Pedro Henrique Di Francia Rosso
Laboratório de Automação e Robótica Móvel
Universidade Federal de Santa Catarina

Documentação Pioneer 3DX

Araranguá
23 de Novembro de 2018

Sumário

1 Introdução

ROS é o sistema utilizado para controle de Robôs como Pioneer 3DX, existem diversos tipos de sistemas utilizados para controle, o ROS é combinado com a Interface ARIA, disponibilizada pela Adept Mobile Robots, a fabricante.

A versão do ROS deve ser de acordo com a versão do Sistema Operacional instalado na máquina. Para este documento, foi utilizada a versão Kinetic Kame, que é compatível com o Ubuntu 16.04.

Os guias para instalação do ROS podem ser consultados na própria wiki do ROS:

Para Instalação no Ubuntu 16.04:
http://wiki.ros.org/kinetic/Installation

Para configurar o ambiente de trabalho:
http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment

O ROS (Robot Operting System) provê bibliotecas e ferramentas que ajudam desenvolvedores de software a criar aplicações robóticas. Ele provém abstrações de hardware, drivers de dispositivos e diversas outras ferramentas que pode auxiliar no desenvolvimento.

ROS pode combinar vários pacotes para o controle de robôs. Neste caso, o ROS já provém um biblioteca prévia para o robô em questão (Pioneer 3DX). O pacote disponibilizado se chama ROSARIA, pode ser baixado neste link: Pacote RosAria. O pacote contém os tópicos básíco para controle do robô Pioneer 3DX.

Os pacotes ROS, baseiam-se em nodos, ou seja, cada um é executado com um processo, e podem comunicar-se entre si. Os nodos funcionam baseados em um núcleo ROS, portanto é essencial que sejam executados juntos com o núcleo.

Este documento, aborda alguns conceitos relacionados ao ROS (*Robot Operating System*, ao ARIA (Biblioteca de Interface fornecida pela *Adept Mobile Robtos*, fabricante do Pioneer 3DX) e pacotes desenvolvidos.

Para desenvolvimento dos códigos para o Pioneer 3DX, para testes, o próprio Pioneer pode ser utilizado, ou antes de utilizá-lo, os testes podem ser feitos utilizando um simulador, sugere-se o MobileSim, disponível no respositório do GitHub do projeto.

2 Robot Operating System

2.1 Instalação

2.1.1 Início

É de grande importância verificar a versão do Sistema Operacional do PC antes de baixar alguma versão do ROS, visto que cada versão do ROS só é compatível com algumas versões de Sistemas Operacionais. Por exemplo, para os casos do OS Ubuntu:

- **ROS Melodic:** Para Ubuntu 17.10 e 18.04.
- **ROS Kinetic Kame:** Para Ubuntu 15.10 e 16.04.
- **ROS Indigo Igloo:** Para Ubuntu 14.04.

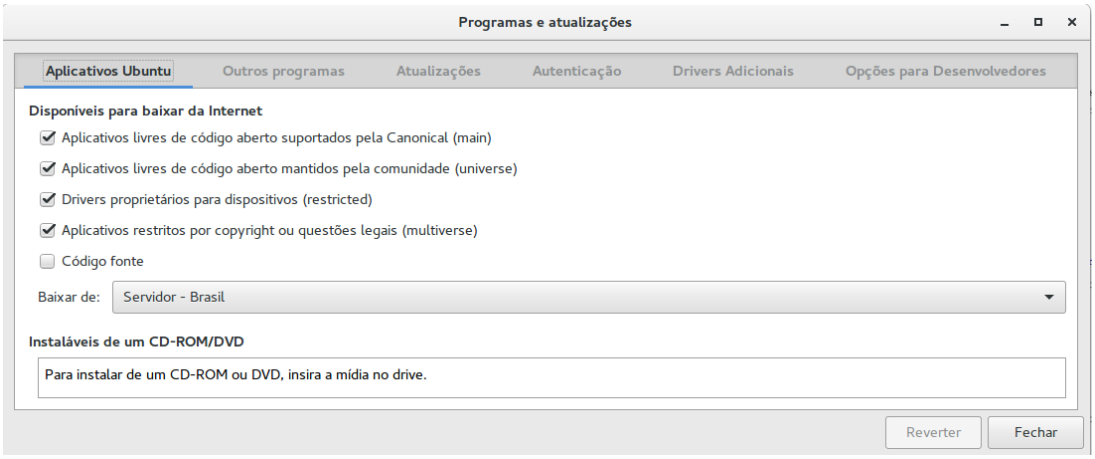
Para acessar todas as distribuições do ROS: http://wiki.ros.org/Distributions

Este guia segue o processo para instalação no OS Ubuntu 16.04, para a versão Kinetic Kame.

2.1.2 Preparação

Para preparar a instalação é necessário que os pacotes “restriced”, “universe” e “multiverse”, conforme a figura:

Figura 1: Configuração dos repositórios Ubuntu



Fonte: Autor.

Então, é necessário atualizar os dados de pacotes para o ROS, no terminal:

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(
    lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.
    list'
```

E as configurações de chave, no terminal:

```
sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80
    --recv-key 421C365BD9FF1F717815A3895523BAEEB01FA116
```

2.1.3 Instalação

Primeiro, checar se o índice de pacotes do sistema está atulizado, no terminal:

```
sudo apt-get update
```

A seguir, deve-se instalar a versão do ROS Kinetic Kame, recomenda-se a versão completa:

```
sudo apt-get install ros-kinetic-desktop-full
```

ou a versão normal:

```
sudo apt-get install ros-kinetic-desktop
```

Caso algum pacote específico seja necessário, no terminal, substituir "PACKAGE" pelo nome do pacote:

```
sudo apt-get install ros-kinetic-PACKAGE
```

2.1.4 Finalização

Antes de usar ROS, é preciso inicilizar o rosdep, que ajuda a instalar facilmente dependências necessárias para compilar e rodar os programas ROS, no terminal:

```
sudo rosdep init
```

```
rosdep update
```

Para que tudo fique configurado de forma correta sem ser necessário reconfigurar o terminal cada vez que um novo é aberto, no terminal:

```
source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

Por último, para instalar as ferramentas de gerenciamento dos workspaces ROS, no terminal:

```
sudo apt-get install python-rosinstall python-rosinstall-
    generator python-wstool build-essential
```

Este tutorial é baseado na wiki do ROS, qualquer dúvida, consultar: http://wiki.ros.org/kinetic/Installation/Ubuntu.

2.2 Configuração

2.2.1 Configurando o *Workspace*

Para utilizar o ROS com o robô, foi utilizado o ambiente de trabalho ***catkin***. O ***Catkin*** é uma ferramenta que porporciona um ambiente de trabalho para criação e execução de pacote ROS.

A configuração do ambiente de trabalho pode ser conferida no inicio da Introdução, no link correspondente.

O ambiente de trabalho contém 4 pastas, **build, devel, install e src**.

As pastas **build, devel e install** são pastas que o catkin coordena, então são pastas que normalmente não devem ser alteradas. A pasta **src** é a pasta que contém os pacotes feitos para ROS.

A pasta **src** o arquivo CMakeLists.txt, que é provido pelo ***catkin*** e também não deve ser alterado. As demais pastas representam os pacotes.

Para compilar os pacotes, basta digitar, na pasta `catkin_ws`:

```
catkin_make
```

2.2.2 Sistemas de Arquivos

O sistema de arquivos ROS é feito para facilitar a nevegação no **workspace**, existem algumas ferramentas que podem ser interessantes, como:

- ***rospack*:** É utilizado para mostrar informações sobre pacotes ROS, normalmente as opções utilizadas são “find” para localizar um pacote e “depends” para mostrar as dependências, “rospack + TAB” mostra as opções disponíveis para utilização do rospack. Rospack deve se utilizado conforme o exemplo:
  ```
  $ rospack find [package_name]
  $ rospack depends [package_name]
  ```

- ***roscd*:** É utilizado para navegação, pode navegar diretamente para um pacote, de qualquer diretório no terminal:
  ```
  $ roscd [package_name]
  ```

- ***roscd log*:** É utilizado para verificar logs de execuções recentes do núcle ROS, o comando é exatamente o descrito:
  `$ roscd log`

- ***rosls*:** Assim como o "ls", lista os arquivos e diretórios correspondentes, pondendo ser diretamente pelo pacote:
  `$ rosls [package_name]`

2.3 Funcionamento

O funcionamento do ROS é bastante simples. Consiste na execução de nodos que se comunicam entre si trocando comandos e informações, através de mensagens, tópicos e serviços, detalhados a seguir:

1. **Mensagens:** As mensagens são os tipos de dados trocados. Uma mensagem sempre tem um Header (Cabeçalho) e algum tipo de dados.

2. **Tópicos:** Os tópicos são as ferramentas que provêm a troca de informações, tópicos consistem em *Publishers* (aqueles que publicam informações) e *Subscribers* (aqueles que recebem informações). Os *subscribers* são orientados a eventos, quando uma mensagem é publicada, a função de *callback* é disparada.

3. **Serviços** Assim como os tópicos, serviços são utilizados para comunicação entre os nodos. Os tópicos são mais simples, e são mais usados.

Para o Pioneer 3DX, o ROS faz uma interface com a biblioteca padrão da *Adept Mobile Robots*, a biblioteca ARIA, que é uma biblioteca de controle para o robô em C++. A utilização do ROS propociona o controle para o robô de uma maneira diferente através de vários nodos, interfaceando os controles da biblioteca ARIA, para a estrutura de funcionamento do ROS, baseado em nodos e tópicos.

Existe um pacote padrão chamado ***RosAria***, este pacote fornece as funcionalidades básicas para o controle do robô utilizando ROS, são elas:

- Controle de velocidade (controle dos motores e direção).
- Aquisição dos valores de distância dos sensores sonar (ultrassônicos).
- Aquisição dos valores de distância do laser.
- Estado dos *bumpers* (parachoques) - Item não presente no Pioneer 3DX do LARM.
- Indicação do posicionamento (a partir da origem, quando o robô começa a se mover).

- Estado da tensão da bateria.

No pacote, existe um serviço de estado dos motores (para ligar e desligar os motores) e vários tópicos relacionados as informações trocadas.

2.4 Criação de Pacotes

Quando um pacote é criado, ele deve ser criado na pasta **src**, utilizando o comando ̈catkin_create_pkg NOMEPACOTE rospy roscpp:

Isto criará o pacote com os componentes e diretórios básicos. O arquivo "CMakeLists.txt" é utilizado pelo catkin para compilar o pacote, e o arquivo "package.xml" é utilizado para descrever as dependências do pacote, o arquivo segue um formato específico onde a ordem é importante. Os diretórios são utilizados para os arquivos fonte do projeto.

O arquivo **CMakeLists.txt** é de grande importância para o pacote, e deve seguir uma ordem específica:

- **find_packages:** Declara os pacote utilizados para execução do projeto, para utilização do ARIA, é recomendado sempre utilizar:

```
find_package(Boost REQUIRED COMPONENTS thread)
find_package(Aria QUIET)
if(Aria_FOUND)
    if(EXISTS "${Aria_INCLUDE_DIRS}/Aria.h")
        add_definitions(-DADEPT_PKG)
    endif()
  include_directories(${Aria_INCLUDE_DIRS})
  link_directories(${Aria_LIBRARY_DIRS})
else()
    # The installation package provided by Adept doesn't follow Debian policies
    if(EXISTS "/usr/local/Aria/include/Aria.h")
      add_definitions(-DADEPT_PKG)
      include_directories( /usr/local/Aria/include)
      link_directories(/usr/local/Aria/lib)
    endif()
endif()
```

  Dessa forma, os componentes ARIA (que são instalados separadamente, são incluídos de forma correta no projeto, demais pacotes pode ser adicionados de acordo que o que está no "CMakeLists.txt" do projeto p3dx_ufsc.

- **Declare ROS messages, services and actions:** Somente se forem criadas mensagens, serviços ou ações (normalmente não serão criados). Devem estar de acordo com o modelo do arquivo https://github.com/LARM-UFSC/ROS-Pioneer3DX/blob/master/Generic-CMakeLists.txt na seção "Declare ROS messages, services and actions". Podem ser excluídos caso não seja criado mensagens, serviços ou ações.

Declare ROS dynamic reconfigure parameters Não se mostra necessário para utilização do ROS com o Pioneer 3DX, pode ser removida. catkin specific configuration Cria as configurações para o pacote catkin, deverá ser retirado o comentário das linhas correspondente ao que foi usado, isso servirá caso o pacote seja usado como dependência de outro pacote:

Neste caso, apenas as dependências de pacotes catkin foram utilizadas, o diretório INCLUDE não foi utilizado, libararies não foram criadas e nenhum biblioteca do sistema foi utilizada.

2.5 Criação de Mensagens

Mensagens podem ser úteis quando deseja-se criar um tópico com um tipo específico, no caso do pacote criado, foi desenvolvida uma mensagem do tipo "Velocidade". Essa mensagem contém dois números de ponto flutuante que indicam as velocidades linear e angular.

Para criar uma mensagem o procedimento é simples:

1. No diretório do pacote, criar a pasta "msg".

2. Na pasta "msg", criar um arquivo com o nome da mensagem, neste exemplo: "Velocidade.msg".

3. O arquivo de mensagens sempre precisará de um *Header*, seguido pelos tipos de variáveis que se deseja criar, por exemplo:
   ```
   Header header
   float32 linearvel
   float32 angularvel
   ```

4. No arquivo CMakeLists.txt adicionar de acordo com o arquivo genérico disponível na página do GitHub:

   ```
   find_package(catkin REQUIRED COMPONENTS message_generation )

   add_message_files( FILES Velocidade.msg )

   generate_messages( DEPENDENCIES std_msgs )

   catkin_package( CATKIN_DEPENDS message_runtime )
   ```

5. No arquivo "Package.xml", adicionar:

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

Qualquer dúvida na criação de mensagens para o ROS, o tutorial (em inglês): http://wiki.ros.org/ROS/Tutorials/CreatingMsgAndSrv, pode ser consultado.

2.6 Criação de Tópicos

A abordagem de tópicos é bem simples. Todo tópico tem *publishers* (quem envia informações para o tópicos), e *subscribers* (quem lê informações do tópico).

A partir do momento que algum nodo publica um dado em algum tópico, todos os *subscribers* recebem disparam uma função de leitura da informação.

Tópicos não precisam ser criados, basta que um nodo publique para um tópico ser criado.

Para criar um Publisher, em C++:

```
//Declaração do publisher
ros::Publisher publish;
//Criação do Tópico "socketCom", "Node" é o nodo de execução do programa
publish = Node.advertise<std_msgs::Char>("socketCom", 1);
//Cria mensagem a ser enviada
std_msgs::Char msg;
msg.data = 'A';
//Publica a mensagem no tópico
publish.publish(msg);
```

Para criar um Subscriber, em C++:

Os *subscribers* funcionam por chamada de eventos, se a função fazer parte de uma classe (como no exemplo, deve se usar o último parâmetro da função como ***this***). A seguir o exemplo:

```
//Declaração do subscriber
ros::Subscriber mobileApp_sub;
//Criação da inscrição no tópico
//Quando algo é publicado no tópico, a função VelCallback é disparada
mobileApp_sub = Node.subscribe("VelTopic", 1, &Monitor::VelCallback, this);
//Função de callback para o tópico
void Monitor::VelCallback(const p3dx_ufsc::Velocidade msg){
```

```
 linearvel = msg.linearvel;
 angularvel = msg.angularvel;
}
```

Quando usar um tópico é necessário incluir a mensagem no código, para isso utilizar o padrão "escopo/Mensagem.h", como mostra o exemplo:

```
#include "p3dx_ufsc/Velocidade.h"
```

2.7 Aria

Aria é o pacote disponibilizado pela *MobileRobots*, também está disponível para download no GitHub do projeto: https://github.com/LARM-UFSC/ROS-Pioneer3DX/tree/master/libaria.

Essa biblioteca deve ser instalada porque o ROS faz o interfaceamento dela com o sistema desenvolvido.

2.8 Utilização

Para facilitar a utilização deste pacote, foi criado um novo pacote, ***p3dx_ufsc***, disponível no GitHub do projeto, cujo link está disponível na seção de links. Neste pacote, foram criadas classes para cada um dos itens acima, deixando-o melhor organizado para futuras modificações. Cada classe, possui um *subscriber* para os tópicos criados no pacote ***RosAria*** dessa forma, os objetos de cada classe sempre terão seus valores atualizados, assim que algo for publicado no tópico correspondente.

Além da abstração do pacote ***RosAria***, foram criados novos mecanismos para controle do robô, numa avaliação da biblioteca ARIA, foi constatada a implementação de Sockets, dessa forma, foi criado um aplicativo para controle do robô, que se comunica com o mesmo através de Sockets, esse controle oferece transição rápida e simples entre os modos de operação: Autônomo e Teleoperado (controlado pelo celular).

Em comparação com o controle tradiconal (JoyStick), o controle pelo aplicativo fornece melhor feedback para o usuário, já que é possível receber informações sobre o status do robô.

Atualmente o robô funciona com 4 nodos de execução:

- **RosAria:** Nodo que possui os *publishers* dos tópicos padrões.
- **p3dx_ufsc_main:** Nodo responsável pelo controle do robô, ele que controla a velocidade e direção do robô, e as leituras dos sensores quando no modo autônomo.

- **p3dx_ufsc_socket:** Nodo responsável pela comunicação de controle do robô, esse nodo é responsável por receber informações do controle através do aplicativo, e publicar no tópico de controle, onde o *subscriber* do tópico é o nodo **p3dx_ufsc_main**.
- **p3dx_ufsc_socket_info:** Nodo responsável por adiquirir o valor dos sensores publicados pelo nodo **RosAria**, essas informações são também publicadas em um tópico de informações (sem *subscribers* definidos, além de serem enviadas para a aplicação via Socket de informações, dessa forma, as informações relacionadas a velocidade, posição, lasers, sonares (não funcionam no robô, apenas na simulação) e bumpers (não presente no Pioneer 3DX do LARM), ficam disponíveis para acesso no celular.

A criação de novos tópicos para controle do robô se baseia na abstração das bibliotecas padrão ARIA, para o sistema do ROS. Já a criação de novas métodologias de controle (autônomo com alguma inteligência artificial) pode ser feita no pacote ***p3dx_ufsc***.

3 Software Desenvolvido

3.1 Biblioteca p3dx_ufsc

O pacote conta com 6 bibliotecas que abstraem o uso do pacote ***rosaria***. A seguir serão abordadas as funções de cada classes descrevendo-as:

3.1.1 Classe Battery

Arquivo: "src/libs/battery.hpp"

Funções:

- **Public float getBattery()**:

  Parâmetros: Nenhum.

  Retorno: Retorna a tensão das baterias

  Descrição: Função retorna o valor de tensão das baterias.

- **Private void callbackVoltage()**:

  Parâmetros: const std_msgs::Float64 msg.

  Retorno: Nenhum.

  Descrição: Função de callback disparada quando alguma informação é publicada no Tópico "battery_voltage".

- **Private void callbackState()**:

  Parâmetros: const std_msgs::Float32 msg.

  Retorno: Nenhum.

  Descrição: Função de callback disparada quando alguma informação é publicada no Tópico "battery_state_of_charge".

- **Private void callbackRecharge()**:

  Parâmetros: const std_msgs::Int8 msg.

  Retorno: Nenhum.

  Descrição: Função de callback disparada quando alguma informação é publicada no Tópico "battery_recharge_state".

3.1.2 Classe Bumpers

Arquivo: "src/libs/bumpers.hpp"

Funções:

- **Public std::vector <bool> getFrontBumpers()**:

  Parâmetros: Nenhum.

  Retorno: Retorna valor do estado dos para-choques frontais.

  Descrição: Função que retorna valor do estado dos para-choques frontais.

- **Public std::vector <bool> getRearBumpers()**:

  Parâmetros: Nenhum.

  Retorno: Retorna valor do estado dos para-choques traseiro.

  Descrição: Função que retorna valor do estado dos para-choques traseiros.

- **Private void callbackBumpers()**:

  Parâmetros: const rosaria::BumperState &msg.

  Retorno: Nenhum.

  Descrição: Função de callback disparada quando alguma informação é publicada no Tópico "bumper_state".

3.1.3 Classe Lasers

Arquivo: "src/libs/lasers.hpp"

Funções:

- **Public std::vector <float> getLasers()**:

  Parâmetros: Nenhum.

  Retorno: Retorna vetor de distâncias da laser.

  Descrição: Função que retorna os valores de distância dos lasers do robô.

- **Private void callbackLasers()**:

  Parâmetros: const sensor_msgs::LaserScan msg.

  Retorno: Nenhum.

  Descrição: Função de callback disparada quando alguma informação é publicada no Tópico "sim_lms1xx_1_laserscan" se estiver usando o simulador MobileSim, ou no Tópico "lms2xx_1_laserscan" se estiver utilizando o robô Pioneer 3DX.

3.1.4 Classe Motors

Arquivo: "src/libs/motors.hpp"

Funções:

- **Public void setVelocity()**:

  Parâmetros: *float* ang, *float* lin.

  Retorno: Nenhum.

  Descrição: Função define as velocidades linear e angular do motor.

- **Public void setSclAng()**:

  Parâmetros: *float* scl_a.

  Retorno: Nenhum.

  Descrição: Função define o escalar que multiplica a velocidade angular dos motores.

- **Public void setSclLin()**:

  Parâmetros: *float* scl_l.

  Retorno: Nenhum.

  Descrição: Função define o escalar que multiplica a velocidade linear dos motores.

- **Public void enableMotors()**:

  Parâmetros: Nenhum

  Retorno: Nenhum.

  Descrição: Função define habilita os motores.

  Descrição: Função define o escalar que multiplica a velocidade linear dos motores.

- **Public void disableMotors()**:

  Parâmetros: Nenhum

  Retorno: Nenhum.

  Descrição: Função define desabilita os motores.

- **Public bool getMotorsState()**:

  Parâmetros: Nenhum

  Retorno: Valor do estado dos motores.

  Descrição: Função que devolve o valor do estado dos motores.

- **Private void callbackMotorsState()**:

  Parâmetros: const std_msgs::Bool msg.

  Retorno: Nenhum.

  Descrição: Função de callback disparada quando alguma informação é publicada no Tópico "motors_state".

3.1.5 Classe Pose

Arquivo: "src/libs/pose.hpp"

Funções:

- **Public RobotPosition getPose()**:

  Parâmetros: Nenhum.

  Retorno: Retorna vetor da posição do robô.

  Descrição: Função que retornas os valores de posição para o robô, em x, y, z e *orientação*, em relação a posição original.

- **Private void callbackPose()**:

  Parâmetros: const nav_msgs::Odometry msg.

  Retorno: Nenhum.

  Descrição: Função de callback disparada quando alguma informação é publicada no Tópico "pose".

3.1.6 Classe Sonar

Arquivo: "src/libs/sonar.hpp"

Funções:

- **Public std::vector < Point32 > getSonar()**:

  Parâmetros: Nenhum.

  Retorno: Retorna com os valores de medição do sensores sonar.

  Descrição: Função que retorna a distância medida pelos sensores sonar.

- **Private void callbackSonar()**:

  Parâmetros: const sensor_msgs::PointCloud msg.

  Retorno: Nenhum.

  Descrição: Função de callback disparada quando alguma informação é publicada no Tópico "sonar".

Além das classes criadas para comunicação com o nodo ***rosaria***, foi criada uma classe que é responsável pelo controle do robô, que instância objetos das classes acima citadas:

3.1.7 Classe AutoRun

Classe responsável pelo controle do Robô.

Arquivo: "AutoRun.hpp"

Funções:

- **Public void mainLoop()**:

  Parâmetros: Nenhum.

  Retorno: Nenhum.

  Descrição: Função responsável pelo controle do robô, é nesta função que deve ser implementadas as rotinas de controle do robô, os controles são definidos através dos objetos instanciados pelas outras classes.

- **Private void callbackMobile()**:

  Parâmetros: const std_msgs::Char msg.

  Retorno: Nenhum.

  Descrição: Função de callback disparada quando alguma informação é publicada no Tópico "socketCom", esse função retorna o comando proveniente do aplicativo desenvolvido.

Para os pacotes relacionados ao controle pelo celular, foram feitas três classes, a classe **monitor**, utilizada para monitorar os dados do robô, classe **SocketServer**, classe utilizada para receber comandos do aplicativo mobile desenvolvido e a classe **SocketServerInfo**, responsável pelo envio das informações provenientes da classe **monitor** para o aplicativo mobile:

3.1.8 Classe Monitor

Arquivo: "monitor.hpp"

Funções:

- **Public std::string getInfo()**:

  Parâmetros: Nenhum.

  Retorno: *String JSON* com os dados medidos do robô.

Descrição: Função que retorna os valores do estado do robô: posição, sonar, laser, velocidade e bateria.

- **Private void VelCallback()**:

  Parâmetros: const p3dx_ufsc::Velocidade msg.

  Retorno: Nenhum.

  Descrição: Função de callback disparada quando alguma informação é publicada no Tópico "VelTopic".

3.1.9 Classe SocketServer

Classe responsável pelo controle do Robô.

Arquivo: "socket.cpp"

Funções:

- **Public bool WaitConnection()**:

  Parâmetros: Nenhum.

  Retorno: Retorna o estado da tentativa de conexão.

  Descrição: Função responsável por retornar o valor do estado da tentativa de conexão, se foi bem sucedida ou não.

- **Public void mainLoop()**:

  Parâmetros: Nenhum.

  Retorno: Nenhum.

  Descrição: Função responsável por verificar se algum comando foi recebido do aplicativo Mobile, se sim, publica no tópico "socketCom".

3.1.10 Classe SocketServerInfo

Classe responsável pelo controle do Robô.

Arquivo: "socket_info.cpp"

Funções:

- **Public bool WaitConnection()**:

  Parâmetros: Nenhum.

  Retorno: Retorna o estado da tentativa de conexão.

  Descrição: Função responsável por retornar o valor do estado da tentativa de conexão, se foi bem sucedida ou não.

- **Public void mainLoop()**:

  Parâmetros: Nenhum.

  Retorno: Nenhum.

  Descrição: Função responsável apenas por fazer as iterações do ROS e manter o nodo funcionando.

- **Private void timerCallback()**:

  Parâmetros: const ros::TimerEvent & event.

  Retorno: Nenhum.

  Descrição: Função que é disparada a cada intervalo de tempo, nesta função a *string JSON* da classe *monitor* é adquirida, a mensagem é publicada no tópico "robotInfo", e também envia uma mensagem com as informações via *socket* para o aplicativo *mobile*.

3.2 Aplicativo Mobifeteira

4 Considerações Finais

Esta documentação mostra todos os passos para utilizar o ROS junto com o robô Pioneer 3DX. São abordados os conceitos desde a instalação do ROS e dos componentes do sistema.

Também foram documentadas as bibliotecas desenvolvidas, a biblioteca de controle do Pioneer Utilizando ROS, e o aplicativo desenvolvido para celular.

Os arquivos utilizados podem ser encontrados no GitHub do Laboratório de Automação e Robótica Móvel.

4.1 Links úteis

- **GitHub do Projeto**

  https://github.com/LARM-UFSC/ROS-Pioneer3DX

- **Instalação ROS**

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

- **Configuração ROS (Usar Catkin)**

  http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment

- **Criação de Pacotes**

  http://wiki.ros.org/ROS/Tutorials/CreatingPackage

- **Criação de Mensagens**

  http://wiki.ros.org/ROS/Tutorials/CreatingMsgAndSrv

- **Entendo Tópicos**

  http://wiki.ros.org/ROS/Tutorials/UnderstandingTopics

- **Manual Pioneer**

  https://www.inf.ufrgs.br/~prestes/Courses/Robotics/manual_pioneer.pdf

- **MobileSim - *Fora do Ar***

  http://robots.mobilerobots.com/wiki/MobileSim

- **Libaria Package**

  https://launchpad.net/ubuntu/+source/libaria