Introduction to ROS Programming

March 5, 2013

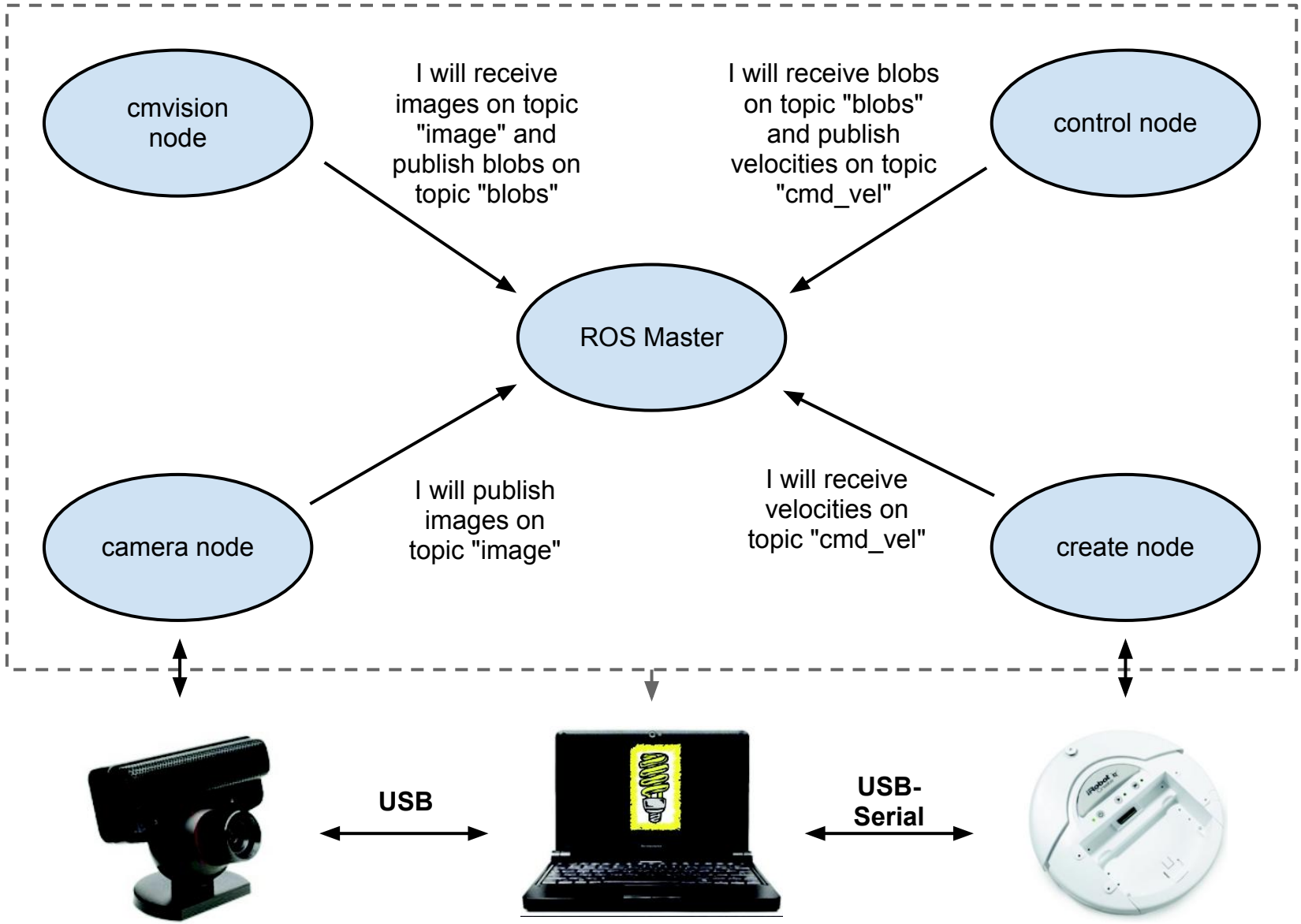
Today

- We'll go over a few C++ examples of nodes communicating within the ROS framework
- We will recap the concepts of ROS nodes, topics and messages.
- We'll also take a look at the rosbuild repository structure and creating and building a simple package using rosmake

Review - ROS Overview

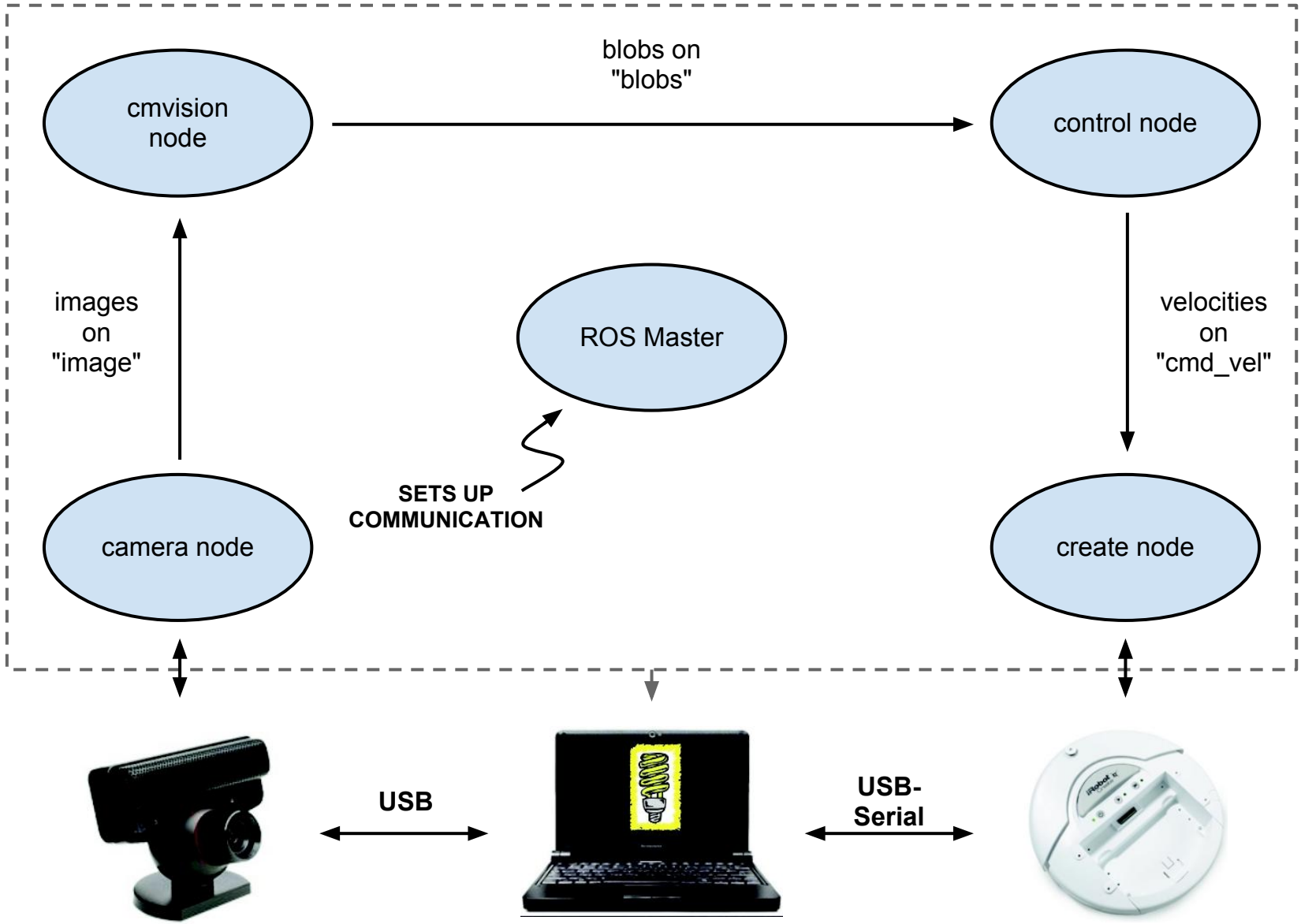
- ROS is a peer-to-peer *robot middleware* package
- We use ROS because it allows for easier *hardware abstraction* and *code reuse*
- In ROS, all major functionality is broken up into a number of chunks that communicate with each other using messages
- Each chunk is called a *node* and is typically run as a separate process
- Matchmaking between nodes is done by the ROS Master

Review - How ROS works



[adapted from slide by Chad Jenkins]

Review - How ROS works



[adapted from slide by Chad Jenkins]

ROS Nodes

- A *node* is a process that performs some computation.
- Typically we try to divide the entire software functionality into different modules - each one is run over a single or multiple nodes.
- Nodes are combined together into a graph and communicate with one another using streaming topics, RPC services, and the Parameter Server
- These nodes are meant to operate at a fine-grained scale; a robot control system will usually comprise many nodes

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

ROS Topics

- Topics are named buses over which nodes exchange messages
- Topics have anonymous publish/subscribe semantics - A node does not care which node published the data it receives or which one subscribes to the data it publishes
- There can be multiple publishers and subscribers to a topic
  - It is easy to understand multiple subscribers
  - Can't think of a reason for multiple publishers
- Each topic is strongly typed by the ROS message it transports
- Transport is done using TCP *or* UDP

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

ROS Messages

- Nodes communicate with each other by publishing *messages* to topics.
- A message is a simple data structure, comprising typed fields. You can take a look at some basic types here
  - std_msgs/Bool
  - std_msgs/Int32
  - std_msgs/String
  - std_msgs/Empty (huh?)
- In week 8 we will look into creating our own messages
- Messages may also contain a special field called header which gives a *timestamp* and *frame of reference*

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

Getting the example code

- These tutorials are based on the beginner ROS tutorials
- All of today's tutorials available here:
  - http://farnsworth.csres.utexas.edu/tutorials/

- Use the following commands to install a tarball in your workspace
  - `roscd`
  - `wget http://farnsworth.csres.utexas.edu/tutorials/intro_to_ros.tar.gz`
  - `tar xvzf intro_to_ros.tar.gz`
  - `rosws set intro_to_ros`
  - `<restart terminal>`
  - `rosmake intro_to_ros`

talker.cpp (intro_to_ros)

```cpp
#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(1);
  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;
}
```

listener.cpp (intro_to_ros)

```cpp
#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<std_msgs::String>("chatter", 1000, chatterCallback);
  ros::spin();
  return 0;
}
```

talker.cpp

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

- *ros/ros.h* is a convenience header that includes most of the pieces necessary to run a ROS System
- *std_msgs/String.h* is the message type that we will need to pass in this example
  - You will have to include a different header if you want to use a different message type
- *sstream* is responsible for some string manipulations in C++

talker.cpp

```
ros::init(argc, argv, "talker");
ros::NodeHandle n;
```

- ros::init is responsible for collecting ROS specific information from arguments passed at the command line
  - It also takes in the name of our node
  - Remember that node names need to be unique in a running system
  - We'll see an example of such an argument in the next example
- The creation of a ros::NodeHandle object does a lot of work
  - It initializes the node to allow communication with other ROS nodes and the master in the ROS infrastructure
  - Allows you to interact with the node associated with this process

talker.cpp

```
ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
ros::Rate loop_rate(1);
```

- NodeHandle::advertise is responsible for making the XML/RPC call to the ROS Master advertising std_msgs:: String on the topic named "chatter"
- loop_rate is used to maintain the frequency of publishing at 1 Hz (i.e., 1 message per second)

talker.cpp

```
int count = 0;
while (ros::ok()) {
```

- *count* is used to keep track of the number of messages transmitted. Its value is attached to the string message that is published
- *ros::ok()* ensures that everything is still alright in the ROS framework. If something is amiss, then it will return *false* effectively terminating the program. Examples of situations where it will return false:
  - You *Ctrl*+*c* the program (SIGINT)
  - You open up another node with the same name.
  - You call *ros::shutdown()* somewhere in your code

talker.cpp

```
std_msgs::String msg;
std::stringstream ss;
ss << "hello world " << count;
msg.data = ss.str();
```

- These 4 lines do some string manipulation to put the count inside the *String* message
  - The reason we do it this way is that C++ does not have a good equivalent to the *toString()* function
- *msg.data* is a *std::string*
- Aside: I typically use *boost::lexical_cast()* in place of the *toString()* function. *lexical_cast()* pretty much does the thing above for you (Look up this function if you are interested)

talker.cpp

```
ROS_INFO("%s", msg.data.c_str());
chatter_pub.publish(msg);
```

- *ROS_INFO* is a macro that publishes a information message in the ROS ecosystem. By default *ROS_INFO* messages are also published to the screen.
  - There are debug tools in ROS that can read these messages
  - You can change what level of messages you want to be have published

- *ros::Publisher::publish()* sends the message to all subscribers

talker.cpp

```
ros::spinOnce();
loop_rate.sleep();
++count;
```

- *ros::spinOnce()* is analogous to the *main* function of the ROS framework.
  - Whenever you are subscribed to one or many topics, the *callbacks* for receiving messages on those topics are not called immediately.
  - Instead they are placed in a queue which is processed when you call *ros::spinOnce()*
  - What would happen if we remove the *spinOnce()* call?
- *ros::Rate::sleep()* helps maintain a particular publishing frequency
- *count* is incremented to keep track of messages

listener.cpp - *in reverse!*

```
int main(int argc, char **argv) {
  ros::init(argc, argv, "listener");
  ros::NodeHandle n;
  ros::Subscriber sub =
    n.subscribe<std_msgs::String>("chatter", 1000, chatterCallback);
  ros::spin();
  return 0;
}
```

- *ros::NodeHandle::subscribe* makes an XML/RPC call to the ROS master
  - It subscribes to the topic *chatter*
  - 1000 is the *queue size*. In case we are unable to process messages fast enough. This is only useful in case of irregular processing times of messages. Why?
  - The third argument is the *callback* function to call whenever we receive a message
- *ros::spin()* a convenience function that loops around *ros::spinOnce()* while checking *ros::ok()*

listener.cpp

```
#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());
}
```

- Same headers as before
- *chatterCallback()* is a function we have defined that gets called whenever we receive a message on the subscribed topic
- It has a *well typed* argument.

Running the code

- You will have to execute the following steps to get this example working
- After you download our code, build the example package
  - *rosmake intro_to_ros*
- In separate terminal windows, run the following programs:
  - *roscore*
  - *rosrun intro_to_ros talker*
  - *rosrun intro_to_ros listener*

Example 2 - Adding a Messenger node

- A number of times in ROS you will have a bunch of nodes processing data in sequence. For instance a *blob detection node* provides the location of blobs *for every* camera image it receives
- To demonstrate this, we'll change our previous example in the following ways:
  - Introduce a *messenger* node that listens for messages on the topic *chatter* and forwards them on the topic *chatter2.* (I couldn't think of a cute name for this topic)
  - At the command line remap the listener to subscribe to *chatter2* instead of *chatter*

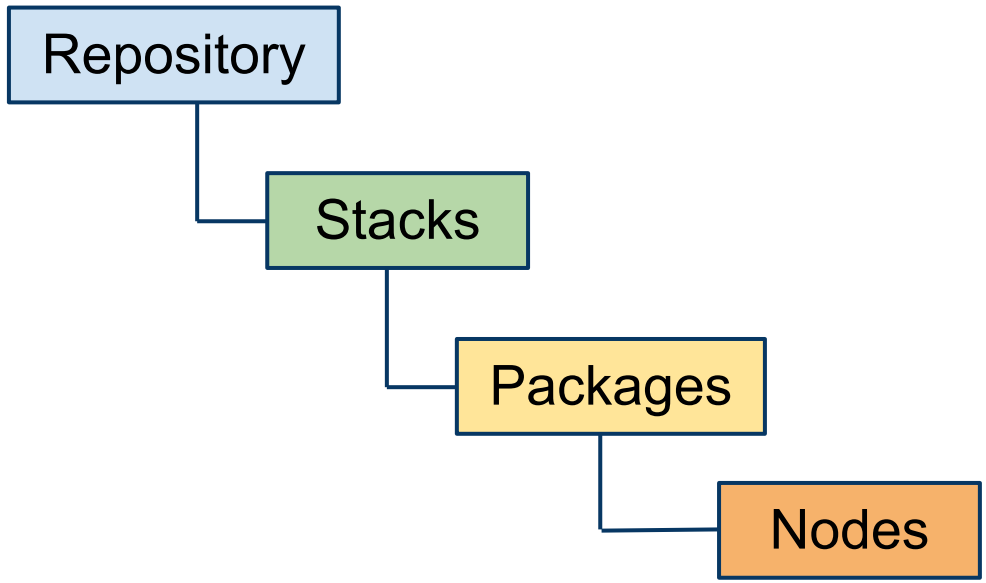
messenger.cpp (intro_to_ros)

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

ros::Publisher chatter_pub;
std_msgs::String my_msg;

void chatterCallback( const std_msgs::String::ConstPtr msg) {
  ROS_INFO("I heard: [%s]", msg->data.c_str());
  my_msg.data = msg->data + ". Dont kill the messenger!";
  chatter_pub.publish(my_msg);
}

int main(int argc, char **argv) {
  ros::init(argc, argv, "messenger");
  ros::NodeHandle n;
  ros::Subscriber sub =
    n.subscribe<std_msgs::String>("chatter", 1000, chatterCallback);
  chatter_pub = n.advertise<std_msgs::String>("chatter2", 1000);
  ros::spin();
  return 0;
}
```

Running the code

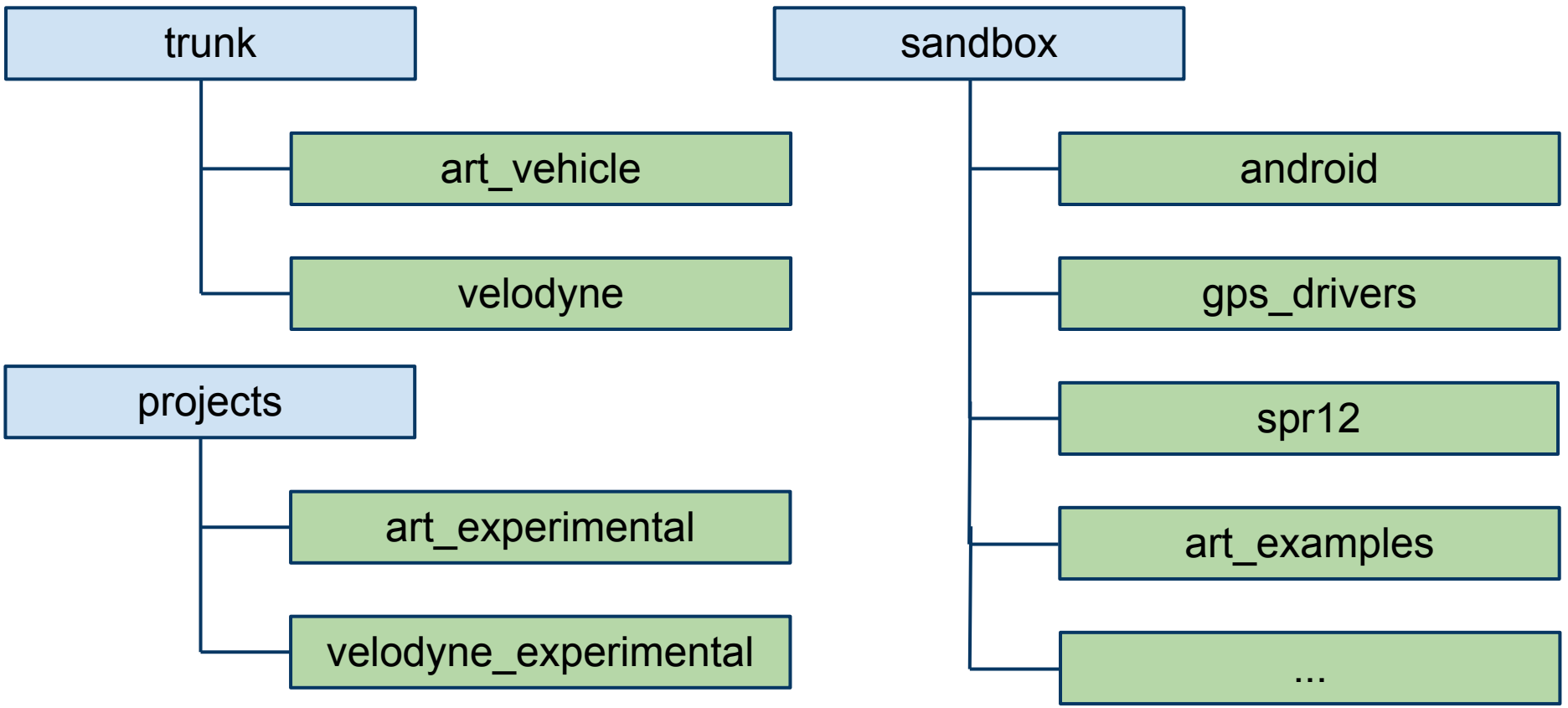
- You will have to execute the following steps to get this example working
- In separate terminal windows, run the following programs:
  - *roscore*
  - *rosrun intro_to_ros talker*
  - *rosrun intro_to_ros listener chatter:=chatter2*
  - *rosrun intro_to_ros messenger*

ROS code hierarchy

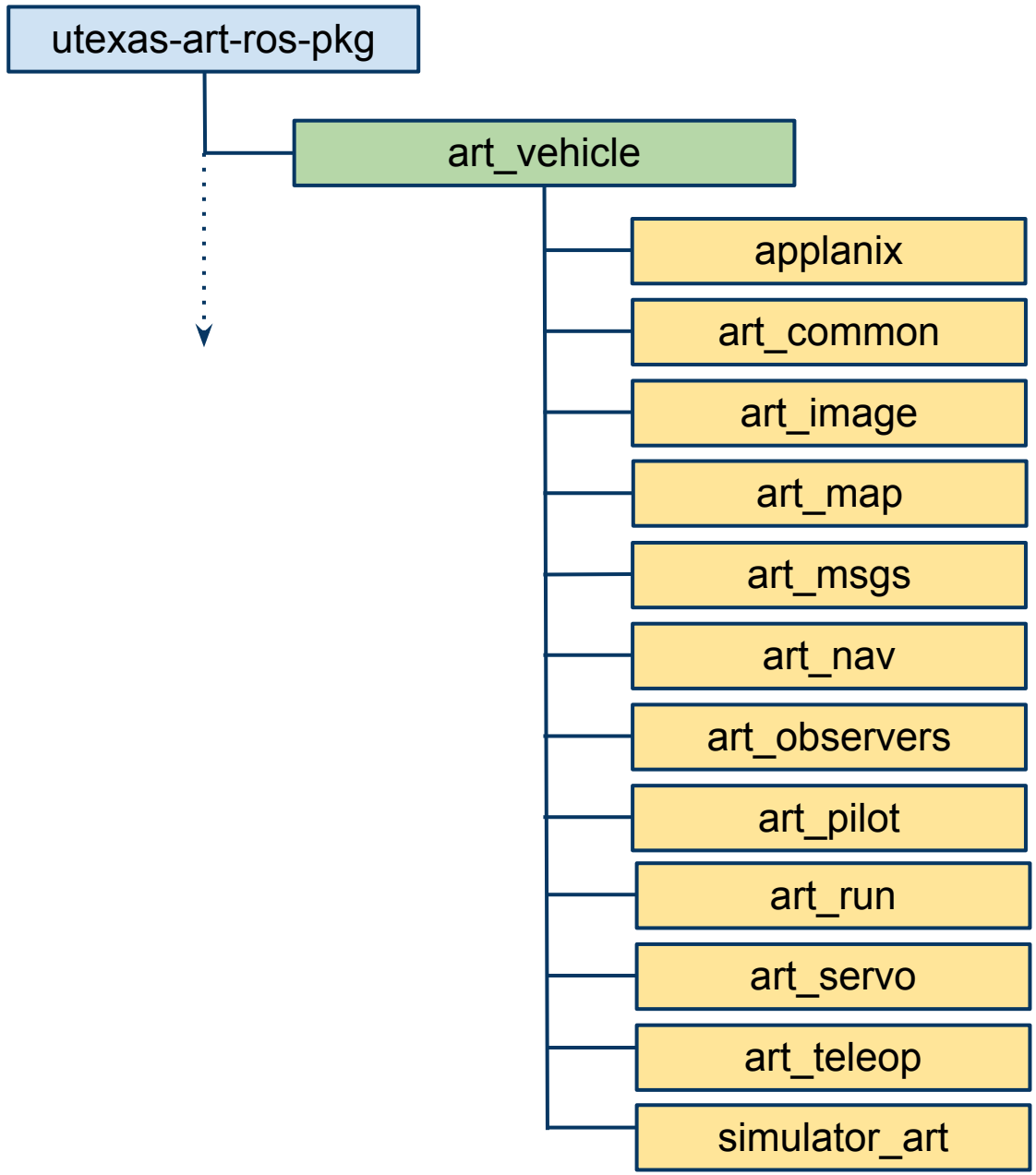


- Repository: Contains all the code from a particular development group (We have 3 repositories from utexas)
- Stack: Groups all code on a particular subject / device
- Packages: Separate modules that provide different services
- Nodes: Executables that exist in each model (You have seen this already)

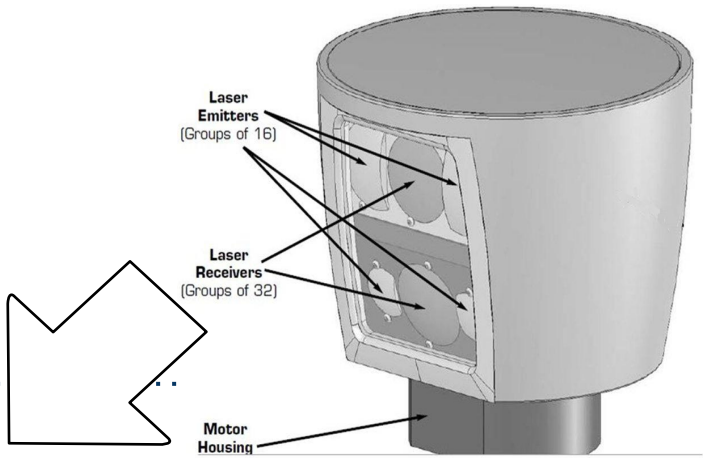
utexas-art-ros-pkg - 3 branches



art_vehicle stack



Example *velodyne* runtime



roscore

velodyne | velodyne_common

read

/velodyne/raw_scan

velodyne | velodyne_common

cloud

/velodyne/cloud
sensor_msgs::PointCloud

velodyne | velodyne_heightmap

height_map

/velodyne/obstacles
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/rosstack]

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]

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.

[http://ros.org/wiki/rosbuild]

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 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:

- *rosbuild_add_library*
  - Creates a library from the given C++ file
  - Places library by default in lib folder
- *rosbuild_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 rosbuild 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
  - `<rosdep 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>` (multiple times for multiple libraries)
  - `-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):

```
<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> --rosdep-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.dox* (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 *rosbuild_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

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.