ROS Best Practices

Lorenz Mösenlechner
Technische Universität München

July 24th, 2012

ROS Overlays

- Never edit files in */opt/ros/...*.

ROS Overlays

- Never edit files in */opt/ros/...*.
- Use ROS Overlays

ROS Overlay

An overlay is a directory in the user's home that contains stacks and packages that are installed from source.

ROS Overlays

- Never edit files in */opt/ros/...*.
- Use ROS Overlays
- Multiple overlays with different versions can exist in parallel.
  - Overlay for development.
  - Overlay for demos.
  - Overlay for experimenting with bleeding edge code of other people.

ROS Overlay

An overlay is a directory in the user's home that contains stacks and packages that are installed from source.

ROS Overlays

- Never edit files in */opt/ros/...*.
- Use ROS Overlays
- Multiple overlays with different versions can exist in parallel.
  - Overlay for development.
  - Overlay for demos.
  - Overlay for experimenting with bleeding edge code of other people.
- Tool support for creating, modifying and managing overlays: *rosws*.

ROS Overlay

An overlay is a directory in the user's home that contains stacks and packages that are installed from source.

ROS Overlays

Creating an overlay with rosws

1. Install rosws: `sudo pip install rosinstall`
2. Create a new overlay:
   `rosws init ~/fuerte /opt/ros/fuerte`
3. Load the created file `setup.bash` in `.bashrc` (optional):
   `echo "source ~/fuerte/setup.bash" >> ~/.bashrc`

ROS Overlays

Adding packages to an overlay

- Add a local directory (e.g. a sandbox for experimental packages) to the overlay:

```
mkdir ~/fuerte/sandbox
rosws set ~/fuerte/sandbox
```

- Install packages from a rosinstall file:

```
rosws merge robohow-cram.rosinstall
rosws update
```

- Install a (released) stack from source:

```
roslocate info turtlebot | rosws merge -
rosws update
```

Rosinstall files

- YAML descriptions of repositories to install.
- Ideal for repository snap shots and collaboration.
- Possibility to specify versions.
- Example:

```
- git:
    local-name: cram_pl
    uri: http://code.in.tum.de/git/cram-pl.git
    version: 0.1.5
- svn:
    local-name: knowrob
    uri: http://code.in.tum.de/pubsvn/knowrob/tags/latest
```

Naming Conventions

File names

- Package names are lower case.
- Packages and stacks must not contain dashes ("-"), only underscores ("_").
- Messages, services and actions are named in camel case: `geometry_msgs/PoseStamped`
- Don't use the word "action" in an action definition. `Foo.action`, not `FooAction.action`.
- C++ source files and header files are named in lowercase, using underscores:
  `my_package/include/my_package/foo_bar.h`
  `my_package/src/foo_bar.cpp`
- C++ classes are normally named in camel case:
  `class FooBar { ... };`

Naming Conventions

Topics, parameters, actions, services

- Nodes, topics, services, actions, parameters are all lower case with underscores as separator.
- Never use global names, always node local topic, service, action and parameter names. Use `ros::NodeHandle handle("~")`

Bad

```
ros::NodeHandle nh();
nh.advertise<Foo>("foo", 10);
```

⇓

```
Topics:
/foo
```

Good

```
ros::NodeHandle nh("~");
nh.advertise<Foo>("foo", 10);
```

⇓

```
Topics:
/node_name/foo
```

Best Practices

Topics vs. Services vs. Actions

- Use topics for publishing continuous streams of data, e.g. sensor data, continuous detection results, ...
- Use services only for short calculations.
- Use actions for all longer running processes, e.g. grasping, navigation, perception, ...

Best Practices

Misc

- Don't require a specific startup order for nodes. Use `waitForService`, `waitForTransform`, `waitForServer`, ...
- Use standard data types when possible.
- Don't define matrix data types for transforms but use `geometry_msgs/PoseStamped`.
- Use `ros::Time`, `ros::Duration` and `ros::Rate` instead of system time.
- Don't use command line parameters but the ROS parameter server.
- Use rosconsole utilities for logging (`ROS_INFO`, `ROS_DEBUG`, ...).
- Never call cmake by hand in a package!

Best Practices

ROS package

- ROS packages are cheap, create many.
- One package per functionality.
- Create separate packages that contain only messages, services and actions (separation of interface and implementation).
- Keep your dependencies clean:
  - only depend on what you need
  - specify all dependencies
  - don't use implicit dependencies
- Provide launch files.
- Group packages in stacks.

3rd Party Libraries

- If possible, try to use libraries from Debian packages.
- Specify rosdep dependencies (tool for installing system packages).
- If you need to compile a library from source create a ROS wrapper package that downloads and compiles the package.
- Don't use sudo in wrapper packages.
- Don't require manual system wide installations.
- Don't copy libraries into packages that need them.

Collaboration

- Use version control systems (e.g. git or svn)
- Create tags for stable versions that others can use.
- Provide a rosinstall file.
- Create a short Wiki page for each package:
  - Document what the node does.
  - Document topics, services and actions that are required and provided.
  - Document ROS parameters and their default values.
- Data can be recorded and exchanged using bag files.

Working with Version Control

General guidelines

- Don't check in auto generated files.
- Don't check in huge binary files (important in git!)
- Set ignores (.gitignore or svn:ignore property)
- Commit often.
- Make one commit per feature.

Working with Version Control

Tagging

- To not break code other people are using, provide stable releases that other people are supposed to use.
- SVN: `svn cp --parents <repo>/trunk <repo>/tags/0.0.1`
- Git: `git tag 0.0.1 master`
- Use version fields in rosinstall files to refer to a tag.

ROS Bag Files

- Recording of a bag:
  `rosbag record <topic> <topic> ...`
- Play a bag:
  `rosbag play foo.bag`
- Play a bag using recorded time (important when stamped data and TF was recorded):
  `rosbag play --clock foo.bag`