CS 378 - Autonomous Vehicles in Traffic I
Week 4b - ROS packages and stacks

Logistics

- I messed up the date on the meeting with the recruiter from the CIA. He will be talking to students on Monday 2/13 from 3:15-4:00 in RLM 4.102
- For this reason, we'll have the schedule car visit on Wednesday (2/15) next week instead of Monday (2/13)
  - I plan to be there at the least between 11AM to 3PM. I might be there earlier and may stay for longer.
  - Instructions for getting to PRC are at http://z.cs.utexas.edu/users/piyushk/marvin/index.php/PRC
  - Call me at 512-923-3240 if you are unable to locate us
  - We will have a regular class on Monday 2/13

Today

- On Monday you saw the inner workings of a simple ROS package with 3 nodes. Today we will see:
  - A top-down view of a typical ROS repository
  - The ROS build system
- Today's lecture will build on the material from Monday and last week. I hope to tie everything together with this lecture.

ROS for collaborative development

- ROS has always focused on developing code in a manner which is easy for collaborative development
- For this reason, there is a particular code hierarchy that most groups try and follow. This helps in easier understanding of code
- Furthermore, the ROS build system is designed to aid in reusing code from other groups.
- Over time the ROS code hierarchy has refined. Let's take a look at how typical repositories are organized
ROS code hierarchy

- Repository: Contains all the code from a particular development group (We have 3 repositories from utexas)
- Stack: Groups all code on a particular subject / device
- Packages: Separate modules that provide different services
- Nodes: Executables that exist in each model (You have seen this already)

utexas-art-ros-pkg - 3 branches

- trunk
  - art_vehicle
  - velodyne
  - projects
    - art_experimental
    - velodyne_experimental
- sandbox
  - android
  - gps_drivers
  - spr12
  - art_examples
  - ...

art_vehicle stack

Example velodyne runtime

- roscore
- velodyne
  - velodyne_common
    - read
    - velodyne/velodyne
  - cloud
  - velodyne/velodyne
  - height_map
    - velodyne/velodyne
    - sensor_msgs:PointCloud
    - velodyne/velodyne
    - sensor_msgs:PointCloud
Command line tools - rospack

- **rospack** is a command-line program used to find packages among the "forest" of code in a typical ROS distribution, calculate dependencies, mangle Makefiles, and in general promote peace and harmony in a ROS distribution.

- Some examples
  - `rospack find intro_to_ros`
  - `rospack list | grep ros`
  - `rospack depends intro_to_ros`

[http://www.ros.org/wiki/rospack]

Command line tools - rosstack

- **rosstack** is a command-line tool for retrieving information about ROS stacks available on the filesystem. It implements a wide variety of commands ranging from locating ROS stacks in the filesystem, to listing available stacks, to calculating the dependency tree of stacks.

- Some examples
  - `rosstack contains intro_to_ros`
  - `rosstack list-names | grep examples`
  - `rosstack depends art_examples`

[http://www.ros.org/wiki/rossack]

Command line tools - roscd

- **roscd** is part of the rosbash suite. It allows you to change directory (i.e., cd) directly to a package or stack by name rather than having to know the filesystem path.

- Some examples
  - `roscd art_examples`
  - `roscd intro_to_ros`
  - `roscd intro_to_ros/src`

[http://www.ros.org/wiki/roscd]

Why is all this important?

- I hope to use this point to make the discussion from last week a bit clearer.
- Let's execute the following commands
  - `roscd intro_to_ros`
  - `make`
  - `ldd bin/talker`
    - This includes both ROS dependencies and system dependencies
- Imagine how much extra work would it be if:
  - Someone wanted to update one of these libraries, and you were not using dynamic linking
  - You structured your code different from the organization I just showed you.
Review

The important points are:
• To use shared libraries with dynamic linking, it is necessary to separate our class implementations into separate files.
• Compile code appropriately to produce these shared libraries
• We need makefiles or an equivalent to easily building code
• Following a particular code structure can help in automation. To see how, let's take a look at the rosbuild system.

CMakeLists.txt

• CMakeLists.txt is the equivalent of a Makefile. It is used by cmake to build code.
• Let us take a look at the CMakeLists.txt file for our intro_to_ros package - available here
• There are a number of good examples of CMakeLists.txt: http://www.ros.org/wiki/rosbuild/CMakeLists/Examples
• We will quickly see some of the parameters and functions that can be used in CMakeLists.txt

rosbuild

• rosbuild contains scripts for managing the CMake-based build system for ROS.
• 3 files are used to build your ROS package
  ○ CMakeLists.txt - standard CMake build file, but allows ROS macros
  ○ manifest.xml - specifies your dependencies. also provides compiler and linker flags.
  ○ Makefile - 1 single line that invokes CMake. You should never have to change this.

rosbuild flags

• ROS_BUILD_TYPE: Set the build type. Options are (default: RelWithDebInfo):
  ○ Debug : w/ debug symbols, w/o optimization
  ○ Release : w/o debug symbols, w/ optimization
  ○ RelWithDebInfo : w/ debug symbols, w/ optimization
  ○ RelWithAsserts : w/o debug symbols, w/ optimization, w/ assertions (i.e., w/o -DNDEBUG). New in ros 1.1.
  ○ MinSizeRel : w/o debug symbols, w/ optimization, stripped binaries
• ROS_BUILD_STATIC_EXES: Build static-only executables (e.g., for copying over to another machine)? true or false; default: false
• ROS_BUILD_SHARED_LIBS: Build shared libs? true or false; default: true
• ROS_BUILD_STATIC_LIBS: Build static libs? true or false; default: false
• ROS_COMPILE_FLAGS: Default compile flags for all source files; default: "-W -Wall -Wno-unused-parameter -fno-strict-aliasing"
• ROS_LINK_FLAGS: Default link flags for all executables and libraries; default: ""

[http://ros.org/wiki/rosbuild]
CMakeLists.txt (contd)

The main ROS macros that you will end up using:
- `robuild_add_library`
  - Creates a library from the given C++ file
  - Places library by default in lib folder
- `robuild_add_executable`
  - Creates an executable from the given C++ file - should have main
  - executables are placed in bin folder
- `target_link_libraries`
  - Link an executable in your package to a library inside the same package.
  - Not required for libraries in other packages.
  - Required for external libraries

manifest.xml

- `manifest.xml` provides dependency information to the robuild system - the intro_to_ros manifest.xml is [here](#)
- Provides some basic documentation for the package. This is good for published packages. For instance the `manifest.xml` of the ROS package velodyne common is used to auto-generate section 1 on the [wiki page](#)
- Provides the system dependencies of a package
  - `<roosedep name="libpcap" />`
- Provides other ROS package dependencies
  - `<depend package="sensor_msgs" />`
- Exports compiler and linker flags
  - These are used when some other ROS package depends on your package.

manifest.xml (contd)

- Compiler flags
  - `-I<path to include directory>`

- Linker flags
  - `-L<path to static/shared object libraries>`
  - `-l<library name>`
  - `-Wl,-rpath,$(prefix)/lib` (path to dynamically linked libraries)

- `So the velodyne_common manifest has these lines. It has a library (velodyne) and a system dependency (pcap):`

```xml
<export>
  <cpp cflags="-I$(prefix)/include" lflags="-L$(prefix)/lib
       -Wl,-rpath,$(prefix)/lib -lvelodyne -lpcap"/>
</export>
```

What is rosmake?

- `rosmake` is a dependency aware build tool for ros packages and stacks

- Some common use cases:
  - `rosmake <package-name>` - will build the ROS packages along with the ROS dependencies
  - `rosmake <stack-name>` - will build all the packages in that stack
  - `rosmake <name> --pre-clean` - runs make clean && make on all the packages in the dependency tree
  - `rosmake <name> --rospack-install` - installs any required system dependencies

- Run: `rosmake --help` to see all options
**rosmake vs make**

- To build a package, you can also go to that package directory and type make
  - `roscd intro_to_ros`
  - `make`

- `make` will only build the package (i.e. not the dependencies)
- `make` is faster than rosmake
  - the entire dependency tree is not checked
- I typically use rosmake when building a package for the first time, or am unclear about the dependencies. After that, I use `make`

**Command line tools - roscreate-pkg**

- `roscreate-pkg` creates a new package in your current directory. For this course, you will only be creating new packages in the `spr12` directory inside sandbox.
- This auto-generates standard files inside the package: CMakelists.txt, Makefile, manifest.xml and mainpage.docx (don't worry about the last one)

- Example:
  - `roscd spr12`
  - `roscreate-pkg piyush_khandelwal_p2`

[http://www.ros.org/wiki/roscreate]

---

**How to write the intro_to_ros package**

- Create the package
  - `roscd art_examples`
  - `roscreate-pkg intro_to_ros`

- Inside the package, create a folder to contain the source files
  - `roscd intro_to_ros OR cd intro_to_ros`
  - `mkdir src`

- Inside the src directory, write the 3 files:
  - `roscd intro_to_ros/src OR cd src`
  - `gedit talker.cpp`
  - `gedit messenger.cpp`
  - `gedit listener.cpp`

**How to write the intro_to_ros package**

- Build these 3 files into executables; update CMakelists.txt
  - `roscd intro_to_ros OR cd ../`
  - `gedit CMakelists.txt`
  - Use the `robuild_add_executable` macro to create executables for these 3 files

- Run `make`; you will get an error message that `ros.h` was not found.
  - Update `manifest.xml` to add `roscpp` dependency
  - `gedit manifest.xml`

- Run `make` and continue editing code to solve compilation and runtime issues
Programming Assignment 2

- I have made a few changes to the code in the *intro_to_ros* tutorial we talked about on Monday. If you have already checked out a copy of our code, then you can get the latest version by issuing out the following commands:
  - `cd ~/svn/sandbox`
  - `svn up`
  - `cd ~/svn/utexas-art-ros-pkg`
  - `svn up`
- These changes remove some syntactical ambiguities, and should prevent some problems you might face in programming assignment 2. I will update the slides shortly.
- In programming assignment 2, there is only 1 question, and an extra credit section for 3 points. Let's take a look

**intro_to_ros** - changes

- I have made some changes to *listener.cpp* and *messenger.cpp*

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

  to

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

  and

  ```cpp
  ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
  ```

  to

  ```cpp
  ros::Subscriber sub =
  n.subscribe<std_msgs::String>("chatter", 1000, chatterCallback);
  ```

Review

- The last 4 lectures have contained fairly difficult material
- However, these slides are for reference only - you just need to understand the general concepts, and look up information in the slides or online to write the actual commands.
- There are links inside the slides to get more information when needed.

With this material, you should:
- Have a general understanding of executables and libraries in C++
- Be able to write simple C++ programs and compile/link them into libraries and executables
- Write simple makefiles to automate the process
Review (continued)

With this material, you should:
• Be able to create new ROS packages
• Write basic ROS code, and be able to update
  `CMakeLists.txt` and `manifest.xml` based on your code
• (Extra Credit) Be able to write libraries through the ROS
  build system, to be used by your code and other packages
• Use some basic command line tools to move around the
  ROS ecosystem, and display basic information about stacks
  and packages.

Think about what steps you are comfortable with. Discuss with
us during office hours.