Aula 5

ECA419 - Robótica

Aulas Passadas

Na terceira aula, vimos como implementar um nó de uma forma simples (vide exercício “aula3” disponível no repositório do curso no GitHub: https://github.com/Expertinos/curso_ROS_UNIFEI).

Na quarta aula, aprendemos a modelar sistemas baseados em ROS utilizando pacotes, nós, mensagens, tópicos e serviços.

Objetivo

Nesta aula, iremos aprender uma outra forma de se implementar um nó em C++, desta vez, utilizando classes.

Com isso:

- temos um projeto mais organizado;
- podemos utilizar o paradigma de POO em C++ (herança, encapsulamento, polimorfismo, classes, etc.);
- podemos trabalhar com threads;
- outros recursos da linguagem.

Objetivo

Ainda nesta aula, implementaremos **publicadores e assinantes de tópicos** do ROS em C++ com **mensagens padronizadas** na comunidade ROS e **mensagens customizadas**. Assim, a teoria apresentada na aula anterior será colocada em termos práticos.

Em seguida, será proposto um **exercício** para reforçar o aprendizado do aluno em sala de aula utilizando o pacote **rosaria** e o simulador **MobileSim**.

Ao final da aula, será também proposto o **trabalho 1**, o qual irá compor 33,33% da nota final desta disciplina.

Pacote aula5

Baixe a pasta **aula5** que se encontra na pasta **Exercícios** do repositório do curso lá no **GitHub** (https://github.com/Expertinos/curso_ROS_UNIFEI).

Salve esta pasta no diretório do nosso workspace, ou seja, no diretório **~/catkin_ws/src** e, em seguida, compile os pacotes do workspace via QtCreator (Ctrl+B) ou via terminal (Ctrl+Alt+T):

```
cd ~/catkin_ws
catkin_make
roslaunch aula5 aula5.launch name:=<seu nome>
```

Pacote aula5

Vamos analisar o que está acontecendo, antes de qualquer coisa!!!

Utilizando o comando **rosnode list** em um terminal auxiliar, responda: quais nós são processados **imediatamente** após executar o comando anterior (roslaunch aula5 aula5.launch name:=<*seu nome*>)?

E, finalmente, execute novamente o comando **roslaunch aula5 aula5.launch name:=**<*seu nome*> e descreva o que acontece, sem olhar o código. Aguarde até uma nova linha de comando aparecer no terminal.

Pacote aula5

Quais nós são processados após executar o comando anterior?

Resp.: Os nós: **/rosout**, */<seu nome>***/emissor_node** e */<seu nome>***/receptor_node** !!

Execute novamente o comando anterior e descreva o que acontece.

Resp.: **Todos nós se inicializam** (algumas mensagens confirmam isso), após um determinado tempo, o nó */<seu nome>***/emissor_node pára** de ser executado e, alguns instantes depois, o nó */<seu nome>***/receptor_node também pára** de ser executado também. Finalmente, uma nova linha de comando fica disponível no terminal.

Vamos ao código

Após análise do pacote, responda:

Quais são arquivos que **implementam** o nó **emissor_node** do pacote **aula5**?

Quais são arquivos que **implementam** o nó **receptor_node** do pacote **aula5**?

Qual é o tipo de relacionamento entre as classes **Node** e **aula5::EmissorNode**: é-um, tem-um ou usa-um?

Qual é o tipo de relacionamento entre as classes **Node** e **aula5::ReceptorNode**: é-um, tem-um ou usa-um?

…

Questionário para entregar via e-mail no final da aula!!!

Vamos ao código

...

Qual é a função da classe **Node** no modelo de classes?

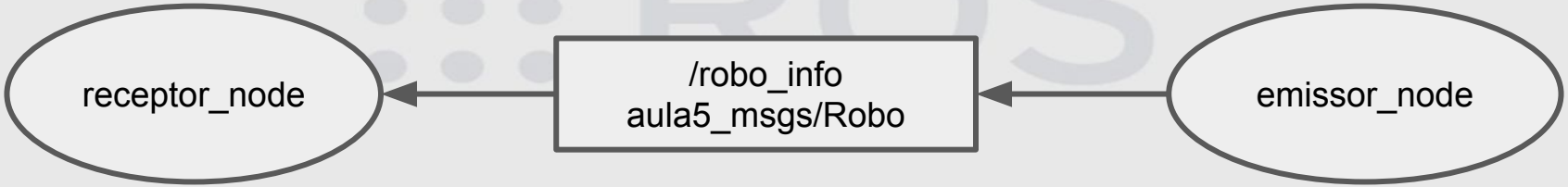
Qual é o papel dos temporizadores nas classes **aula5::EmissorNode** e **aula5::ReceptorNode**?

Qual é a frequência de atualização dos nós **emissor_node** e **receptor_node**?

Após quantos segundos os nós **emissor_node** e **receptor_node** param de executar, respectivamente?

Questionário para entregar via e-mail no final da aula!!!

Modelagem da aula5



Mensagens

O tipo de uma mensagem é definido por um arquivo de extensão **.msg**, o qual define seu conteúdo, e também pelo pacote onde ela foi criada (conforme vimos na aula anterior).

Por exemplo, o arquivo **String.msg** contido no pacote **std_msgs** define o tipo de mensagem **std_msgs/String**. Seu conteúdo é:

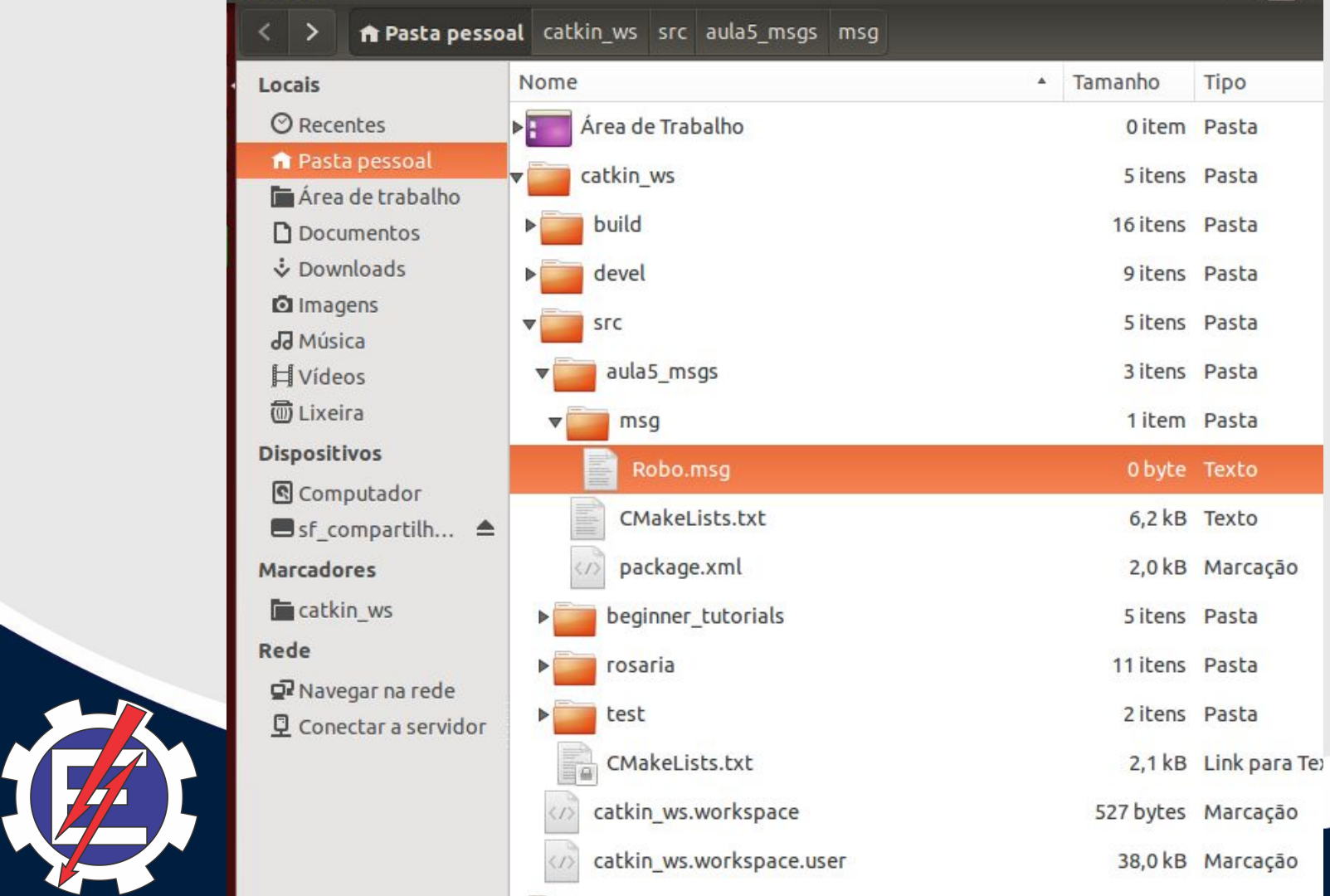
```
string data
```

Ou seja, uma mensagem do **tipo std_msgs/String** transmite apenas uma informação do tipo **string** que pode ser acessada através do seu campo **data**.

Mensagem Customizada

Vamos então à prática:

1) Crie um pacote chamado **aula5_msgs**, cujas dependências serão: **std_msgs**, **message_generation** e **geometry_msgs**;
2) De modo a manter o projeto organizado, crie uma pasta chamada **msg** dentro do pacote **aula5_msgs**;
3) Crie um arquivo de texto dentro da pasta **msg** do pacote **aula5_msgs**;
4) Nomeie o arquivo criado no passo anterior como **Robo.msg**;


Pasta pessoal
catkin_ws
src
aula5_msgs
msg
Locais
Recentes
Pasta pessoal
Área de trabalho
Documentos
Downloads
Imagens
Música
Vídeos
Lixeira
Dispositivos
Computador
sf_compartilh...
Marcadores
catkin_ws
Rede
Navegar na rede
Conectar a servidor
Nome
Tamanho
Tipo
Área de Trabalho
0 item
Pasta
catkin_ws
5 itens
Pasta
build
16 itens
Pasta
devel
9 itens
Pasta
src
5 itens
Pasta
aula5_msgs
3 itens
Pasta
msg
1 item
Pasta
Robo.msg
0 byte
Texto
CMakeLists.txt
6,2 kB
Texto
package.xml
2,0 kB
Marcação
beginner_tutorials
5 itens
Pasta
rosaria
11 itens
Pasta
test
2 itens
Pasta
CMakeLists.txt
2,1 kB
Link para Te
catkin_ws.workspace
527 bytes
Marcação
catkin_ws.workspace.user
38,0 kB
Marcação

Mensagem Customizada

5) Altere o conteúdo do arquivo criado no passo anterior para:

```
Header header                       # info de envio dessa msg
string nome                         # nome do robô
geometry_msgs/Pose2D postura        # posição e orientação do robô
uint8 colisao                 # (1) distante, (2) proximo, (3) muito_proximo
```

Mensagem Customizada

6) Altere o arquivo **package.xml** do pacote **aula5_msgs** apropriadamente:

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

Mensagem Customizada

7) Altere o arquivo **CMakeLists.txt** do pacote **aula5_msgs** apropriadamente:

- find_package(catkin REQUIRED COMPONENTS

  ...

  message_generation)

- catkin_package(

  ...

  CATKIN_DEPENDS message_runtime ...

  ...)

Mensagem Customizada

7) Altere o arquivo **CMakeLists.txt** do pacote **aula5_msgs** apropriadamente:

- add_message_files(
  FILES
  Robo.msg
)
- generate_messages(
  DEPENDENCIES
  std_msgs geometry_msgs
)

Mensagem Customizada

Abra um terminal e analise cada comando:

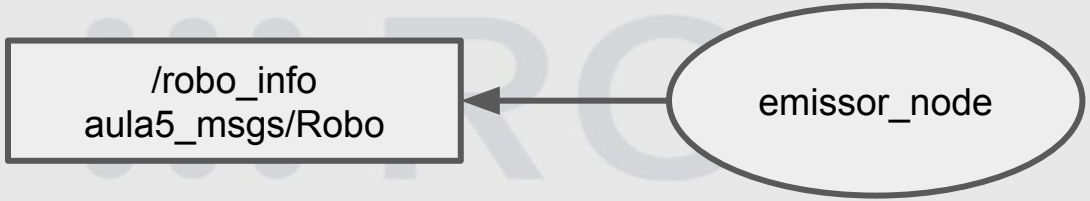
$ rosmsg -h

$ rosmsg list

$ rosmsg show aula5_msgs/Robo

$ rosmsg show Robo

Modelagem da aula5: o publicador



Temos que colocar a **dependência** do pacote **aula5_msgs** no pacote **aula5** para podermos utilizar a **aula5_msgs/Robo** (lá definida) no nosso nó **emissor_node**.

Dependência de aula5_msgs em aula5

Para isso, temos que alterar o arquivo **package.xml** do pacote **aula5**, adicionando a ele as seguintes linhas:

```
<build_depend>aula5_msgs</build_depend>
<run_depend>aula5_msgs</run_depend>
```

Dependência de aula5_msgs em aula5

Para isso, temos que alterar o arquivo **CMakeLists.txt** do pacote **aula5**, modificando os seguintes blocos:

```
find_package(catkin REQUIRED COMPONENTS
      aula5_msgs roscpp
      )
catkin_package(
    INCLUDE_DIRS include
    CATKIN_DEPENDS aula5_msgs roscpp
    )
```

Publicadores de Tópicos no ROS

Primeiramente, temos que incluir o cabeçalho da mensagem:

```
#include <std_msgs/String.h>
```

Construção do objeto, via um objeto do tipo **ros::NodeHandle** (nh):

```
ros::Publisher pub = nh.advertise<std_msgs/String>(“nome_topico”, 8);
```

Publicadores de Tópicos no ROS

Publicação da mensagem, via um objeto **ros::Publisher** (pub):

```
std_msgs/String msg;
msg.data = “O texto que eu desejo publicar.”;
pub.publish(msg);
```

É altamente recomendado desligar o objeto publicador antes dele ser destruído. Para isso, faça:

```
pub.shutdown();
```

Modelagem da aula5: o publicador

Primeiramente, temos que incluir o cabeçalho da mensagem e declarar um membro privado no cabeçalho da nossa classe **aula5::EmissorNode**. Assim, teremos em include/aula5/EmissorNode.h:

```
#include <aula5_msgs/Robo.h> // novo include
namespace aula5 {
    class EmissorNode : public Node {
        // membros públicos e protegidos da classe
    private:
        ros::Publisher robo_info_pub_; // membro novo
        // outros membros privados
    }
}
```

Modelagem da aula5: o publicador

Então, poderemos iniciá-lo no construtor da classe, conforme visto anteriormente. Ou seja, teremos em src/aula5/EmissorNode.cpp:

```
EmissorNode::EmissorNode(ros::NodeHandle *nh)
  : Node(nh, 10)
{
      robo_info_pub_ =
            nh->advertise<aula5_msgs/Robo>("robo_info", 1);
      timer_ = nh->createTimer(ros::Duration(6.75),
            &EmissorNode::timerCallback, this);
}
```

Modelagem da aula5: o publicador

Com isso, poderemos publicar mensagens do tipo **aula5_msgs/Robo** através do membro interno **robo_info_pub_** da classe **EmissorNode** dentro da própria classe, da seguinte forma:

```
aula5_msgs/Robo info_msg;
info_msg.nome = “aqui vem o nome do meu robô!!!”;
info_msg.postura.x = 10.5;
info_msg.postura.y = 9.7;
info_msg.postura.theta = 0;
info_msg.colisao = 2; // porque está bem próximo da parede;
robo_info_pub_.publish(info_msg);
```

Modelagem da aula5: o publicador

Finalmente, temos que nos lembrar de destruir este membro corretamente no destruidor da classe. Para isso, teremos em src/aula5/EmissorNode.cpp:

```
EmissorNode::~EmissorNode()
{
    robo_info_pub_.shutdown(); // linha adicionada no destrutor
}
```

Modelagem da aula5: o assinante

Assinantes de Tópicos do ROS

Construção do objeto, via um objeto do tipo **ros::NodeHandle** (nh):

```
ros::Subscriber sub = nh.subscribe("topic_name", 8, &MyNode::cb, this);
```

Onde a callback **MyNode::cb** é dada por:

```
void MyNode::cb(const std_msgs::String::ConstPtr& msg)
{
    // posso fazer o que eu quiser com msg.data, por exemplo:
    ROS_INFO("Mensagem recebida: %s", msg.data.c_str());
}
```

Assinantes de Tópicos no ROS

É altamente recomendado desligar o objeto assinante antes dele ser destruído. Para isso, faça:

```
sub.shutdown();
```

Modelagem da aula5: o assinante

Primeiramente, temos que incluir o cabeçalho da mensagem e declarar um membro privado no cabeçalho da nossa classe **aula5::ReceptorNode**. Assim, teremos em include/aula5/ReceptorNode.h:

```
#include <aula5_msgs/Robo.h> // novo include
namespace aula5 {
    class ReceptorNode : public Node {
        // membros públicos e protegidos da classe
    private:
        ros::Subscriber robo_info_sub_; // membro novo
        // outros membros privados
    }
}
```

Modelagem da aula5: o assinante

Então, poderemos iniciá-lo no construtor da classe, conforme visto anteriormente. Ou seja, teremos em src/aula5/ReceptorNode.cpp:

```
ReceptorNode::ReceptorNode(ros::NodeHandle *nh)
  : Node(nh, 20)
{
      robo_info_sub_ =
            nh->subscribe("robo_info", 10,
            &ReceptorNode::infoCallback, this);
      timer_ = nh->createTimer(ros::Duration(11.5),
            &ReceptorNode::timerCallback, this);
}
```

Modelagem da aula5: o assinante

Desta forma, sempre que uma nova mensagem do tipo **aula5_msgs/Robo** for publicada por um publicador qualquer através do tópico **robo_info**, o método **ReceptorNode::infoCallback** da classe **aula5::ReceptorNode** será invocada. Com isso, teremos acesso aos dados da mensagem conforme:

Modelagem da aula5: o assinante

```
void ReceptorNode::infoCallback(const aula5_msgs::Robo::ConstPtr& msg)
{
        double x(msg.postura.x), y(msg.postura.y),
    th_deg(msg.postura.theta * 180 / 3.14159);
        std::string colisao("próximo"), nome(msg.nome);
        if  (msg.colisao == 0) {
            colisao = "distante";
        } else if (msg.colisao == 2) {
            colisao = "muito " + colisao;
        }
        ROS_INFO("Infos de %s: (%f [m], %f [m], %f [º])", nome.c_str(), x, y,
        th_deg, colisao.c_str());
}
```

Modelagem da aula5: o assinante

Finalmente, temos que nos lembrar de destruir este membro corretamente no destruidor da classe. Para isso, teremos em src/aula5/ReceptorNode.cpp:

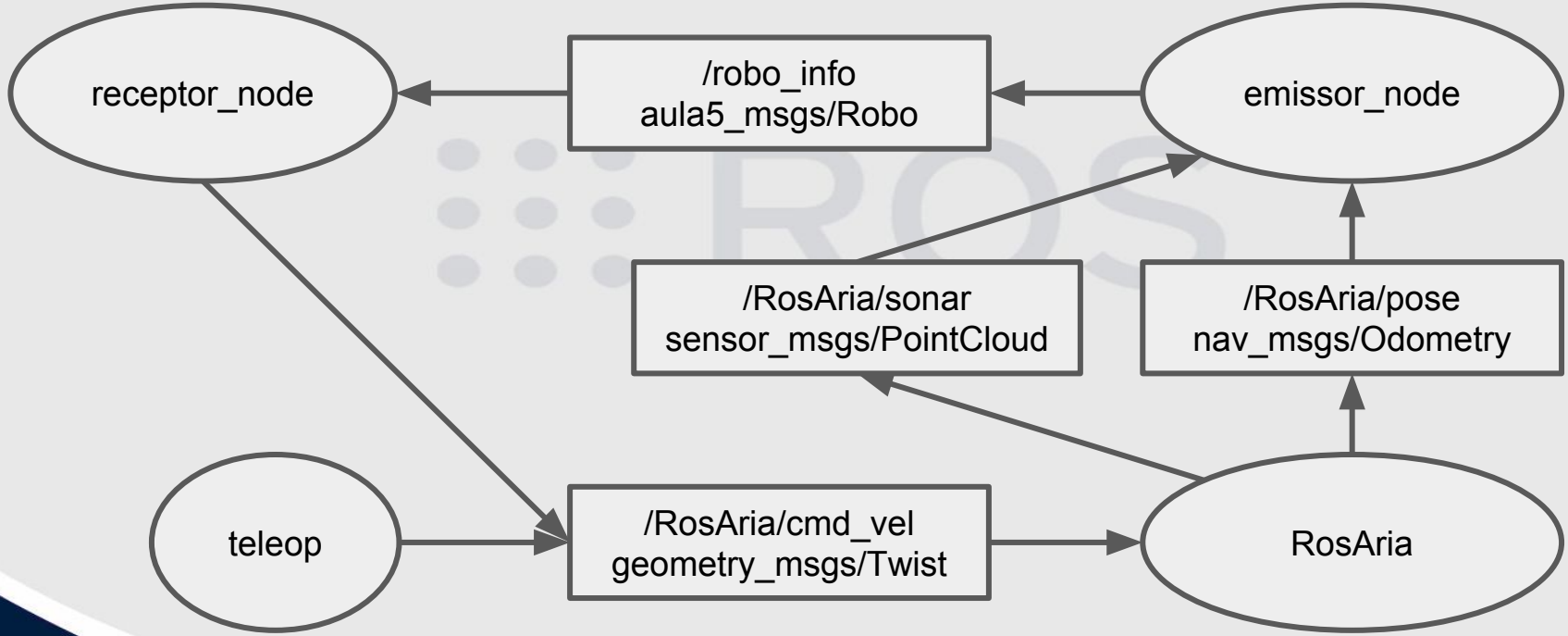
```
ReceptorNode::~ReceptorNode()
{
    robo_info_sub_.shutdown(); // linha adicionada no destrutor
}
```

Testando o modelo da aula5

Vamos testar tudo então.

1) Vamos **compilar** tudo: via QtCreator pressione **Ctrl+B** ou via terminal (**Ctrl+Alt+T**) entre: **cd ~/catkin_ws && catkin_make**
2) Faça as correções necessárias e repita o passo anterior até que não ocorra nenhum problema de compilação;
3) Execute os nós, via QtCreator pressione **Ctrl+R** (se estiver devidamente configurado) ou via terminal entre: **roslaunch aula5 aula5.launch name:=**<*seu nome*>
4) Avalie a execução dos nós usando os comandos **rosnode list**, **rostopic list**, **rostopic info** /<*seu nome*>**/robo_info**, **rosnode echo** /<*seu nome*>**/robo_info** e, finalmente, **rqt_graph**.

Modelagem da aula5 com o rosaria



Agora é com vocês

Identifique:

Quais são os novos publicadores e assinantes no modelo? (detalhe o nome de e o tipo de mensagem transportada por cada um deles)

Em que nó estes objetos estarão localizados?

Agora é com vocês

Agora é só implementar isso nas classes apropriadas, conforme o nó hospedeiro, a natureza do objeto (publicador ou assinante) e tipo de mensagem.
Compile o projeto e, em seguida, execute os nós usando o comando:

roslaunch aula5 rosaria.launch name:=<*seu nome*> **port:=**<*IP do windows*>**:8101**

Finalmente, faça uma análise dos nós em execução!!!
Faça as devidas correções, se for necessário.

Criação de um nó sonar_subscriber

- Abrir o mobilesim;
- Executar: *rosrun rosaria RosAria*
- Analisar os tópicos existentes e focar no tópico dos sonares;
- Analisar o arquivo RosAria.cpp do pacote rosaria;
- Identificar o *publisher* dos sonares;
- Criar um nó para subscrever este tópico.

Criação de um nó cmd_vel_publisher

- Abrir o mobilesim;
- Executar: *rosrun rosaria RosAria*
- Analisar os tópicos existentes e focar no tópico *cmd_vel*;
- Analisar o arquivo RosAria.cpp do pacote rosaria;
- Identificar o *subscriber* do *cmd_vel*;
- Criar um nó para publicar velocidades neste tópico.

Trabalho 1 - Data de Entrega: 20/10/2016

- A partir dos nós criados anteriormente, criar uma aplicação em malha fechada, de tal forma que:
  - Robô deverá inicializar parado e após 2,5 segundos iniciar seu movimento linear;
    - Dicas: http://wiki.ros.org/roscpp/Overview/Time e /cmd_vel.
  - Através da leitura da distância dos sonares:
    - caso o robô esteja próximo de um obstáculo, deverá:
      - informar que existe um obstáculo a uma determinada distância e desviar.
      - É preciso customizar uma mensagem informando a distância e o sentido do desvio (direita ou esquerda).
      - A velocidade deverá ser proporcional à distância do robô ao obstáculo (aplicar PID).
  - Informar também através de mensagem o deslocamento linear do robô.
  - Após 2 minutos a aplicação deverá ser encerrada.
  - Criar variáveis globais para todos os parâmetros a fim de setá-los com maior facilidade (tempo, distância, kp, ki, kd, etc).
  - Fazer o grafo da modelagem utilizando o software dia:
    - https://sourceforge.net/projects/dia-installer/

Entrega do Trabalho 1

Este trabalho poderá ser realizado em dupla!!!

Deverá ser entregue via e-mail (**eca419.unifei@gmail.com**) com todos os pacotes desenvolvidos (**não enviar o pacote rosaria**) com os dados dos alunos envolvidos até o dia **20/10/2016** às **23:51**.

Comentar o código (conforme foi feito na classe Node da aula5), pode ser em inglês ou em português.