



Contents

1. Overview
 1. What is a Build System?
 2. Why Does ROS Have a Custom Build System?
2. Why catkin? Motivation to move away from rosbuilt
 1. Portability through Python and Pure CMake
 2. Decoupling from ROS
 3. Out-of-Source Builds
 4. Install Targets and Seamless Release
 5. Easier Overlays with Workspace Environment Chaining
 6. Explicit Export of Build Flags
 7. Compliance with Filesystem Hierarchy Standard (FHS)
 8. Metapackages and the Elimination of Stacks
3. Moving from rosbuilt to catkin
 1. Terminology: Dry and Wet Packages
 2. catkin/rosbuilt Compatibility
 3. Goodbye Stacks, Hello Metapackages
 4. Difference in rosbuilt and catkin Environment Setup Files and Environment Variables
 5. Differences in rosbuilt and catkin Build Step
 6. Differences in rosbuilt and catkin Target Paths
 7. Differences in rosbuilt and catkin at Runtime
4. Dependency Management
 1. package.xml
 1. Explanation Details
 2. CMakeLists.txt
 1. Explanation Details
 3. setup.py

1. Overview





catkin (/catkin) is the official build system of ROS and the successor to the original ROS build system, rosbuilt (/rosbuilt). catkin (/catkin) combines  CMake (<http://www.cmake.org/>) macros and Python scripts to provide some functionality on top of CMake's normal workflow. catkin (/catkin) was designed to be more conventional than rosbuilt (/rosbuilt), allowing for better distribution of packages, better cross-compiling support, and better portability. catkin (/catkin)'s workflow is very similar to  CMake (<http://www.cmake.org/>)'s but adds support for automatic 'find package' infrastructure and building multiple, dependent projects at the same time.

The name *catkin* comes from the tail-shaped flower cluster found on willow trees -- a reference to Willow Garage where catkin was created.

1.1 What is a Build System?

A build system is responsible for generating 'targets' from raw source code that can be used by an end user. These targets may be in the form of libraries, executable programs, generated scripts, exported interfaces (e.g. C++ header files) or anything else that is not static code. In ROS terminology, source code

is organized into 'packages' where each package typically consists of one or more targets when built.

Popular build systems that are used widely in software development are  GNU Make (<http://www.gnu.org/software/make/>),  GNU Autotools (<http://www.gnu.org/software/autoconf/>),  CMake (<http://www.cmake.org/>), and  Apache Ant (<http://ant.apache.org/>) (used mainly for Java). In addition, virtually all integrated development environments (IDEs) such as Qt Creator, Microsoft Visual Studio, and Eclipse add their own build system configuration tools for the respective languages they support. Often the build systems in these IDEs are just front ends for console-based build systems such as Autotools or CMake.

To build targets, the build system needs information such as the locations of tool chain components (e.g. C++ compiler), source code locations, code dependencies, external dependencies, where those dependencies are located, which targets should be built, where targets should be built, and where they should be installed. This is typically expressed in some set of configuration files read by the build system. In an IDE, this information is typically stored as part of the workspace/project meta-information (e.g. Visual C++ project file). With CMake, it is specified in a file typically called 'CMakeLists.txt' and with GNU Make it is within a file typically called 'Makefile'. The build system utilizes this information to process and build source code in the appropriate order to generate targets.

ROS utilizes a custom build system, catkin, that extends CMake to manage dependencies between packages.

1.2 Why Does ROS Have a Custom Build System?

For development of single software projects, existing tools like Autotools, CMake, and the build systems included with IDEs tend to be sufficient. However, these tools can be difficult to use on their own with large, complex, and/or highly heterogeneous code ecosystems -- mainly because of the sheer number of dependencies, complex code organization, and custom build rules a particular target could have. As these tools are very general and designed to be used by software developers, they also tend to be difficult to use by those without a software development background.

ROS is a very large collection of loosely federated packages. That means lots of independent packages which depend on each other, utilize various programming languages, tools, and code organization conventions. Because of this, the build process for a target in some package may be completely different from the way another target is built. catkin (/catkin) specifically tries to improve development on large sets of related packages in a consistent and conventional way. In other words, both rosbuilt (/rosbuild) and now catkin (/catkin) aim to make building and running ROS code easier by using tools and conventions to simplify the process. Efficiently sharing ROS-based code would be more difficult without it.

2. Why catkin? Motivation to move away from rosbuilt

Though rosbuilt served well as the build system for ROS since its inception, the rapid growth of the ROS codebase as well as years of experience have exposed some drawbacks to its approach which catkin attempts to alleviate. As rosbuilt was created in the early days of ROS, it has had to evolve from its original

design to support the needs of the ROS community over the years. This has led to suboptimal design decisions, hacks, and unnecessary complexity. These problems motivated the creation of a build system on a new foundation.

2.1 Portability through Python and Pure CMake

One of the biggest problems with `roscpp` is that it is not [easily] portable to all operating systems, especially Microsoft Windows. This is because `roscpp` utilizes a mixture of Bash scripts, GNU Make, and CMake to build code. In `roscpp`, when we invoke the build system, we have to call custom scripts provided with `roscpp` such as `rosmake`. `rosmake` is a Bash script that calls `make` which itself calls CMake which generates yet another makefile and finally invoking `make` again! Catkin is much more elegant and is invoked by simply invoking CMake.

Catkin is implemented as custom CMake macros along with some Python code. As CMake and Python are portable, catkin is easily portable to any system that supports both Python and CMake. In fact, catkin projects can be used seamlessly with other CMake projects -- when catkin projects are built they also generate export information that allows them to be found with the CMake `find_package()` function.

2.2 Decoupling from ROS

One of the philosophies behind ROS is to minimize the number of ROS-specific tools needed to create, manage, and utilize ROS packages and to always try to defer to well-established, widely-used, third-party, open-source tools (e.g. using `libtinyxml` instead of writing a custom XML parser) instead. Catkin is independent of the ROS ecosystem and can even be used on non-ROS projects. That means catkin lets you easily mix your codebase with non-catkin projects. Catkin adds a lot of features on top of vanilla CMake that make it an appealing development tool even for non-ROS related projects.

2.3 Out-of-Source Builds

When `roscpp` builds a package, it generates the targets and any intermediate files (e.g. object files) within the folder containing the code. This is called an **in-source build** and is often undesirable as it can potentially pollute your local source tree with generated files that are not part of the baseline. With catkin, you can build your targets to any folder -- even one that is external to your package folder. Building targets outside the source folder this way is known as an **out-of-source build**.

2.4 Install Targets and Seamless Release

One of the main advantages of catkin, via CMake, is the ability to specify **install targets**. After code is built and targets are created, they are located in the folders specified by the user to the build system. At this point, though there are usable targets, they have not been *installed* to the system. All installation means is that the targets are then copied to an area such as a system folder where they can be used by users. For those who have used **GNU Make** to build code, this is the difference between `make` and `make install` where the former builds the code and the latter copies targets to the installation folder. With `roscpp`, there is no ability to install targets. When a ROS distribution is released, the core ROS team has to build all packages and then use specialized scripts to extract out targets and build installable packages (e.g. `.deb`

files) for the target operating system. With catkin, one can simply do a "make install" step and have all targets installed. This makes it not only easier for end users to work with ROS, but also allows the core ROS team to release ROS distributions more smoothly.

In addition, catkin allows users to specify which targets are installable and which are not. In a given package, not all targets may be useful and not all may be intended for exporting. For example, it may not be desirable to install unit tests, libraries used only by unit tests, and optional components. With rosbuilt, this is not possible and distribution packages tend to contain unnecessary targets which bloat the distribution.

2.5 Easier Overlays with Workspace Environment Chaining

The concept of overlays has existed in ROS since its early days. Each overlay is associated with a setup file (i.e. setup.bash, setup.py, or setup.sh) that sets the appropriate environment when sourced.


Catkin keeps the concept of overlays utilizing these environment setup files. However, there is a subtle difference from rosbuilt -- when a catkin environment setup file is sourced, it overwrites instead of extending existing environment variables. You may then be asking how chaining can work. Simply put, catkin generates lots of setup files in different contexts that chain multiple environments together with a single setup file. When you build code within your workspace, setup files are generated within the "devel space". When these are sourced, any setup files from other workspaces that were used at build time are automatically sourced. For example, if you source /opt/ros/groovy/setup.bash then build your workspace, sourcing the setup.bash file in the devel space will automatically pull in /opt/ros/groovy/setup.bash. Likewise if you install your workspace, the install space will contain a setup.bash file as well that will overlay the install space on top of any workspaces that were used to build that workspace.

2.6 Explicit Export of Build Flags

With rosbuilt, packages do not explicitly define resources and resource paths such as include paths, libraries, and library paths. Instead, any additional compiler/linker flags are hardcoded into the manifest.xml file and automatically appended by the build system based on package dependencies specified in the same file. The difference is subtle -- rosbuilt handles dependencies but does it through ROS-level mechanisms. Catkin on the other hand makes exported information an explicit part of its configuration and checks it through CMake-level mechanisms.

2.7 Compliance with Filesystem Hierarchy Standard (FHS)

catkin install targets are compliant with the Filesystem Hierarchy Standard (FHS), a standard directory layout for organizing files on Unix-like systems. This makes ROS more consistent with the wider open source ecosystem.

For more information about the motivation behind compliance with FHS, see  REP 122 (<http://ros.org/reps/rep-0122.html>).

2.8 Metapackages and the Elimination of Stacks

As of ROS Groovy, ROS no longer makes a distinction between the concept of a "package" and a "stack" (the latter being a collection of packages). With rosbuilt, each package had a CMakeLists.txt file and a manifest.xml file and each stack had a stack.xml file. Each stack was packaged into a Debian package. With catkin (since Groovy) packages have only a CMakeLists.txt file and a "package.xml" that is similar to manifest.xml and packages are packaged directly.

However, a new concept of **metapackages** has been introduced which behaves similar to the concept of a stack. Metapackages are a special type of package that depend on other packages.

Metapackage folders should not contain other files or folders, in particular not the packages they depend on. This is because packages cannot be nested in the filesystem.

3. Moving from rosbuilt to catkin

The workflow for catkin is different from that of rosbuilt in some ways, but similar in others. Though at first you may find it a little annoying to have to learn a new build system, you will likely find that building, running, and testing code with catkin is just as easy if not easier.

3.1 Terminology: Dry and Wet Packages

During the transition period between rosbuilt to catkin, the term **dry** packages is used to describe packages that use the deprecated rosbuilt build system. **Wet** packages are ones that have been updated for use with catkin.

3.2 catkin/rosbuilt Compatibility

As rosbuilt has been so inherently tied to ROS and catkin is so significantly different, converting all packages from dry (rosbuilt) to wet (catkin) is not an overnight endeavor. That means rosbuilt still exists in ROS Groovy and coexists with catkin, but it has been deprecated. All of the core and popular packages have been migrated to catkin for Groovy, but many still use rosbuilt. However, both can coexist.

An effort has been made to retain backwards compatibility between catkin and rosbuilt. Wet and dry packages will be able to coexist. Core packages have been the first to be made wet and with time all packages will be moved to catkin.

3.3 Goodbye Stacks, Hello Metapackages

The concept of stack (/Stacks) has been removed from ROS, retaining only the concept of package (/Packages). It was decided that stacks are an unnecessary level of aggregation and make ROS more confusing, both in its internal implementation and to its users. The concept of stacks has been replaced by **metapackages** which are packages with a package.xml containing a metapackage tag. See catkin/package.xml (/catkin/package.xml) for more information.

3.4 Difference in rosbuilt and catkin Environment Setup Files and Environment Variables

Both rosbuilt and catkin depend on environment variables to operate. These environment variables are set through an **environment setup file** called **setup.*** The extension of the setup file depends on the shell the user is using - either bash, zsh, or sh.

However, the environment variables in rosbuilt are no longer important in catkin, except for the case when a rosbuilt-based package needs to be built. In rosbuilt, the most important environment variable is `ROS_PACKAGE_PATH` which is a list of all paths where ROS packages reside for use at both build and runtime. Without this variable, `rosmake` and `roslaunch` would not be able to find resources. When a setup file is sourced for a dry package, this appends the `ROS_PACKAGE_PATH` with an additional entry to the path containing the setup file.

In catkin, the most important environment variable is the `CMAKE_PREFIX_PATH` which helps CMake find other CMake projects. Catkin packages are a special case of a CMake project, hence the need for this variable. When a catkin setup file is sourced, this variable gets appended with the corresponding package path. In a sense, the `CMAKE_PREFIX_PATH` essentially has the same role as `ROS_PACKAGE_PATH`.

3.5 Differences in rosbuilt and catkin Build Step

With rosbuilt, packages are built using a special script called `rosmake`. With catkin, the `cmake` command is used followed by the `make` command which will build all packages in the workspace:

```
cd ~/catkin_ws/build
cmake ../src
make
```

In rosbuilt, you could build individual packages by passing the package name to `rosmake`:

```
rosmake rviz
```

With catkin the makefile generated by CMake allows you to build individual targets (where one package might have multiple targets). I.e., if `rviz` is in your workspace, you can build it by passing `rviz` (which the target building the `rviz` executable) as an argument to `make`:

```
cd ~/catkin_ws/build
cmake ../src
make rviz
```

Tab completion is supported so one can see all the available targets by hitting the tab key after typing `make`.

3.6 Differences in rosbuilt and catkin Target Paths

With rosbuilt, targets are built within the package folder. This does not matter to the majority of ROS users as they are used to using utilities like `rosmake` and `roslaunch` which figure out where things are based on the environment that was set by environment setup file). With catkin, they are built to the devel space and optionally installed to the install space. Again, this does not matter as catkin also has a concept of environment setup files that behave in the same way as those provided with rosbuilt projects.

3.7 Differences in rosbuilt and catkin at Runtime

With both rosbuilt and catkin, ROS applications are expected to be launched using `roslaunch` or `roslaunch` rather than being directly started. `roslaunch` and `roslaunch` are able to find executable files and runtime libraries based on environment variables that are set by sourcing the environment setup files. Though internally `roslaunch` has to do some work to deal with both rosbuilt and catkin targets, this is abstracted to the user and there is no distinction between rosbuilt and catkin at runtime.

4. Dependency Management

There are multiple places where the same dependencies have to be specified with catkin (in the current version). We look at an example called "example_pkg" with 2 dependencies: the catkin package `cpp_common` and the system library `log4cxx`

4.1 package.xml

```
<package>
  ...
  <name>example_pkg</name>
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>cpp_common</build_depend>
  <build_depend>log4cxx</build_depend>
  <test_depend>gtest</test_depend>
  ...
  <run_depend>cpp_common</run_depend>
  <run_depend>log4cxx</run_depend>
</package>
```

The package.xml definitions were specified in REP127 (<http://ros.org/reps/rep-0127.html>) and some details are provided here (<http://catkin/package.xml>)

In general, `package.xml` is responsible for:

- Ordering of the configure step (cmake) sequence for catkin-packages in catkin workspaces
- Define packaging dependencies for bloom (/bloom) (what dependencies to export when creating debian pkgs)
- Define system (non-catkin-pkgs) build dependencies for rosdep
- Document build or install or runtime dependency for roswiki / graph tool (rqt_graph (/rqt_graph))

4.1.1 Explanation Details

Any `<build_depend>` and `<buildtool_depend>` entry will make sure the given dependency is configured first when it is present in the same workspace. If the entry is not as source in this workspace, the dependency is ignored for configure step ordering. In the example this works only for `roscpp_common`, `log4cxx` will be ignored, since it is not a catkin package.

`<buildtool_depend>` implies that this dependency should be used off the current architecture, not the target architecture, when cross-compiling. When not cross-compiling, it is equivalent to `<build_depend>`.

`<test_depend>` are dependencies that are required for running tests. Catkin packages use macros that define make targets prefixed with `run-tests`. They can be run by invoking `catkin_make run_tests[_...]` or just using `make run_tests[_...]`. `<test_depend>` should declare dependencies that are only used during this testing process.

`<build_depend>` entries can also be used with `rosdep` for system installs so that users (or scripts) can easily install system dependencies before attempting to build.

A `<run_depend>` entry has two purposes. One is to declare what executable in the package will need to run. But it will also define any dependency that a 3rd package using our new package needs to also have in the environment (If your package B depends on package A, and another package C depends on your package B). This can happen for libraries at runtime, or for headers at build-time of the 3rd package. So "run" in `<run_depend>` can be slightly misleading. When a debian package is created, it should make sure that all `<run_depend>` dependencies are also installed by `apt-get`.

Very often, a `<run_depend>` entry will also be a `<build_depend>` entry. However, in the future we can imagine cases like this:

```
<build_depend>cpp_common-dev</build_depend>
<run_depend>cpp_common</run_depend>
```

which is why the dependencies have been separated.

4.2 CMakeLists.txt

```
find_package(catkin REQUIRED COMPONENTS cpp_common geometry_msgs)
find_package(Log4cxx QUIET)
generate_messages(DEPENDENCIES geometry_msgs)
catkin_package(
  CATKIN_DEPENDS cpp_common geometry_msgs
  DEPENDS Log4cxx
)
add_library(example_lib src/example.cpp)
target_link_libraries(example_lib ${catkin_LIBRARIES} ${LOG4CXX_LIBRARIES})
add_dependencies(example_lib geometry_msgs_gencpp)
```

In general, `CMakeLists.txt` is responsible for preparing and executing the build process.

As you can see with `log4cxx`, the name of the system library and the name to be used to find the package are often not the same, that's a reason why we have to specify that dependency again, instead of reading it from the `package.xml`.

4.2.1 Explanation Details

The calls to


```
find_package(catkin REQUIRED COMPONENTS cpp_common)
target_link_libraries(example_lib ${catkin_LIBRARIES} ${LOG4CXX_LIBRARIES})
```

are equivalent to these calls:

```
find_package(catkin REQUIRED)
find_package(cpp_common REQUIRED)
target_link_libraries(example_lib ${catkin_LIBRARIES} ${cpp_common_LIBRARIES} ${LOG4CXX_LIBRARIES})
```

with the main benefit that with the COMPONENTS notation you can use "catkin_" prefixed variables, and those will also have cpp_common values. In this example we only create one library "example_lib", but if we created more, it could be better to list for each library individually which dependency it has, rather than using `${catkin_LIBRARIES}`.

The dependencies in

```
generate_messages(DEPENDENCIES geometry_msgs)
```

belongs to the message_generation macros, not catkin macros, but in this context they are still worth mentioning, showing that message generation does not use other declared catkin dependencies for the purpose. When Groovy was released, all transitive dependencies of messages had to be listed here, a later patch allows to only state direct dependencies.

The declarations in:

```
catkin_package(
  CATKIN_DEPENDS cpp_common
  DEPENDS Log4cxx
)
```

makes sure that if a 3rd package want to use our "example_pkg", the build flags for cpp_common and log4cxx are automatically used as well. So you need these only if that is the case (for dependencies that are `<run_depend>` entries in your package.xml).

The declaration in

```
add_dependencies(example_lib geometry_msgs_gencpp)
```

makes sure that the c++ message headers from geometry_msgs are build before attempting to build example_lib, else you would get compiler errors about missing headers, when working with a catkin workspace (Outside catkin workspaces, this line has no effect).

4.3 setup.py

If a package declares python modules for other packages to use, those need to be declared in a setup.py. Technically this also leaves the possibility to declare dependencies here, too:

```
from distutils.core import setup

setup(
    ...
    requires=['rospy'],
)
```

the names used there could be names of catkin packages or packages distributed over pypi (/pypi.python.org). The names for packages given in pypi is often different from the names used e.g. for apt-get packages, e.g. "nose" (pypi) vs. "python-nose" (apt-get). However catkin currently does not care about the requires section, and packages like "rospy" are also currently not yet provided over pypi, so at the time of this writing, those dependencies are not that important. If your catkin package happens to have only dependencies that exist in pypi, you might as well use this section to make your package also installable via pip.

Except where otherwise

noted, the ROS wiki is

Wiki: catkin/conceptual_overview (last edited 2020-03-26 14:42:14 by FrancoisVanEeden (/FrancoisVanEeden))

licensed under the

Creative Commons Attribution 3.0 (<http://creativecommons.org/licenses/by/3.0/>)

Brought to you by:  Open Robotics

(<https://www.openrobotics.org/>)