bagunça. É, esse carteiro da analogia é diferenciado mesmo.

Aliás, chegou a perceber algo? Até agora, nosso sistema ROS parece funcionar na base da magia. Olha só: temos os Nós, códigos independentes; temos os Tópicos, que são canais de comunicação por onde as Mensagens passam. Já se perguntou como um Nó sabe que outro está publicando um Tópico? Onde fica registrado qual é o tipo de Mensagem que cada Tópico carrega? Para isso, temos o processo principal de todo sistema ROS: o Núcleo ROS (em inglês, *"Roscore"*).

Essas interações entre Nós (inclusive a própria existência ou não de Tópicos) ficam registradas no *Roscore.* Você não precisa saber muito dele para poder usar ROS, mas precisa saber que, sem ele, você não consegue executar Nós. Mais pra frente, veremos sobre arquivos inicializadores (*launch*) e que, quando eles são chamados, já executam o *Roscore* antes de fazer qualquer outra coisa - ou seja, ele É IMPORTANTE.

The "Hello World" of ROS

Já ouviu dizer que aprender uma linguagem de programação sem fazer o *"Hello World"* dá má sorte? Então, ROS não é muito bem uma linguagem, mas sabe como é né: a gente não tá aqui para pagar pra ver a má sorte, independente do caso. A ideia do *"Hello World"* em ROS é explorar duas coisas principais: a possibilidade de executar mais de um Nó simultaneamente e trocar informações entre eles. Para isso, escreveremos dois códigos.

O primeiro código será o Publicador (*"Publisher"*) - ele será responsável por enviar algumas informações arbitrárias para outro Nó. O segundo código será o Subscritor (*"Subscriber"*), responsável por receber as informações e mostrá-las na tela para o usuário. Bem simples, não? Ah, nesta seção também aprenderemos umas outras coisinhas básicas para configurar o projeto, então fica esperto.

Configurando o workspace ROS

A primeira coisa que precisamos fazer é criar o workspace ROS. Para fazer isso, basta criarmos duas pastas. A primeira delas tem o nome do workspace - é um nome qualquer, arbitrário. É costume usar *"catkin_ws"*. Para isso, abra um terminal, escolha uma pasta do seu computador e digite:

```
mkdir catkin_ws
```

Dentro da *"catkin_ws"*, criaremos uma segunda pasta, chamada de *"src"*. Dentro dela fica todo o código fonte do seu projeto. Então,

```
cd catkin_ws
mkdir src
```

Perfeito. Agora precisamos executar um comando chamado *"catkin_make"*. Ele é o responsável por gerar os arquivos iniciais e compilar o projeto ROS. TODA VEZ que você fizer QUALQUER modificação nos códigos fonte, você precisa executar esse comando na pasta do workspace (a *"catkin_ws"*). Então vamos lá, volte para ela e execute

```
catkin_make
```

Se Alan Turing estiver do seu lado (amém), vai tudo gerar bonitinho e você vai perceber algumas coisinhas novas. Vão surgir duas pastas novas - uma chamada *"devel"* e uma outra, chamada *"build"*. A primeira tem arquivos de configuração gerados automaticamente e a segunda tem os executáveis do seu projeto. Dessas duas, você vai precisar usar um arquivo.

Você já ouviu falar sobre "variáveis de ambiente"? São variáveis de configuração que geralmente têm caminhos específicos para determinados pontos do seu computador. É configurando elas que você consegue executar um comando sem passar um caminho específico para um executável que tá escondido pior que uma agulha no palheiro. Existe um arquivo (o *"setup.sh")* que você SEMPRE precisa executar quando vai trabalhar em um workspace ROS. Para isso, se você estiver na pasta *"catkin_ws"*, execute

```
source devel/setup.sh
```

Ele configura as variáveis de ambiente deste projeto em específico no terminal atual e É NECESSÁRIO! Para os *nerds* de plantão, você poderia sim colocar ele no seu *".bashrc"* para sempre ser executado toda vez que você abre um terminal, mas, no geral, isso não é uma boa prática já que você pode estar trabalhando em projetos ROS diferentes em paralelo.

Criando um pacote ROS

Agora vamos precisar criar um pacote ROS. Um pacote é um conjunto de códigos, separados por alguma razão semântica. Um pacote, inclusive, é independente de outros em um mesmo *workspace.* Modularização é a coisa mais bonita disso - você pode usar um pacote fornecido por um fabricante para utilizar algum *hardware* específico com facilidade, por exemplo.

Para criarmos o pacote, entraremos na pasta *"src"* e utilizaremos o comando *"catkin_create_pkg"* descrito a seguir.

```
cd src
catkin_create_pkg *NOME_PACOTE* std_msgs rospy roscpp
```

O comando recebe alguns parâmetros. O primeiro deles é o nome do pacote que você deseja criar. É importante tentar achar um nome que seja autodescritivo e fácil de entender o objetivo do pacote. A única regra que eu já me deparei com (e que tive até problemas de compilação depois) foi não começar por um número. Também é uma boa prática iniciar com letra minúscula. Fica a dica para você não bater cabeça debugando depois (como eu fiz rs). Se quiser mais detalhes, pode encontrar aqui.

Os demais parâmetros são as dependências do pacote - nesse caso, estaremos colocando três dependências. A primeira delas, *"std_msgs"* importa algumas Mensagens padrão que são bem úteis (a carta e o pacote do carteiro da analogia, lembra?). A segunda - *"rospy"* - trás os arquivos necessários para utilizar Python com ROS. Analogamente, a terceira - *"roscpp"* - importa os arquivos necessários para utilizar C++ com ROS. Fazendo isso, você terá uma nova pasta com o nome do seu pacote.

Explorando a pasta do pacote, teremos algumas coisas importantes. Primeiro de tudo, temos uma OUTRA pasta *"src"*. A ideia é que os códigos fontes DESTE pacote em específico fiquem dentro dessa pasta. Existe também uma segunda pasta, chamada de *"include"* - nela, podem ser inseridas bibliotecas específicas ou até mesmo cabeçalhos (arquivos *header* para C++) pertencentes à organização do seu código.

Além disso, existem dois arquivos importantes. O primeiro é chamado de *"package.xml"*. Ele é um arquivo com informações para desenvolvedores. É onde você pode colocar o nome do desenvolvedor do pacote, o contatinho dele (o EMAIL ok) e também as dependências de compilação. É importante prestar atenção nele, principalmente se você quer trabalhar junto com outras pessoas da comunidade ROS.

O segundo arquivo é o famoso *"CMakeLists.txt"*. Se você já mexeu com CMake, já deve ter visto algo sobre ele. Em linhas gerais, ele é responsável por indicar ao compilador quais são os executáveis (os Nós) que serão criados e também quais são as dependências desses programas. Tem bastante coisa comentada nele e você pode ler com cuidado, se quiser. Quando chegar a hora, iremos mexer nele juntos. Ah, além disso, você conseguirá ver também as dependências que inseriu na hora de criar o pacote já explícitas nesse arquivo.

Criando um Nó tagarela (Publicador)

Como vimos, um nó é, em última análise, um código que é compilado e se torna um executável. Para criar um, vamos primeiro fazer o arquivo fonte. Aqui, vou fazer o passo a passo em C++ (é o que estou mais acostumado) - insiro depois um exemplo em Python.

Bom, mãos à obra! Crie um arquivo chamado *"publisher.cpp"* dentro da pasta *"src"* que fica dentro do pacote recém criado. A primeira coisa que precisamos adicionar é a interface ROS para C++. Para isso, é só inserir o arquivo *header* logo no começo do código.

```
#include <ros/ros.h>
```

Depois disso, vamos inserir a rotina principal do nosso código, a famosa "*main":*

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

}
```

Dentro dela, vamos primeiro inicializar o nosso nó ROS. Para isso, utilizaremos a linha de código a seguir.

```
ros::init(argc, argv, "publisherNode");
```

Vale ressaltar duas coisas. A primeira delas é o trecho *"ros::"* - ele define o *namespace* de onde vem aquele comando. Resumindo num tweet: o mesmo nome de método/função pode estar definida em mais de uma biblioteca. Logo, nós explicitamente indicamos que queremos executar a função *"init"* da interface do ROS colocando o *namespace* desejado. É possível definir que um certo *namespace* será utilizado ao longo de um código, mas, no geral, não é uma boa prática (e eu não gosto rs), então tudo aqui terá *namespace* explícito. A segunda delas é a própria estrutura do comando: ele recebe os argumentos vindos da main e também uma *string*, que é o nome que você batiza o Nó em questão. Em seguida, vamos construir nosso *NodeHandle*.

```
ros::NodeHandle _nh;
```

Esse objeto é, como o próprio nome diz, uma estrutura especial que possibilita a manipulação do Nó. Já vamos usar ele para algumas coisinhas, fica de olho.

Agora, vamos construir o publicador. Para isso, utilizaremos o comando a seguir.

```
ros::Publisher topicoExemploRef =
_nh.advertise<std_msgs::Int64>("topicoExemplo", 1);
```

Vamos por partes. Em `ros::Publisher topicoExemploRef`, nós estamos declarando uma variável que armazena a referência para o publicador. Para construí-lo de fato, utilizamos o objeto do NodeHandle, chamando o método `advertise`. Aí vem uma parte importante: nós estamos criando o carteiro, mas precisamos indicar qual é o tipo de carga que ele vai entregar. Nesse caso, queremos que a Mensagem que seja transportada pelo Tópico seja um número inteiro, de 64 bits. Logo, utilizaremos o trecho `<std_msgs::Int64>` para indicar isso. Escrever isso provavelmente vai indicar um erro - já explico o porquê. Cotinuando, o método `advertise` recebe dois parâmetros: o primeiro é o nome do tópico, batizado aqui de `"topicoExemplo"` e o segundo é o tamanho da fila. Fila? É, fila. A ideia é que, dependendo de como o processo estiver executando (e até mesmo de como o código está escrito), talvez ele não consiga publicar na frequência que você determinou. Pensando nisso, você pode escolher quantas mensagens podem ficar "armazenadas" sem serem entregues, esperando que o computador realize esse procedimento. Elas entram nessa fila, sendo enviadas assim que possível/necessário. Para os propósitos deste tutorial, usaremos o valor 1 para o tamanho da fila - não queremos mensagens acumuladas.

Bom, agora temos que corrigir o erro, né? Nós utilizamos uma mensagem específica - a "Int64" - sem antes indicar a referência dela na API de ROS para C++. Teu código tá tipo: *"mas que raios de classe é essa que eu não sei onde tá?"*. Para resolver isso é fácil. A mensagem em questão faz parte do conjunto de mensagens padrão (*namespace* chamado de "*std_msgs*") - basta, portanto, importarmos ela ao código. Logo em baixo do primeiro *include*, insira

```
#include "std_msgs/Int64.h"
```

Perfeito. Voltando, agora precisamos instanciar a mensagem em si que o código vai enviar. Para isso, utilizamos

```
std_msgs::Int64 mensagem;
```

Tá, mas essa mensagem não tem informação nenhuma ainda. Para modificar seu conteúdo, utilizamos a expressão a seguir.

```
mensagem.data = 200;
```

Bem simples, né? É como uma *struct* mesmo (mas não exatamente igual). Existem mensagens com mais informações, como data/hora e um dado de um sensor, por exemplo. Ou ainda, mensagens com três números, cada um indicando uma coordenada de um espaço 3D. As possibilidades são literalmente infinitas. Se você utilizar uma Mensagem pronta, consulte a documentação para saber como acessar cada campo específico. Se você construir sua própria mensagem (ensino isso mais pra frente), bom, aí você também vai saber como acessá-la. E ah, sim! O número 200 é completamente arbitrário. Fica à vontade para escolher outro número.

Beleza! O que já temos? Temos um Nó, com um Publicador criado e também uma Mensagem a ser publicada. Precisamos agora fazer uma última coisa: mandar o Nó publicar! Sim, isso não é automático, já que você poderia por exemplo mudar o valor da mensagem lendo informações de um sensor ou fazendo alguma operação matemática. Nos nossos propósitos de tutorial, vamos criar um *loop* infinito (ou quase). Queremos que o nosso Nó publique até que todo o sistema ROS seja desligado. Para isso, utilizaremos a seguinte estrutura

```
while(ros::ok())
{

}
```

O "*while*" você conhece. O termo `ros::ok()` é uma função da interface ROS que retorna um valor booleano (verdadeiro ou falso). Esse valor depende se o *roscore* está funcionando ou não. Se sim, retorna verdadeiro. Se não, falso. Logo, o código ficará "preso" nesse *while* até que você desligue o sistema. Dentro do *loop*, vamos inserir o comando de publicar.

```
topicoExemploRef.publish(mensagem);
```

Estamos utilizando o método `publish` do objeto `topicoExemploRef` (o Publicador) para enviar a Mensagem que criamos anteriormente. Perceba: o objeto da Mensagem é o mesmo, mas podemos trocar a informação dele a qualquer momento. Por fim, vamos utilizar um comando importante (dentro do *loop*).

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

O método `spinOnce()` é responsável por executar todas as rotinas deste Nó que dependem de algum *trigger*. Isso vai ser meio difícil de explicar agora, então, vamos aceitar por enquanto e voltamos nisso depois, tudo bem?

Pronto! Temos nosso código publicador criado bonitinho! O código completo está abaixo.

```
#include "ros/ros.h"
#include "std_msgs/Int64.h"

#include <iostream>

int main(int argc, char **argv)
{
    ros::init(argc, argv, "publisherNode");
    ros::NodeHandle _nh;

    ros::Publisher topicoExemploRef =
_nh.advertise<std_msgs::Int64>("topicoExemplo", 1);

    std_msgs::Int64 mensagem;
    mensagem.data = 200;

    while(ros::ok())
    {
        topicoExemploRef.publish(mensagem);
        ros::spinOnce();
    }
}
```

Criando um Nó do relacionamento saudável (o que ouve, o Subscritor)

Bom, agora vamos ao segundo Nó - o que vai receber as informações do primeiro. Algumas coisas eu já expliquei para o primeiro nó, então vamos partir de um começo menos cru.

```
#include "ros/ros.h"
#include "std_msgs/Int64.h"

int main(int argc, char **argv)
{
    ros::init(argc, argv, "subscriberNode");
    ros::NodeHandle _nh;
}
```

Do primeiro nó, a única diferença até agora é o nome que definimos para ele no método `ros::init`.

Qual é nosso objetivo? Esse nó vai receber as informações do primeiro e vai somente imprimir a informação no terminal para nós. Bem simples, mas é por isso que é um *Hello World*, né? Primeiro de tudo é que vamos ter que incluir uma nova biblioteca para mostrar o dado na tela. Para isso, vamos inserir (antes da *main*)

```
#include <iostream>
```

Eu espero que você já esteja familiarizado com essa biblioteca. Se não, ela é bem utilizada para realizar entrada e saída em um código em C++.

Em seguida, vamos construir nosso subscritor. É BEM parecido com o do Publicador, saca só:

```
ros::Subscriber topicoExemploRef = _nh.subscribe("topicoExemplo",
1, subscriberCallback);
```

Bom, em `ros::Subscriber topicoExemploRef`, *criamos a variável que armazena a referência do Subscritor.* O método `subscribe` do NodeHandle é quem constrói o subscritor. Ele precisa de três parâmetros. O primeiro deles é o nome do tópico que estará enviando as mensagens; o segundo é a quantidade de mensagens que podem ficar na fila simultaneamente, antes de serem tratadas; o terceiro é o nome da função de *callback*, chamada toda vez que uma mensagem chega pelo Tópico.

Bom, vamos com calma. Sobre o primeiro, é muito importante que o nome seja idêntico àquele utilizado quando você instanciou o Publicador. Sobre o segundo, é o mesmo princípio da fila do Publicador, não tem mais segredo. O terceiro é novidade. Um Subscritor funciona de forma que, toda vez que uma Mensagem chega, ele chama uma função específica (a função de *"callback"*), passando para ela como parâmetro a

mensagem em questão. Um detalhe importante é que essa função NUNCA retorna nenhuma informação - portanto, ela é do *"tipo void"*.

Precisamos agora criá-la, com o cabeçalho a seguir - ela vai antes da *main* no código, belê?

```
void subscriberCallback(const std_msgs::Int64::ConstPtr& msg)
{

}
```

Bom, a única coisa estranha aqui é o parâmetro `const std_msgs::Int64::ConstPtr& msg`. Bom, vamos por partes. O termo `const` é uma palavra reservada da linguagem C++ que indica que o parâmetro não pode ser alterado. Isso faz sentido, já que não é interessante alterarmos a mensagem que está chegando, mas copiá-la para uma outra variável ou utilizá-la em algum cálculo. O trecho `std_msgs::Int64::ConstPtr&` indica que estamos recebendo nesta função um ponteiro "constante" (ou seja, que não modificaremos) para a mensagem recebida. E ah, `msg` é o nome que estamos dando para o parâmetro para referenciarmos dentro da função. Bem tranquilo, né?

Dentro da função de *"callback"*, vamos imprimir a informação que recebemos na mensagem para o usuário (no caso, nós mesmos). Para isso, usaremos o seguinte:

```
std::cout << "[subscriberNode] Valor recebido: " << msg->data <<
std::endl;
```

Considerando que você tenha alguma experiência em C++, não tem nada muito estranho. O único detalhe particular de ROS aqui é o termo `msg->data` , que acessa o ponteiro e pega a informação, de acordo com a estrutura da Mensagem *"Int64"*.

Falta um último detalhe. No final da *main*, precisamos adicionar uma última linha de código.

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

Olha só! Não parece muito aquela última função que inserimos no Publicador? Não só parece como tem uma função bem semelhante. O método `spin` é responsável por colocar o nó em um *loop* infinito, igual aquele que fizemos com `while` no

Publicador. Ele automaticamente mantém o Subscritor funcionando, executando a função de "*callback*" toda vez que uma mensagem chega. Basicamente, ele mantém o nosso Nó vivo, apenas atendendo os procedimentos que funcionam com *triggers*.

Mas aí você pergunta: "Poxa, Toschi, mas por que raios então a gente não usou isso no Publicador?". Ótima pergunta. No código do Publicador, é interessante termos o *loop* infinito explícito, já que podemos alterar o conteúdo da mensagem antes de publicá-la a cada iteração. Para isso, usamos o `spinOnce()`, que executa as funções com *trigger* a cada iteração do *loop*, mas não cria o *loop* propriamente dito (o fizemos explícito no `while`). No nó do Subscritor, essa necessidade não se faz tão presente para os fins deste tutorial - se tivessem mais coisas a serem feitas, talvez fosse necessário também. Existem algumas nuances nesse processo, mas podemos deixar para falar disso depois.

Teeeerminamos mais um! O código completo fica assim ó:

```
#include "ros/ros.h"
#include "std_msgs/Int64.h"

#include <iostream>

void subscriberCallback(const std_msgs::Int64::ConstPtr& msg)
{
    std::cout << "[subscriberNode] Valor recebido: " << msg->data
<< std::endl;
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "subscriberNode");
    ros::NodeHandle _nh;

    ros::Subscriber topicoExemploRef =
_nh.subscribe("topicoExemplo", 1, subscriberCallback);

    ros::spin();
}
```

Lidando com a burocracia (CMakeLists.txt)

Bom, fizemos a parte legal. Agora, falta ajustarmos alguns detalhes. Precisamos indicar ao compilador onde e como chamam nossos arquivos. Para isso, utilizaremos o arquivo *"CMakeLists.txt"* que está dentro do nosso pacote *"helloWorld"*. Olha, tem MUITA configuração nesse arquivo e nem eu sei tudo o que ele pode fazer. Mas, estamos atrás de duas coisas bem específicas.

A primeira delas é a declaração `add_executable`. Originalmente, já existe um modelo que você pode copiar e colar comentado no documento. Dê CTRL+F e procure o termo em questão. Abaixo, dele, insira

```
add_executable(publisherNode
  src/publisher.cpp
)

add_executable(subscriberNode
  src/subscriber.cpp
)
```

Vamos adicionar dois executáveis. O primeiro parâmetro é o nome do Nó (não precisa colocar entre aspas). O segundo parâmetro é o caminho relativo para o arquivo fonte do Nó em questão. Eles estão dentro da pasta *"src"*, como indicado.

Logo em seguida, precisamos mostrar para o compilador que é necessário *linkar* as bibliotecas do ROS em cada nó. Podemos procurar a declaração `target_link_libraries`. Existe também o modelo para ele comentado no documento. Abaixo do comentário, insira

```
target_link_libraries(publisherNode
  ${catkin_LIBRARIES}
)

target_link_libraries(subscriberNode
  ${catkin_LIBRARIES}
)
```

Perceba que o primeiro parâmetro é o nome do nó, que colocamos na declaração do executável. Ambos têm de ser idênticos. O segundo parâmetro é uma referência às bibliotecas necessárias do ROS. Para TODO Nó que você criar, você precisa fazer NO

MÍNIMO esse processo, ok? Talvez tenha que fazer mais alterações, dependendo das necessidades - mas o mínimo é esse.

Para compilar tudo direitinho, precisamos voltar na pasta *"catkin_ws"* (o nosso *workspace*) e executar no terminal o comando `catkin_make`. Se tudo der certo, o processo de compilação vai terminar tranquilamente. Se der um erro, de duas uma: ou você fez alguma coisa errada ou eu ensinei alguma coisa errada. Acontece. De toda forma, existem fóruns de ROS que podem te ajudar a tirar dúvidas ou até mesmo você pode entrar em contato comigo! Allons-y!

Funciona, fi! (arquivo *"launch"*)

Nós poderíamos executar os nós individualmente, mas isso não é muito prático. Imagine um sistema com 10 Nós diferentes. Você precisaria abrir 10 terminais e executar 10 comandos para fazer tudo funcionar. Para evitar esse problema, existem os arquivos *launch*.

Já adianto, eles não são complicados. Eu gosto de criar uma pasta dentro da pasta do pacote, chamada de *"launch"*. Dentro dela, criaremos um arquivo chamado *"helloWorld.launch"*. Agora, vamos editá-lo.

Todo arquivo *.launch* começa com duas diretrizes:

```
<launch>

</launch>
```

É como se fossem chaves. Eles abrem e fecham o espaço do arquivo em questão. Dentro deles, vamos inserir quais Nós queremos que sejam executados simultaneamente. (?)

```
<node name="publisherNode" pkg="helloWorld" type="publisherNode"
output="screen"/>

<node name="subscriberNode" pkg="helloWorld" type="subscriberNode"
output="screen"/>
```

Feito isso, tá pronto! Rápido, né? Agora, só precisamos executar. Pra isso, podemos abrir um terminal, entrar na pasta do workspace (*"catkin_ws"*) e digitar o seguinte comando

```
roslaunch src/helloWorld/launch/helloWorld.launch
```

O comando roslaunch serve para executar o sistema, uma vez que ele já está compilado (importante!). O *roscore* abrirá imediatamente e você deve ver a saída do sistema como uma repetição infinita de

"[subscriberNode] Valor recebido: 200"

Ah! Se você receber um erro logo de cara ao executar o *roslaunch*, pode ser que você tenha esquecido de executar `source devel/setup.sh` no terminal atual, como eu ensinei lá atrás. Para finalizar o sistema, basta pressionar CTRL + C no terminal.

Parabéns! Você sabe o feijão com arroz de ROS! :)

Complicando um pouquinho

Você aprendeu o básico do *modus operandi* em ROS. O processo de criação de nós e sua estrutura básica será repetido várias vezes durante o desenvolvimento de um sistema, vai se acostumando! Bom, agora podemos complicar mais um tequinho, *shall we?*

Customizando Mensagens

Construindo a definição

Fazendo o *Hello World,* nós utilizamos uma mensagem padrão do ROS, que faz parte do conjunto das *std_msgs* - tá lembrado? No entanto, nem sempre vamos ficar satisfeitos com as mensagens padrão. Na verdade, isso é bem difícil de acontecer - as prontas são bem completas. No entanto, por experiência própria, saber fazer isso já me ajudou em algumas situações, deixando o sistema e o código mais legível e elegante. Então, tô considerando valer a pena!

Para começar, vamos criar uma pasta dentro daquela do pacote chamada de *"msg"*. Nela, armazenaremos todas as definições das novas mensagens. Como eu já citei antes, elas são *structs*, ou seja, temos que definir de quais componentes básicos ela é formada. Existem alguns tipos "básicos":

- Os tipos primitivos, que incluem:
  - ***bool***
    - um booleano, que pode ser "verdadeiro" ou "falso"

- **int8, uint8, int16, uint16, int32, uint32, int64, uint64**
  - "int" é um número inteiro, como você está acostumado. O número em seguida indica o número de bits que ele é composto (considerando um bit para sinal). Exemplificando, um int de 8 bits pode armazenar valores de -127 a 127.
  - "uint" significa um número inteiro sem sinal (*"unsigned"*). Isso significa que você pode aproveitar mais um bit caso você não se importe com positivo/negativo. Exemplificando, um "uint" de 8 bits pode armazenar valores de 0 a 255.
- **float32, float64**
  - "float" é um "número flutuante", ou seja, com vírgula. Ele pode ser de 32 ou 64 bits.
- **string**
  - "string" é um conjunto de caracteres, um texto.
- **time**
  - variável especial para contagem de tempo - é expressa como inteiros de 32 bits.
- **duration**
  - variável especial para contagem de intervalos de tempo - é expressa como inteiros de 32 bits.

- Mensagens previamente construídas (pode ser de pacotes prontos ou outras mensagens que você mesmo tenha criado)
  - Por exemplo, existe um conjunto de mensagens bem útil chamado de *geometry_msgs*. Nele, você consegue encontrar mensagens que se referem a informações geométricas (posição, vetores, aceleração, velocidade).
  - Você poderia, por exemplo, construir uma mensagem que possui duas posições no R2. Para isso, você pode usar a referência "geometry_msgs/Pose2D" na definição da nova mensagem
- Vetores de tamanho variável ou fixo com os tipos citados acima
  - Você pode querer armazenar um vetor de inteiros de 8 bits (com sinal). Para fazer isso, pode usar "int8[]" (tamanho variável) ou "int8[10]" (tamanho fixo de 10 posições)
- Tipo especial chamado de Cabeçalho ("Header")
  - Ele é uma mensagem previamente construída (do conjunto das *"std_msgs"*), mas você não precisa identificar o conjunto como as demais (é exceção à regra)

- **Ela contém informações de tempo (atreladas a uma outra informação) com precisão de segundos e nanosegundos. Possui também um espaço para observações sobre aquela informação em específico. Muito útil quando você precisa *"gravar"* informações que serão utilizadas posteriormente.**

Vale ressaltar que, apesar de eu estar entregando muita coisa mastigada nesse tutorial, você ainda é um programador que precisa se virar. Dado isso, pesquise sobre as mensagens e utilize a referência oficial do ROS para saber em detalhes a estrutura interna de cada uma - ajuda MUITO!

Certo, mão na massa. Crie um arquivo chamado de *"MinhaMensagem.msg"* dentro da pasta *"msg"*. Dentro do arquivo, precisamos inserir só o tipo e o nome de cada componente da mensagem. É legal também sempre deixar comentários sobre o objetivo da mensagem e de cada componente. Assim, vamos inserir inicialmente comentários gerais.

```
#Mensagem de teste para o Tutorial de ROS do Toschi
#O objetivo da vida (e dessa mensagem) é ser feliz
```

Depois disso, vamos inserir dois inteiros aleatórios e um texto. Vamos tentar fazer alguns testes: utilizaremos um Int64 da *std_msgs* e um int64 como tipo primitivo. Além disso, utilizaremos o texto como uma string do tipo primitivo também. Assim,

```
int32 primeiroInteiro
#Inteiro de tipo primitivo

std_msgs/Int32 segundoInteiro
#Inteiro usando uma mensagem pré-pronta

string texto
#Texto de tipo primitivo
```

E, pronto! Isso deve ser suficiente para definir a Mensagem que estamos criando!

Lidando com a burocracia (de novo CMakeLists.txt)

Precisamos avisar o compilador que queremos uma nova mensagem. Pensa comigo: como o código em C++ interpreta a estrutura da mensagem? Precisa existir alguma forma de definição da *struct* em algum lugar - nesse caso, não existe magia. O próprio compilador nos ajuda criando o arquivo de definição, só precisamos configurar ele direitinho. Para isso, voltemos ao CMakeLists.

Primeiro, precisamos adicionar uma nova dependência no nosso projeto. Para isso, pesquise o termo `find_package` e, na lista já existente, insira no final o pacote `message_generation`. Fica assim:

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

Depois, precisamos indicar qual é o nome do arquivo da mensagem que queremos criar. Pesquise `add_message_files` e descomente todo o bloco. Apague todos os nomes de mensagens genéricas e mantenha o termo `FILES`. Logo abaixo dele, insira o nome da mensagem que você quer gerar - no nosso caso aqui, *"MinhaMensagem.msg"*.

```
add_message_files(
  FILES
  MinhaMensagem.msg)
```

Precisamos ainda procurar o termo `generate_messages` e também descomentá-lo. Apague todo o resto e mantenha (ou insira) o termo `DEPENDENCIES`. Logo abaixo dele, insira o nome das dependências das mensagens que serão geradas. No nosso caso, além dos tipos primitivos, usamos o conjunto *"std_msgs"*. Assim, temos que o inserir neste campo para poder utilizá-lo.

```
generate_messages(
  DEPENDENCIES
  std_msgs)
```

Ainda não acabou! Precisamos atualizar também o `catkin_package` (pesquise ele no documento). Ele é o pacote responsável por gerar os arquivos de compilação. Descomente todo o bloco e adicione a dependência `message_runtime` na linha que se inicia com `CATKIN_DEPENDS`. No final, é para ser algo parecido com

```
catkin_package(
  INCLUDE_DIRS include
  LIBRARIES helloWorld
  CATKIN_DEPENDS roscpp rospy std_msgs message_runtime
  DEPENDS system_lib
)
```

Pronto! Agora o nosso CMakeLists.txt já está configurado para gerar a nova mensagem.

Mais um teco de burocracia (arquivo package.xml)

Tem mais um detalhezinho para arrumar. Lembra que o arquivo *"package.xml"* serve para descrever quais são as dependências necessárias para que o nosso pacote funcione? Então, a *"message_runtime"* precisa aparecer nesse arquivo, se não seu *workspace* nem compila.

Mas é rápido! Você só precisa adicionar três linhas, indicando uma nova dependência. Quase no final do arquivo, você vai encontrar algumas *tags* escritas como

```
<build_depend></build_depend>

<build_export_depend></build_export_depend>

<exec_depend></exec_depend>
```

Iremos inserir as três (nos lugares pertinho de onde as correspondentes estão), colocando dentro da *tag* o nome da dependência, *"message_runtime"*. O bloco inteiro é para ficar mais ou menos assim:

```
<buildtool_depend>catkin</buildtool_depend>
<build_depend>roscpp</build_depend>
<build_depend>rospy</build_depend>
<build_depend>std_msgs</build_depend>
```

```
<build_depend>message_runtime</build_depend>

<build_export_depend>roscpp</build_export_depend>
<build_export_depend>rospy</build_export_depend>
<build_export_depend>std_msgs</build_export_depend>
<build_export_depend>message_runtime</build_export_depend>

<exec_depend>roscpp</exec_depend>
<exec_depend>rospy</exec_depend>
<exec_depend>std_msgs</exec_depend>
<exec_depend>message_runtime</exec_depend>
```

Agora sim! Acabou a burocracia (finalmente). Vá para a pasta do seu *workspace* e execute o comando *catkin_make*, para que os arquivos sejam criados e configurados direitinho e você possa utilizá-los em algum código.

Modificando o Hello World para inserir a mensagem criada

Agora que temos a mensagem criada, seria legal que nós tentássemos utilizá-la em um código, certo? Bom, vamos modificar nosso *Hello World* para que ele possa utilizar a nova mensagem. Para isso, é bem simples. Primeiro de tudo, vamos retormar o código do Publicador.

Voltando ao publicador

O código final era esse aqui:

```
#include "ros/ros.h"
#include "std_msgs/Int64.h"

#include <iostream>

int main(int argc, char **argv)
{
    ros::init(argc, argv, "publisherNode");
    ros::NodeHandle _nh;

ros::Publisher topicoExemploRef =
_nh.advertise<std_msgs::Int64>("topicoExemplo", 1);
```

```
    std_msgs::Int64 mensagem;
    mensagem.data = 200;

    while(ros::ok())
    {
        topicoExemploRef.publish(mensagem);
        ros::spinOnce();
    }
}
```

Bom, nossa mensagem vai ser outra agora, certo? Logo, podemos retirar a linha que inclui a *std_msgs/Int64*. No lugar dela, vamos inserir o seguinte

```
#include "helloWorld/MinhaMensagem.h"
```

O *namespace* da mensagem que criamos é o nome do pacote. Assim, podemos substituir a linha de declaração do Publicador, fazendo com que o *"topicoExemplo"* não tenha mais a *std_msgs/Int64*, mas sim a *helloWorld/MinhaMensagem*.

```
ros::Publisher topicoExemploRef =
_nh.advertise<helloWorld::MinhaMensagem>("topicoExemplo", 1);
```

Depois disso, basta declararmos o conteúdo da mensagem que queremos publicar. Logo, vamos substituir a linha

```
std_msgs::Int64 mensagem;
```

Pela linha

```
helloWorld::MinhaMensagem mensagem;
```

Agora, precisamos acessar cada um dos campos da nossa mensagem. Para isso, utilizamos os nomes definidos lá no arquivo *MinhaMensagem.msg*. Assim,

```
mensagem.primeiroInteiro = 200;
mensagem.segundoInteiro.data = 300;
mensagem.texto = std::string("IAAAAAAAAAAAAAAAA, disse o
Tarzan");
```

Vale ressaltar alguns pontos. O primeiro deles é que *primeiroInteiro* foi declarado como um tipo primitivo de cara. Logo, você pode acessar ele diretamente. No caso do *segundoInteiro*, ele é uma mensagem pré-pronta, a *std_msgs/Int64.* Logo, temos que verificar na definição desta mensagem qual é o nome do campo que podemos acessar diretamente - no caso, *"data"*.

O terceiro ponto é que o campo *string* de tipo primitivo do ROS corresponde ao objeto *std::string* em C++. Logo, o instanciamos para conseguir armazená-lo na mensagem. Precisamos, inclusive, incluir a biblioteca que contém essa ferramenta. Logo, lá em cima, colocaremos

```
#include <string>
```

Caso você não tenha familiaridade com a *std::string*, eu recomendo pesquisar um pouquinho sobre ela - é bem útil!

Pronto! Nossa mensagem já está sendo publicada normalmente! O código final ficou assim:

```
#include "ros/ros.h"
#include "helloWorld/MinhaMensagem.h"

#include <iostream>
#include <string>

int main(int argc, char **argv)
{
    ros::init(argc, argv, "publisherNode");
    ros::NodeHandle _nh;

    ros::Publisher topicoExemploRef =
_nh.advertise<helloWorld::MinhaMensagem>("topicoExemplo", 1);

    helloWorld::MinhaMensagem mensagem;
```

```
    mensagem.primeiroInteiro = 200;
    mensagem.segundoInteiro.data = 300;
    mensagem.texto = std::string("IAAAAAAAAAAAAAAAAA, disse o
Tarzan");

    while(ros::ok())
    {
        topicoExemploRef.publish(mensagem);
        ros::spinOnce();
    }
}
```

Voltando ao subscritor

O código original era:

```
#include "ros/ros.h"
#include "std_msgs/Int64.h"

#include <iostream>

void subscriberCallback(const std_msgs::Int64::ConstPtr& msg)
{
    std::cout << "[subscriberNode] Valor recebido: " << msg->data
<< std::endl;
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "subscriberNode");
    ros::NodeHandle _nh;

    ros::Subscriber topicoExemploRef =
_nh.subscribe("topicoExemplo", 1, subscriberCallback);

    ros::spin();
}
```

Semelhante ao que fizemos no publicador, vamos mudar o *"#include "std_msgs/Int64.h""* por

```
#include "helloWorld/MinhaMensagem.h"
```

Além disso, vamos adicionar a biblioteca *string* como anteriormente,

```
#include <string>
```

Agora, precisamos mudar o argumento recebido pela função *subscriberCallback*. Ficará assim:

```
void subscriberCallback(const helloWorld::MinhaMensagem::ConstPtr&
msg)
```

Perceba que o compilador criou todas as estruturas necessárias para utilizarmos a mensagem em C++. Eu particularmente acho isso MUITO brabo. Enfim. Agora vamos mudar a forma como mostramos as informações na tela. Dentro da função de *callback* podemos colocar:

```
std::cout << "[subscriberNode] primeiroInteiro: " <<
msg->primeiroInteiro
          << ", segundoInteiro: " << msg->segundoInteiro.data
          << ", texto: " << msg->texto
          << std::endl;
```

Fácil, fácil! O código todo ficou assim:

```
#include "ros/ros.h"
#include "helloWorld/MinhaMensagem.h"

#include <iostream>
#include <string>

void subscriberCallback(const helloWorld::MinhaMensagem::ConstPtr&
msg)
```

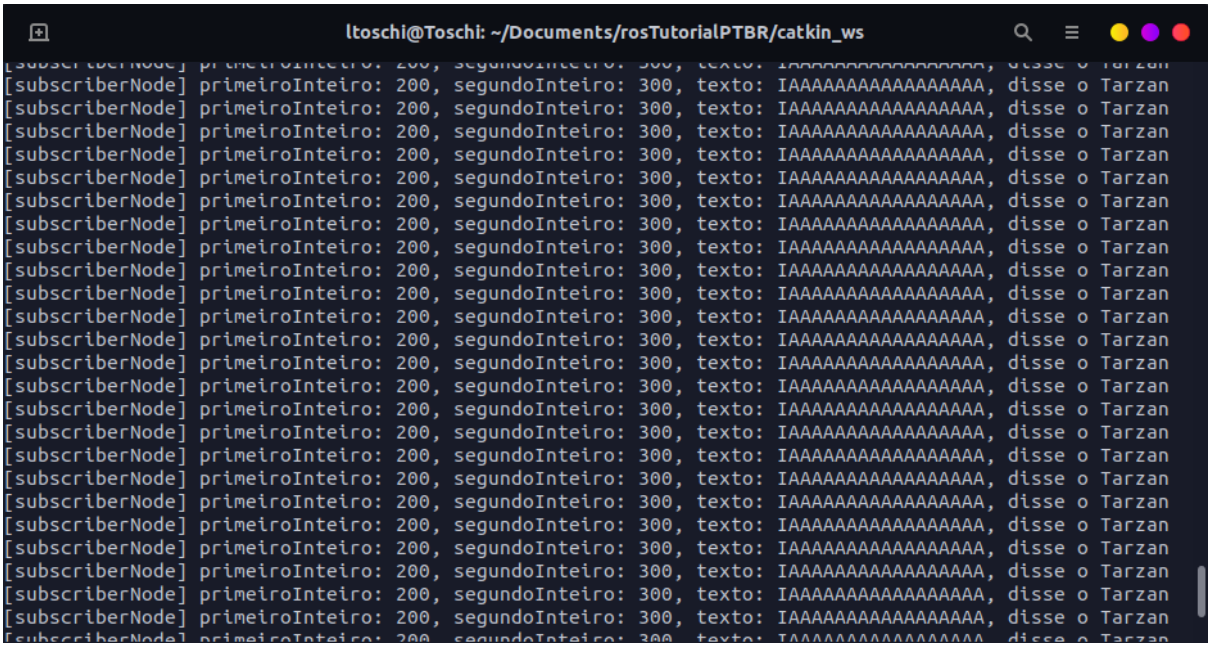
```
{
    std::cout << "[subscriberNode] primeiroInteiro: " <<
msg->primeiroInteiro
              << ", segundoInteiro: " << msg->segundoInteiro.data
              << ", texto: " << msg->texto;
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "subscriberNode");
    ros::NodeHandle _nh;

    ros::Subscriber topicoExemploRef =
_nh.subscribe("topicoExemplo", 1, subscriberCallback);

    ros::spin();
}
```

Pronto! Só executar o arquivo *launch* como antes e deve funcionar bem! Você deve ver mais ou menos isso aqui:



Utilizando comandos ROS no terminal

Durante o treinamento, utilizamos alguns comandos importantes no terminal. Existem vários que são importantes de saber e agilizam o entendimento do que está acontecendo. Segue o fio:

- **roscd**

Esse comando permite que você acesse a pasta de um determinado pacote só inserindo o nome dele. Entretanto, só funciona se você já utilizado o arquivo *"setup.sh"* no terminal em questão. Se seu pacote se chama *"helloWorld"*, você pode acessá-lo assim:

```
roscd helloWorld
```

- **rosclean**

Limpa os recursos de sistema, liberando espaço. É útil quando você imprimiu na tela publicadores que mandaram muitas e muitas mensagens.

- **roscore**

Esse comando executa o **roscore.** Você pode executá-lo em um terminal e executar Nós em outros terminais (basicamente, fazer o processo que o arquivo *launch* faz automaticamente, após configurado).

- **rosrun**

Esse comando permite que você execute um Nó (já compilado pelo *catkin_make*). Só funciona se você já tiver um **roscore** funcionando. Um exemplo desse comando funcionando está abaixo. O primeiro parâmetro é o nome do pacote e o segundo o nome do Nó, como definido no *CMakeLists.txt* antes da compilação. Você precisa ter executado o arquivo *"setup.sh"* antes de usar esse comando.

```
rosrun helloWorld publisherNode
```

- **catkin_create_pkg**

Cria um novo pacote ROS, com as dependências necessárias. O primeiro parâmetro é o nome do novo pacote e os demais são as dependências. Um exemplo está mostrado abaixo.

```
catkin_create_pkg *NOME_PACOTE* std_msgs rospy roscpp
```

- **roslaunch**

Executa um arquivo *launch*. Precisa ter executado antes o arquivo "*setup.sh*". O único parâmetro é o arquivo *.launch* que será rodado. Um exemplo de utilização está mostrado abaixo.

```
roslaunch src/helloWorld/launch/helloWorld.launch
```

- **rosmsg**

Retorna informações sobre mensagens no geral. Para mais detalhes, execute o comando sem parâmetros. Se precisar se referir à uma Mensagem ou pacote criados por você, é necessário executar o arquivo *setup.sh*.

- **rosnode**

Retorna informações sobre Nós no geral. Para mais detalhes, execute o comando sem parâmetros. É possível listar os Nós ativos e até mesmo verificar o atraso de comunicação.

- **rostopic**

Retorna informações sobre Tópicos no geral. Para mais detalhes, execute o comando sem parâmetros. É possível obter a lista de Tópicos ativos, a frequência de publicação deles, entre outros fatores.

Existe ainda um comando para obter informações sobre Serviços. Como isso ainda não foi abordado neste tutorial, será omitido desta lista, por enquanto.

Ainda a ser adicionado

- **Exemplos em Python**
- **Como utilizar serviços**
- **Como utilizar ações**
- **Utilizando POO com ROS**