Software Architectures for Robotics

Lab Session 2

Building your first application

UNIVERSITÀ DEGLI STUDI DI GENOVA | Dibris

EMARO LAB

Table of contents

Definitions

- **Packages:** Packages are the software organization unit of ROS code.

  Each package can contain libraries, executables, scripts, or other artifacts.

- **Manifests (`package.xml`):** A manifest is a description of a *package*.

  It serves to define dependencies between *packages* and meta information about the *package* like version, maintainer, license, etc...

Creating a new package

A package has the following requirements:

- The package must contain a catkin compliant `package.xml` file
- The package must contain a catkin compliant `CMakeLists.txt`
- Each package must have its own folder

Tip: you can group multiple packages in a single folder as long as every package has **its own folder!**

Structure Example

```
workspace_folder/      -- WORKSPACE
  src/                 -- SOURCE FOLDER
    CMakeLists.txt     -- 'toplevel' file (ignore it)
    package_1/
      CMakeLists.txt   -- package_1 CMakeLists.txt
      package.xml      -- package_1 package manifest
    ...
    package_n/
    CMakeLists.txt     -- package_n CMakeLists.txt
    package.xml        -- package_n package manifest
```

Creating a new package

Navigate to your workspace then create a package

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg beginner_tutorials
std_msgs rospy roscpp
```

This creates a package `beginner_tutorials` with the related `CMakeLists.txt` and *`package.xml`*

`std_msgs rospy roscpp` will automatically be added to both files

Creating a new package

The general command is:

```
$ catkin_create_pkg <package_name> [depend1] ...
```

This command must be executed in the workspace source directory; e.g. `~/catkin_ws/src`

Package decription | manifest

Toggle line numbers

```
 1 <?xml version="1.0"?>
 2 <package>
 3   <name>beginner_tutorials</name>
 4   <version>0.1.0</version>
 5   <description>The beginner_tutorials package</description>
 6
 7   <maintainer email="you@yourdomain.tld">Your Name</maintainer>
 8   <license>BSD</license>
 9   <url type="website">http://wiki.ros.org/beginner_tutorials</url>
10   <author email="you@yourdomain.tld">Jane Doe</author>
11
12   <buildtool_depend>catkin</buildtool_depend>
13
14   <build_depend>roscpp</build_depend>
15   <build_depend>rospy</build_depend>
16   <build_depend>std_msgs</build_depend>
17
18   <run_depend>roscpp</run_depend>
19   <run_depend>rospy</run_depend>
20   <run_depend>std_msgs</run_depend>
21
22 </package>
```

Package decription | `cmake`

`CMakeLists.txt` is used by the compiler to build the package.

Catkin relies on `cmake` as build tool.

You need to include all external *dependencies*, *libraries* and *folders* inside your `CMakeLists.txt`,

just like a standard C++ project!

Package decription | `CMakeLists`

Open `CMakeLists` located in:

`~/catkin_ws/src/[your_package]/`

ROS creates a templated `CMakeLists` for you to use.

Uncomment and fill the desired fields in order to compile your project.

(don't worry now, we will do that later on)

Build your empty package

Building is very simple at this stage:

```
$ cd ~/catkin_ws
$ catkin_make
```

This command must be invoked on the **top-level** `CMakeLists` in your workspace directory.

Every package `CMakeLists` will be processed and every package will be compiled independently, taking into account cross-dependencies!

Behind the scenes, every `CMakeLists` is converted into a `Makefile` with the necessary *build instructions* for the compiler.

Creating a publisher

A publisher is a ROS node that publishes messages on desired topics.

As an example, we will try to publish a `String` on a predefined topic

```
$ cd ~/catkin_ws/[your_package]     //navigate to package
$ cd src
$ touch talker.cpp                  //create a file
```

 `src/talker.cpp` is our node source code!

At the moment just copy/paste the code here:
http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber(c++)

Publisher breakdown

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>

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

  ros::init(argc, argv, "talker");
  ros::NodeHandle n;
  ros::Publisher chatter_pub =
                    n.advertise<std_msgs::String>("chatter", 1000);

  ros::Rate loop_rate(10);
  int count = 0;

  while (ros::ok()){
    std_msgs::String msg;
    std::stringstream ss;
    ss << "hello world " << count;
    msg.data = ss.str();
    ROS_INFO("%s", msg.data.c_str());
    chatter_pub.publish(msg);
    ros::spinOnce();
    loop_rate.sleep();
    ++count;
  }

  return 0;
}
```

Publisher breakdown

The code is commented at the previous link!

Notes:

- Messages are classes with a header (optional, timestamp) and several data fields
- **Only one message type** can be published on a specific topic (still, you have to pay attention...)
- Messages buffer may **add delays!**

Compiling the code

We are ready to compile the chatter node but first we need to adapt the *compiler's recipe* to our purpose.

1. Open `~catkin_ws/src/[package_name]/CMakeLists.txt`
2. Uncomment `add_executable` line (add talker.cpp as node main file)
3. Uncomment `target_link_libraries` block (adds `catkin` libraries in the executable)

Compiling the code

W we
ne

1.

2. p

3.

Toggle line numbers

```
 1 cmake_minimum_required(VERSION 2.8.3)
 2 project(beginner_tutorials)
 3
 4 ## Find catkin and any catkin packages
 5 find_package(catkin REQUIRED COMPONENTS roscpp rospy std_msgs genmsg)
 6
 7 ## Declare ROS messages and services
 8 add_message_files(FILES Num.msg)
 9 add_service_files(FILES AddTwoInts.srv)
10
11 ## Generate added messages and services
12 generate_messages(DEPENDENCIES std_msgs)
13
14 ## Declare a catkin package
15 catkin_package()
16
17 ## Build talker and listener
18 include_directories(include ${catkin_INCLUDE_DIRS})
19
20 add_executable(talker src/talker.cpp)
21 target_link_libraries(talker ${catkin_LIBRARIES})
22 add_dependencies(talker beginner_tutorials_generate_messages_cpp)
23
24 add_executable(listener src/listener.cpp)
25 target_link_libraries(listener ${catkin_LIBRARIES})
26 add_dependencies(listener beginner_tutorials_generate_messages_cpp)
```

Compiling the code

Build!

```
$cd ~/catkin_ws
$catkin_make
```

Source! `$source ~/catkin_ws/devel/setup.bash`

Run! `$rosrun [package_name] talker`

Only the current shell will be sourced this way!

Add the source to .bashrc in your home folder.

With the node running, try to launch `$rostopic list`

Check topic contents with `$rostopic echo [topic_name]`

Creating a subscriber

A subscriber is a ros node that *reads from* a desired topic.

Let's subscribe to the topic published by our `Talker`

```
$ cd ~/catkin_ws/[your_package] //navigate to package
$ cd src                        //navgate to src dir
$ touch listener.cpp            //create a file
```

`src/listener.cpp` is our node source code!

At the moment just copy/paste the code here:
http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber(c++)

Subscriber breakdown

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

void chatterCallback(const std_msgs::String::ConstPtr& msg){
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}

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

  ros::init(argc, argv, "listener");
  ros::NodeHandle n;
  ros::Subscriber sub =
       n.subscribe("chatter", 1000, chatterCallback);
  ros::spin();

  return 0;
}
```

Subscriber breakdown

The code is commented at the previous link!

- Subscriber buffer **adds delays!**
- `Spin()` is **blocking! It** waits for incoming messages and calls the desired callback.
- Callbacks are executed **everytime** your nodes process an incoming message and they take some time to return...

Message passing

- We associate topics to callback functions.
- We use callbacks to parse a single message.
- Callbacks are executed everytime a message is processed (not instantanously when published...).

Message passing

When a message arrives it enters a **FIFO** queue and waits to be processed.

Your node is **not** blocked by an incoming message...

...but **it is generally blocked** by the callback execution!

Keep your callbacks simple!!

Message passing | Pipeline

1. You subscribe to a *topic* and specify in the *subscriber* constructor the FIFO queue size.

2. If the queue is full, the first message (oldest) is discarded.

3. When a *spinner* (`spin()` or `spinOnce()`) is called, the **first** available message is processed by the callback associated to the subscriber.

Message passing | Spinners

We will look at single threaded spinners:

- `Spin()`: blocking function
- `SpinOnce()`: non blocking, requires looping

Spinners are required functions when using callbacks: usually they are found at the bottom of your program.

Spin everytime you need message processing!!

Message passing | `Spin()`

When Spin() instruction is executed:

- It blocks your program,
- enters a wait state until a ctrl-c or a shutdown,
- executes callbacks every time a message is available on a subscribed topic list.

This pattern is **usually** adopted when processing is fast and there are no intense operations to perform on the data

Message passing | `SpinOnce()`

When SpinOnce() instruction is executed:

It processes the pending messages in the queue; when done, the context swiches back to your program

Toggle line numbers

```
1 ros::Rate r(10); // 10 hz
2 while (should_continue)
3 {
4   ... do some work, publish some messages, etc. ...
5   ros::spinOnce();
6   r.sleep();
7 }
```

`Rate` forces the while loop to run at a specific frequency.
This is useful when you want your node to run at a specific frequency.

More on spinning http://wiki.ros.org/roscpp/Overview/Callbacks%20and%20Spinning

Node Handle

`ros::NodeHandle` manages an internal reference count to make starting and shutting down a node as simple as:

Toggle line numbers

```
1 ros::NodeHandle nh;
```

- On creation, if the internal node has not been started already, `ros::NodeHandle` will start the node.
- Once all `ros::NodeHandle` instances have been destroyed, the node will be automatically `shutdown()`.

Node Handle

NodeHandles let you specify a namespace to their constructor, you can refer to the link for more info.

 Pay particular attention to:

```
ros::NodeHandle nh;
ros::NodeHandle nh(~);
```

You may find any of the two in several tutorials.

Node Handle

```
ros::NodeHandle nh(~);
```

Here, you are actively specifying a namespace.

~ is a special id for this node private namespace.

Hence, all names will be preceded by the node name:

```
/topic1 -> /mynode/topic1
```

This is useful to avoid naming conflicts but be aware of it or you will encounter **communications issues!**

Names

Graph Resource Names provide a hierarchical naming structure that is used for all resources in a ROS Computation Graph, such as Nodes, Parameters, Topics, and Services.

There are four types of Graph Resource Names in ROS:

- `Base:` `base`
- `Relative:` `relative/name`
- `Global:` `/global/name`
- `Private:` `~private/name`

Names

Node	Relative (default)	Global	Private
/node1	bar -> /bar	/bar -> /bar	~bar -> /node1/bar
/wg/node2	bar -> /wg/bar	/bar -> /bar	~bar-> /wg/node2/bar
/wg/node3	foo/bar -> /wg/foo/bar	/foo/bar -> /foo/bar	~foo/bar-> /wg/node3/foo/bar

More on names here

Adding custom messages

`.msg`

Simple text files that describe the fields of a ROS message. They are used to generate source code for messages in different languages.

`.srv`

Text file that describes a service. It is composed of two parts: a request and a response `msg` type.

`.msg` example

```
string first_name
string last_name
uint8 age
uint32 score
```

`.msg` files are stored in the `/msg` directory of the respective package.

Every line is composed by two fields: **type** and **name.**

As a **good practice**, create a different package with no executables for custom messages !!

Create a custom `.msg`

Create `/msg` folder and `.msg` file

```
$ cd beginner_tutorials
$ mkdir msg
$ echo "int64 num" > msg/Num.msg  //creates Num.msg
```

Open Num.msg with a text editor to examine the content.

Create a custom `.msg` build

To build messages, you need `message_generation` and `message_runtime` library dependencies.

Add the following lines to your `package.xml`.

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

Create a custom `.msg` | build

Now, let's add the same dependencies to the package's `CMakeLists.txt`:

```
# Do not just add this to your CMakeLists.txt, modify the existing text to add message_generatio
n before the closing parenthesis
find_package(catkin REQUIRED COMPONENTS


   std_msgs
   message_generation
)
```

```
catkin_package(
  ...
  CATKIN_DEPENDS message_runtime ...
  ...)
```

Create a custom `.msg` | build

Finally, specify in the `CMakeLists.txt` the files you wish to build...

```
add_message_files(
  FILES
  Num.msg
)
```

...and the messages they depend upon!

```
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

Create a custom `.msg` | build

If you are building a package of **only messages** this should be enough!

If in the same package you also have executables that depend on those messages, don't forget to state that dependency!

```
add_dependencies(client beginner_tutorials_gencpp)
```

Create a custom `.msg` | build

Build with the following commands:

```
$ cd ~/catkin_ws
$ catkin_make
$ catkin_make install
```

Done!

More on message types here

Create a custom `.srv`

As for services, create a `/srv` folder inside your package and a `.srv` file. In the file, add the following:

```
int64 A
int64 B
---
int64 Sum
```

The `.srv` files are divided in request (first part) and response (second part), **divided by "---"**

Create a custom `.srv` | build

Modify `Cmakelists.txt` and `package.xml` just like it was done for creating custom .msg (see slides 35, 36, 37)

 Except! in this case ...

In `CMakeLists.txt`, instead of uncommenting `add_message_files`, uncomment `add_service_files`.

```
add_service_files(
  FILES
  AddTwoInts.srv
)
```

Build as you did for messages!!

Next Time

- Writing a Service
- Writing a Client
- The Parameter Server
- Using Parameters
- Launchfiles
- Rosbag