



Orocos walk-through

Best Practices in integrating complex robotic systems

Gianni Borghesan

KU Leuven

Febraury 2020

Outline

- 1 Preparation steps
- 2 Exercise 1 – make a source and a sink
- 3 Exercise 2 - Add operations to set value, Marshalling
- 4 Exercise 3 - Basic Ros integration, streaming on topics

Install the software I

Install Ubuntu 16.04

- ▶ Download the ISO
- ▶ Create a Bootable USB
- ▶ Launch the installer and follow the instructions

Install extra packages

- ▶ e.g. `apt-get install git ssh kst`
- ▶ install some editor of your liking, `qtcreator`, `kate`, `kdevelop..`

Install the software II

Install ROS Kinetic

- ▶ <http://wiki.ros.org/kinetic/Installation/Ubuntu>
- ▶ choose desktop full meta-package
`apt-get install ros-kinetic-desktop-full`
- ▶ follows all the instructions of the wiki page up to point 1.7.

Install the software III

ROSCon 2019 is happening in Macau Oct 3rd and Nov 1st, Register Here!

ROS.org [About](#) | [Support](#) | [Discussion Forum](#) | [Service Status](#) | [Q&A answers.ros.org](#)

Documentation **Browse Software** **News** **Download**

kinetic/ Installation/ Ubuntu

Ubuntu install of ROS Kinetic

We are building Debian packages for several Ubuntu platforms, listed below. These packages are more efficient than source-based builds and are our preferred installation method for Ubuntu. Note that there are also packages available from Ubuntu upstream. Please see [Upstream Packages](#) to understand the difference.

Ubuntu packages are built for the following distros and architectures.

Distro amd64 i386 armhf
Wily X X
Xenial X X X
Jessie X X X

If you need to install from source (not recommended), please see [source \(download-and-compile\) installation instructions](#).



If you rely on these packages, please support OSRF.

These packages are built and hosted on infrastructure maintained and paid for by the [Open Source Robotics Foundation](#), a 501(c)(3) non-profit organization. If OSRF were to receive one penny for each downloaded package for just two months, we could cover our annual costs to manage, update, and host all of our online services. Please consider [donating to OSRF](#) today.

Contents

1. Ubuntu install of ROS Kinetic
- 1.1. Installation
- 1.1.1. Configure your Ubuntu repositories
- 1.2. Setup your sources.list
- 1.3. Set up your keys
- 1.4. Environment setup
- 1.5. Initialize rosdep
- 1.6. Build farm status
- 1.7. Dependencies for building packages
2. Tutorials

1. Installation

ROS Kinetic **ONLY** supports Wily (Ubuntu 15.10), Xenial (Ubuntu 16.04) and Jessie (Debian 8) for debian packages.

1.1 Configure your Ubuntu repositories

Configure your Ubuntu repositories to allow "restricted," "universe," and "multiverse." You can [follow the Ubuntu guide](#) for instructions on doing this.

1.2 Setup your sources.list

Setup your computer to accept software from packages.ros.org

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

Warning: Source Debs are also available

1.3 Set up your keys

```
sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key C1CF621646ADE686 8B7867C42E95462C0E84
```

Wily

[Distributions](#)
[ROS Installation](#)
[ROS Subsets](#)
[Recent Changes](#)
[Upstream Packages](#)

Ubuntu

Page

[Introduction Page](#)
[FAQ](#)
[About ROS](#)
[More Actions](#)

User

[Login](#)

or [click here](#)

Individual Package: You can also install a specific ROS package (replace underscores with dashes of the package name):

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

e.g.

```
sudo apt-get install roa-kinetic-etam-g-mapping
```

To find available packages, use:

```
apt-cache search roa-kinetic
```

1.5 Initialize rosdep

Before you can use ROS, you will need to initialize `rosdep`. `rosdep` enables you to easily install system dependencies for source you want to compile and is required to run some core components in ROS.

```
sudo rosdep init  
rosdep update
```

1.6 Environment setup

It's convenient if the ROS environment variables are automatically added to your bash session every time a new shell is launched:

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

If you have more than one ROS distribution installed, `~/.bashrc` must only source the `setup.bash` for the version you are currently using.

If you just want to change the environment of your current shell, instead of the above you can type:

```
source /opt/ros/kinetic/setup.bash
```

If you use `zsh` instead of `bash` you need to run the following commands to set up your shell:

```
echo "source /opt/ros/kinetic/setup.zsh" >> ~/.zshrc  
source ~/.zshrc
```

1.7 Dependencies for building packages

Up to now you have installed what you need to run the core ROS packages. To create and manage your own ROS workspaces, there are various tools and requirements that are distributed separately. For example, `catkin` is a frequently used command-line tool that enables you to easily download many source trees for ROS packages with one command.

To install the tool and other dependencies for building ROS packages, run:

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

1.8 Build farm status

The packages that you installed were built by the [ROS build farm](#). You can check the status of individual packages [here](#).

2. Tutorials

Now, to test your installation, please proceed to the [ROS Tutorials](#).

Except where otherwise noted, the ROS wiki is licensed under the [Creative Commons Attribution 3.0](#)

Wiki: kinetic/installation/Ubuntu (last edited 2014-07-25 00:49:29 by TullyPaine)

Support us by Open Source Robotics Foundation

Install the software IV

Install Orocos

- ▶ `apt-get install ros-kinetic-rtt-ros-integration`

Test Orocos

- ▶ run `deployer` in a shell, and see if it works.

Setup a Catkin Workspace

We will use the ros build system, so you will need a catkin workspace to build the exercise:

- ▶ `mkdir -p ~/ws/src`
- ▶ `cd ~/ws`
- ▶ `catkin_make`
- ▶ `source devel/setup.bash`

see http://wiki.ros.org/catkin/Tutorials/create_a_workspace.

Automatically generate an orocos package skeleton I

Luckily, there is a command for that:`orocreate-catkin-pkg`

```
Usage: orocreate-catkin-pkg [-f] [--catkin] package_name  
[component] [plugin] [service] [typekit] [support] [empty]  
[cmake]
```

We consider only the **component** option, e.g.:

- ▶ `cd ~/ws/src`
- ▶ `orocreate-catkin-pkg hello1 component`

Automatically generate an orocos package skeleton II

Generated files

```
- hello1/  
  - CMakeLists.txt  
  - package.xml  
  - src  
    - CMakeLists.txt  
    - hello1-component.cpp  
    - hello1-component.hpp
```

Automatically generate an orocos package skeleton III

Root CMake

```
...  
project(hello1)  
...  
# Rest of the configuration is in src/  
add_subdirectory( src )  
  
orocos_generate_package()
```

src CMake

```
...  
orocos_component(hello1 hello1-component.hpp hello1-component.cpp)  
    # ...you may add multiple source files  
...
```

Automatically generate an orocos package skeleton IV

Header

```
#ifndef OROCOS_HELLO1_COMPONENT_HPP
#define OROCOS_HELLO1_COMPONENT_HPP

#include <rtt/RTT.hpp>

class Hello1 : public RTT::TaskContext{
public:
    Hello1(std::string const& name);
    bool configureHook();
    bool startHook();
    void updateHook();
    void stopHook();
    void cleanupHook();
};
#endif
```

Automatically generate an orocos package skeleton V

CPP file

```
#include "hello1-component.hpp"
#include <rtt/Component.hpp>
#include <iostream>

Hello1::Hello1(std::string const& name) : TaskContext(name){
    std::cout << "Hello1 constructed !" <<std::endl;
}
bool Hello1::configureHook(){
    std::cout << "Hello1 configured !" <<std::endl;
return true;
}
bool Hello1::startHook(){
    std::cout << "Hello1 started !" <<std::endl;
return true;
}
void Hello1::updateHook(){
    std::cout << "Hello1 executes updateHook !" <<std::endl;
}
...

ORO_CREATE_COMPONENT( Hello1)
```

Automatically generate an orocos package skeleton VI

Now compile...

Just `cd` to the root directory, and run `cakin_make` !

...and play with the Deployer

- ▶ `import ("hello1")`
- ▶ `displayComponentTypes`
- ▶ `loadComponent("hello", "Hello1")`
- ▶ `cd hello`
- ▶ `ls`
- ▶ `start`
- ▶ `trigger`
- ▶ `stop`
- ▶ `setPeriod (1)`
- ▶ `start`

Inspecting components and commands

All the properties, ports, and operations (included inherited ones) that are documented can be inspected. Try:

- ▶ `ls`
- ▶ `help name-of-component` *or* `help this`
- ▶ `help name-of-component`
- ▶ `help name-of-operation/service`

Outline

- 1 Preparation steps
- 2 Exercise 1 – make a source and a sink
- 3 Exercise 2 - Add operations to set value, Marshalling
- 4 Exercise 3 - Basic Ros integration, streaming on topics

Objectives

- ▶ Make a component source that outputs a number
 - Value configurable from property
 - Then a time-dependent function
- ▶ Make a component sink that reads the number
 - print out if a new value arrives if periodic and async
- ▶ deploy and connect ports

Preparation

- ▶ Make a new package `exe1`.
- ▶ Prepare the files for two components, `sink` and `source`, and modify `CMakeLists.txt` accordingly.
- ▶ Prepare a file called `start.ops`.

Source I

To add Property to the interface

to add a port and write something on it:

```
//hpp; I normally put in private
int output_value;
//cpp, constructor
source::source(std::string const& name) : TaskContext(name, PreOperational)
, output_value(0){
addProperty("output_value", output_value).doc("Value to write out");
}
```

Source II

To add Output Port to the interface

to add a port and write something on it:

```
//hpp
RTT::OutputPort<int> output_value;

//cpp, constructor
addPort("output_value", output_value).doc("My output value");

//cpp, updatehook
output_value.write(output_value);
```

Source III

Basic start.ops

To import the package and load the two components

```
import ("exel")
loadComponent("source", "source")
loadComponent("sink", "sink")
```

Now compile, and check if data is written with “leave” (in Deployer)

```
cd source
configure
start
leave
var int i
outport_value.read (i)
i
enter
```

Sink I

To add Ports to the interface
to add input ports and get data

```
//hpp
RTT::InputPort<int> inport_value;

//cpp, constructor
addPort("inport_value", inport_value)
    .doc("My input port");

//cpp, updatehook
int data_in;
RTT::FlowStatus fs = inport_value.read(data_in);
if (fs==RTT::NoData){ std::cout<<"no data"<<std::endl;}
else if (fs==RTT::OldData){ std::cout<<"old data"<<std::endl;}
else {std::cout<<"data: " << data_in <<std::endl;} //fs=RTT::NewData
```

Sink II

Now compile, and check if data can be written with “leave” (in Deployer)

```
cd sink
start
leave
inport_value.write(1)
sink.trigger
```

Sink III

Event Port

they works similarly to normal input ports, but a function will be called every time they receive data. (**default: updateHook**)

```
//hpp
RTT::InputPort<int> event_inport_value;
//cpp, constructor
addEventPort("event_inport_value", event_inport_value)
    .doc("My input port - event");
```

Sink IV

Event Port make it complex

Use a **callback** in place of the default updateHook

```
//cpp, constructor
addEventPort("event_inport_value", event_inport_value , boost::bind(&sink::my_callback , this , _1))
    .doc("My input port - event");
//cpp
void sink::my_callback(RTT::base::PortInterface*p){
    this->trigger();
    std::cout<<"callback"<<std::endl;
}
```

- ▶ to call the update from the callback, use `this->trigger()`
- ▶ to put a generic (e.g. member class) function, use `boost::bind`

Sink V

Question

why callback is written before the text in the updateHook?

Deployment

```
import ("exel")
loadComponent("source", "source")
loadComponent("sink", "sink")
source.setPeriod(1)
source.configure
source.output_value=3
connect("sink.event_inport_value", "source.output_value", ConnPolicy())
sink.start
source.start
```

Make a the source a sine generator

write a signal (e.g. sine with period and amplitude configurable from properties) to a port. Hint: a periodic component can retrieve his period with `get period`: otherwise, you can get time with the following:

```
//hpp
#include <rtt/os/TimeService.hpp>
...
private:
...
RTT::os::TimeService::ticks time_begin;
//cpp, starthook
time_begin=RTT::os::TimeService::Instance()->getTicks();
//cpp, updateHook
double time_passed=RTT::os::TimeService::Instance()->secondsSince(time_begin);
```

Outline

- 1 Preparation steps
- 2 Exercise 1 – make a source and a sink
- 3 Exercise 2 - Add operations to set value, Marshalling
- 4 Exercise 3 - Basic Ros integration, streaming on topics

Objectives

- ▶ Add an operation `set_amplitude` and `set_frequency` to the source component
- ▶ protect the amplitude and frequency so that they can be loaded only at configure time.
- ▶ Make a configuration file, and load before configure.
- ▶ protect the change via operation so that is possible to do it only when the source component is stopped.

Preparation

- ▶ make another component `source2`, copying `source`.
- ▶ make another deployment script, `start2.ops`.

Properties I

- ▶ Properties are variables that you can read and write
 - from the component and
 - from the component interface.
- ▶ The main use of properties is to configure a component, via the **Marshaller** service, or scripts.

Properties II

```
//hpp
int my_property;
//cpp, constructor
addProperty("my_property", my_property).doc("A property");
//cpp, updateHook
my_property=34;
int var=my_property;
```

Property changes dynamically!

If you do not want to allow to change the variable at anytime, cache it, or consider a setter/getter operation!

Protect Properties

- ▶ create other two variables for caching (hint, rename properties as `_prop`)
- ▶ copy values in configure hook
- ▶ Maybe is a good idea to document in the `.doc` the behaviour!

Make a file for properties

- ▶ load the marshalling `source.loadService ("marshalling")`
- ▶ play around a little with it...
- ▶ generate a file
- ▶ modify the file and read it again

```
Deployer [S]> source.loadService ("marshalling")  
= true  
Deployer [S]> source.marshalling.writeProperties ("prop.cpf")  
= true  
//modify file  
Deployer [S]> source.marshalling.readProperties ("prop.cpf")  
= true
```

A configuration file example

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE properties SYSTEM "cpf.dtd">
<properties>
  <simple name="amplitude" type="double"><description>Value amplitude</description>
    <value>2</value></simple>
  <simple name="frequency" type="double"><description>Value frequency</description>
    <value>0.1</value></simple>
</properties>
```

Exercise: Now make read persistent, before the configure hook (write command in deployment script)

Add Operations I

Add an operation to the interface

```
//hpp - declare class function
bool set_amplitude(double);
//cpp constructor
addOperation( "set_amplitude", &source2::set_amplitude , this , RTT::OwnThread)
    .doc("Set The amplitude");

//cpp, function body
bool source2::set_amplitude(double a){
    ...
    return true;
}
```

Add Operations II

Now make sure that the operation cannot modify variables during running

Hint

the task context offers introspection on his on life cycle state machine:

- ▶ `this->isRunning()`
- ▶ `this->isConfigured()`
- ▶ `this->inException()`
- ▶ ...

Add Operations

Now make sure that the operation cannot modify variables during running

```
bool source2::set_amplitude(double a){
    if (this->isRunning()){
        RTT::Logger::In in(this->getName());
        RTT::log(RTT::Error)<<" set amplitude failed, component is running"<<RTT::endlog();
        return false;
    }
    amplitude_prop=amplitude=a;
    return true;
}
```

Outline

- 1 Preparation steps
- 2 Exercise 1 – make a source and a sink
- 3 Exercise 2 - Add operations to set value, Marshalling
- 4 Exercise 3 - Basic Ros integration, streaming on topics

Steps for streaming on a topic

- ▶ using for ports the typekit generated from ros messages.
- ▶ loading the Ros services.
- ▶ use the stream command

see https://github.com/jhu-lcsr/rtt_ros_examples

Integrate a ros messages-generated typekits in a component

Using a Ros message is the same of using any other type.

```
// hpp
#include <std_msgs/Float64.h>
...
RTT::OutputPort<std_msgs::Float64> output_value_ros;
//cpp, constructor
addPort("output_value_ros", output_value_ros).doc("My output value, ros");
//cpp, update hook
std_msgs::Float64 f;
f.data=val;
output_value_ros.write(f);
```

In deployment...

Using a Ros message is the same of using any other type.

```
import("rtt_ros")//optional, allows to import with the dependencies of package.xml
import("rtt_std_msgs")
import("rtt_rosnode")//makes orocos a ros node. will complain if roscore is not running
import("rtt_roscomm")//contains the topic services
... load, configure, ...
stream("source_output_value_ros", ros.topic("float_out"))
```

Build a new typekit from a ROS message I

create a Ros package with only msgs

```
catkin_create_pkg my_custom_msgs std_msgs rospy roscpp
```

write a msg in the msg folder

```
Float64 x  
Float64 y
```

Modify, following the instructions in `CMakeLists.txt`, the files

- ▶ `CMakeLists.txt`
- ▶ `Package.xml`

...and compile with `catkin_make`.

More info: <http://wiki.ros.org/ROS/Tutorials/CreatingMsgAndSrv>

Build a new typekit from a ROS message II

Generate the typekit

We will use a facility of the `rtt_roscomm` package
(github.com/orocos/rtt_ros_integration/tree/indigo-devel/rtt_roscomm)

```
roslaunch rtt_roscomm create_rtt_msgs my_custom_msgs
```

then compile and try to load the typekit, as for the `rtt_std_msgs`