Intro to ROS (Robot Operating System):
Wyatt Newman, November, 2012

1) What is ROS?

From the ROS wiki:

> ROS is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers.

> The primary goal of ROS is to support code *reuse* in robotics research and development. ROS is a distributed framework of processes (aka *Nodes*) that enables executables to be individually designed and loosely coupled at runtime. These processes can be grouped into *Packages* and *Stacks*, which can be easily shared and distributed. ROS also supports a federated system of code *Repositories* that enable collaboration to be distributed as well. This design, from the filesystem level to the community level, enables independent decisions about development and implementation, but all can be brought together with ROS infrastructure tools.

On-line comment by Brian Gerkey:

> I usually explain ROS in the following way:

> 1. plumbing: ROS provides publish-subscribe messaging infrastructure designed to support the quick and easy construction of distributed computing systems.
> 2. tools: ROS provides an extensive set of tools for configuring, starting, introspecting, debugging, visualizing, logging, testing, and stopping distributed computing systems.
> 3. capabilities: ROS provides a broad collection of libraries that implement useful robot functionality, with a focus on mobility, manipulation, and perception.
> 4. ecosystem: ROS is supported and improved by a large community, with a strong focus on integration and documentation. ros.org is a one-stop-shop for finding and learning about the thousands of ROS packages that are available from developers around the world.

From the text <u>ROS by Example</u>:

> The primary goal of ROS (pronounced "Ross") is to provide a unified and open source programming framework for controlling robots in a variety of real world and simulated environments.

Regarding ROS "plumbing" (from <u>ROS by Example</u>):

> The core entity in ROS is called a node. A node is generally a small program written in Python or C++ that executes some relatively simple task or process. Nodes can be started and stopped independently of one another and they communicate by passing messages. A node can publish messages on certain topics or provide services to other nodes.

For example, a publisher node might report data from sensors attached to your robot's microcontroller. A message on the `/head_sonar` topic with a value of 0.5 would mean that the sensor is currently detecting an object 0.5 meters away. Any node that wants to know the reading from this sensor need only subscribe to the `/head_sonar` topic. To make use of these values, the subscriber node defines a callback function that gets executed whenever a new message arrives on the subscribed topic. How often this happens depends on the rate at which the publisher node updates its messages.

A node can also define one or more services. A ROS service produces some behavior or sends back a reply when sent a request from another node. A simple example would be a service that turns an LED on or off. A more complex example would be a service that returns a navigation plan for a mobile robot when given a goal location and the starting pose of the robot.

1.1) Approach to this presentation: ROS has many features, tools, style expectations and quirks. ROS has a steep learning curve, since there is much detail to master before one can be productive. The ROS Wiki has links to documentation and a sequence of tutorials. However, these can be hard to follow for one new to ROS, as the definitions are scattered and the level of detail presented vary broadly from unexplained examples to explanations oriented towards sophisticated users. The intent of this document is to introduce the reader to essential components of ROS with detailed explanations of simple code examples along with the corresponding theory of operation. This introduction only scratches the surface, but it should get the reader started on building useful ROS nodes—as well as bring the reader to a state where the tutorials become more readable.

For the present examples, I am using the following versions: Linux Ubuntu 12.04 and ROS fuerte. These are the recommended versions for the DARPA Robotics Challenge simulator. To try out the examples presented herein, install ROS as described here: http://www.ros.org/wiki/fuerte/Installation/Ubuntu

This introduction will focus on the concepts and examples of subscriber nodes and publisher nodes, starting with minimal examples. After the minimal listener/publisher node examples, specific listeners/publishers will be described for interfacing to the Gazebo model of the DARPA Robotics Challenge/ORSF simulator (see http://gazebosim.org/ and http://gazebosim.org/wiki/InstallDRC).

The examples here are in C++. Python is also supported, but is not described here.

2) Some ROS communications concepts: Communications among nodes is at the heart of ROS. A "node" is a ROS program that uses ROS's middleware for communications. A node can be "launched" independent of other nodes and in any order among launches of other nodes. Many nodes can run on the same computer, or nodes may be distributed across a network of computers. A node is only useful if it can communicate with other nodes—and ultimately with sensors and actuators.

Communication among nodes uses the concepts of "messages", "topics", "roscore", "publishers" and "subscribers". ("Services" are also useful, and these are closely related to publishers and subscribers). All communications among nodes is serialized network communications. A "publisher" publishes a "message", which is a packet of data that is interpretable using an associated key. Since messages are received as a stream of bits, it is necessary to consult the key (the message type description) to know how to parse the bits and populate the corresponding data structure. A simple example of a message is Float64, which is defined in the "std_msgs" that comes with ROS. The message type helps the publisher pack a floating-point number into the defined stream of 64 bits, and it also helps the subscriber interpret how to unpack the bitstream as a representation of a floating-point number. A more complex example of a message is a "twist", which consists of multiple fields describing 3-D translational and rotational

velocities. Some messages also accommodate optional extras, such as time stamps and message identifier codes.

When data gets published by a publisher node, it is made available to any interested subscriber nodes. Subscriber nodes must be able to make connections to the published data. Often, the published data is made published by different nodes—whether these publishers have changed due to software evolution or whether some publisher nodes are relevant in some contexts and other nodes in other contexts. For example, a publisher node responsible for commanding joint velocities might sometimes be a stiff position controller but in other scenarios a compliant-motion controller may be needed. This hand-off can occur by changing which node is publishing the velocity commands. This presents the problem that the subscriber does not know who is publishing its input. In fact, the need to know what node is publishing complicates construction of large systems. This problem is addressed by the concept of a "topic."

A topic may be introduced and various publishers may take turns publishing to that topic. Thus a subscriber only needs to know the name of a topic and does not need to know what node (or nodes) publishes to that topic. For example, the topic for commanding velocities may be "vel_cmd", and the robot's low-level controller should subscribe to this named topic to receive velocity commands. Different publishers might be responsible for publishing velocity-command messages on this topic, whether these are nodes under experimentation or trusted nodes that are swapped in to address specific task needs.

Although creating the abstraction of a "topic" helps some, a publisher and a subscriber both need to know how to communicate via a topic. This is accomplished through communications middleware in ROS via the executable "roscore." Roscore is responsible for coordinating communications, like an operator. Although there can be many ROS nodes distributed across multiple networked computers, there can be only one instance of "roscore" running, and where roscore runs establishes the "master" computer of the system.

A publisher node initiates a topic by informing roscore of the topic (and the corresponding message type). This is called "advertising" the topic. To accomplish this, the publisher instantiates an object of the class "ros::Publisher". This class definition is part of the ROS distribution, and using publisher objects allows the designer to avoid having to write communications code. After instantiating a publisher object, the user code invokes the member function "advertise" and specifies the message type and declares the desired topic name. At this point, the user code can start sending out messages to the named topic using the publisher member function "publish", which takes as an argument the message to be published.

Since a publisher node communicates with roscore, roscore must running before the any ROS node is launched. To run roscore, open a terminal in Linux and enter "roscore." The response to this command will be a confirmation "started core services." It will also print out the ROS_MASTER_URI, which is useful for informing nodes running on non-master computers how to reach roscore. The terminal running roscore will be dedicated to roscore, and it will be unavailable for other tasks. Roscore should continue running as long as the robot is actively controlled (or as long as it is desired to access the robot's sensors).

After roscore has been launched, a publisher node may be launched. The publisher node will advertise its topic and may start sending out messages (at any rate convenient to the node, though at a frequency limited by system capacity). Publishing messages at 1kHz rate is normal for low-level controls and sensor data.

Introducing a sensor to a ROS system requires specialized code (and possibly specialized hardware) that can communicate with the sensor. For example, a LIDAR sensor may require RS488 communications, accept commands in a specific format, and start streaming data in a predefined format. A dedicated microcontroller (or a node within the main computer) must communicate with the LIDAR, receive the

3

data, then publish the data with a ROS message type on a ROS topic. Such specialized nodes convert the specifics of individual sensors into the generic communications format of ROS.

When a publisher begins publishing, it is not necessary that any nodes are listening to the messages. Alternatively, there may be many subscribers to the same topic. The publisher does not need to be aware of whether it has any subscribers nor how many subscribers there may be. This is handled by the ROS middleware. A subscriber may be receiving messages from a publisher, and the publisher node may be halted and possibly replaced with an alternative publisher of the same topic, and the subscriber will resume receiving messages with no need to restart the subscriber.

A ROS subscriber also communicates with roscore. To do so, it uses an object of class ros::Subscriber. This class has a member function called "subscribe" that requires an argument of the named topic. The programmer must be aware that a topic of interest exists and what is the name of the topic. Additionally, the subscriber function requires the name of a "callback" function. This provides the necessary hook to the ROS middleware, such that the callback function will start receiving messages. The callback function suspends until a new message has been published, and the designer may include code to operate on the newly-received message.

Subscriber functions can be launched before the corresponding publisher functions. ROS will allow the subscriber to register its desire to receive messages from a named topic, even though that topic does not exist. At some point, if/when a publisher informs roscore that it will publish to that named topic, the subscriber's request will be honored, and the subscriber will start receiving the published messages.

A node can be both a subscriber and a publisher. For example, a control node would need to receive sensor signals as a subscriber and send out control commands as a publisher. This only requires instantiating both a subscriber object and a publisher object within the node. It is also useful to pipeline messages for sequential processing. For example, as low-level image processing routine (e.g. for edge finding) could subscribe to raw camera data and publish low-level processed images. A higher-level routing might subscribe to the edge-processed images, look for specific shapes within those images, and publish its results for further use downstream. A sequence of nodes performing successive levels of processing can be modified incrementally by replacing one node at a time. As long as the new module subscribes to the same input topic name and publishes to the same topic name (with the same message type), the other nodes within the chain will be unaffected.

The flexibility to launch publisher and subscriber nodes in any order eases system design. Additionally, individual nodes may be halted at any time and additional nodes may be "hot swapped" into the running system. This can be exploited, for example, to launch some specialized code when it is needed and then halt the (potentially expensive) computations when no longer needed. Additionally, diagnostic nodes (e.g. interpreting and reporting on published messages) may be run and terminated *ad hoc*. This can be useful e.g. to examine the output of selected sensors to confirm proper functioning.

It should be appreciated that the flexibility of launching and halting publishers and subscribers at any time within a running system can also be a liability. For time-critical code—particularly control code that depends on sensor values to generate commands to actuators—an interruption of the control code or of the critical sensor publishers could result in physical damage to the robot or its surroundings. It is up to the programmer to make sure that time-critical nodes remain viable. Disruptions of critical nodes should be tested and responded to appropriately (e.g. with halting all actuators).

From the system architecture point of view, ROS helps one implement a desired software architecture and supports teamwork in building large systems. Starting from a predetermined software architecture, one can construct a "skeleton" of a large system constructed as a collection of nodes. Initially, each of the

nodes might be "dummy" nodes, capable of sending and receiving messages via predefined topics (the software interfaces). Each module in the architecture could then be upgraded incrementally by swapping out an older (or dummy) node for a newer replacement—and no changes would be required elsewhere throughout the system. This supports distributed software development and incremental testing, which are essential for building large, complex systems.

3) Writing ROS nodes: In this section, design of a minimal publisher node and a minimal subscriber node will be detailed. The concept of a ROS "package" is introduced, along with a "manifest" and CMakeLists. Several ROS tools and commands are introduced, including: rosmake, rosrun, rostopic, rosnode, and rxgraph. Specific C-code examples for a publisher and a subscriber are detailed and results of running the compiled nodes are shown.

3.1) Creating a ROS package: The first thing to do when starting to design new ROS code is to create a "package." A package is a ROS concept of bundling multiple, necessary components together to ease building ROS code and coupling it with other ROS code. Packages should be logical groups of code, e.g. separate packages for low-level joint control, for high-level planning, for navigation, for mapping, for vision, etc. Although these packages normally would be developed separately, nodes from separate packages would ultimately be run together simultaneously on a robot (or on a network of computers collectively controlling the robot).

One creates a new "package" using the "roscreate-pkg" command (a command that is part of the ROS installation). For a given package, this only needs to be done once. You can go back to this package and add more code incrementally (without needing to create the package again). However, the code added to a package should logically "belong" to that package (e.g., avoid putting low-level joint-control code in the same package as mapping code). We proceed as follows:

To make a ROS "package," open a terminal and move (cd) to the directory where you will be creating your ROS code. For my desktop system, this is: /home/wyatt/fuerte_workspace/sandbox (or just ~/fuerte_workspace/sandbox). In this terminal, type:

```
roscreate-pkg   minimal_nodes   roscpp std_msgs
```

The effect of this command is to create and populate a new directory: ~/fuerte_workspace/sandbox/minimal_nodes. If you move to this directory, you will see that it is already populated with multiple items, notably: manifest.xml, CMakeLists.txt and the subdirectory "src". The roscreate-pkg command has just created a new "package" by the name "minimal_nodes" which will reside in a directory of this name.

As we create new code, we will depend on some ROS tools and definitions. Two dependencies were listed during the roscreate-pkg command: "roscpp" and "std_msgs". The "roscpp" dependency establishes that we will be using a C++ compiler to create our ROS code, and we will need C++-compatible interfaces (such as the classes ros::Publisher and ros::Subscriber, introduced earlier). The "std_msgs" dependency says that we will need to rely on some datatype defnitions (standard messages) that have been predefined in ROS. (an example is std_msgs::Float64).

If you examine the manifest file (which is just ASCII text, and thus you can open it with any editor), you will see two lines explicitly declaring dependency on the package "roscpp" and on the package "std_msgs". Eventually, we will want to bring in large bodies of 3rd-party code (other "packages"). In

order to integrate with these packages (e.g., utilize objects and datatypes defined in these packages), we will want to add them to the manifest. This can be done by editing manifest.xml in our package directory, emulating the existing lines that declare dependence on roscpp and std_msgs.

The "src" directory is where we will put our user-written C++ code. We will write illustrative nodes "minimal_publisher.cpp" and "minimal_subscriber.cpp" as examples. It will be necessary to edit the CMakeLists.txt file to inform the compiler that we have new nodes to be compiled. This will be described further later.

3.2) Writing a minimal ROS publisher: In a terminal window, move to the "src" directory within the "minimal_nodes" package that has been created. Open up an editor, create a file called "minimal_publisher.cpp" and type in the following code: (note: the print, copy, paste applied here has wrapped some comment lines. Make sure that all code in blue appears as comments in your file).

File: /home/wyatt/fuerte_workspace/…odes/src/minimal_publisher.cpp Page 1 of 1

```cpp
#include <ros/ros.h>
#include <std_msgs/Float64.h>
int main(int argc, char **argv)
{
ros::init(argc,argv,"minimal_publisher1"); // name of this node will be "minimal_publisher1"
ros::NodeHandle n; // two lines to create a publisher object that can talk to ROS
ros::Publisher my_publisher_object = n.advertise<std_msgs::Float64>("topic1",1);
//"topic1" is the name of the topic to which we will publish
// the "1" argument says to use a buffer size of 1; could make larger, if expect network backups
std_msgs::Float64 input_float; //create a variable of type "Float64", as defined in: /opt/ros/fuerte/
share/std_msgs
// any message published on a ROS topic must have a pre-defined format, so subscribers know how to
// interpret the serialized data transmission
input_float.data=0.0;
while(ros::ok()) // do work here in infinite loop (desired for this example), but terminate if
detect ROS has faulted
{
// this loop has not sleep timer, and thus it will consume excessive CPU time
// expect one core to be 100% dedicated to this small task
input_float.data = input_float.data+0.001; //increment by 0.001 each iteration
my_publisher_object.publish(input_float); // publish the value--of type Float64-- to the topic
"topic1"
}
}
```

The above code is dissected here. The #include items bring in header files from the two packages on which we depend—ROS and std_msgs. As you use more packages, you will need to include their associated header files in your code.

The minimal publisher consists of only one "main()" program, which is declared in the standard fashion with generic argc, argv arguments. This gives the node the opportunity to use command-line options—though that is not needed for this example.

The three lines ros::init(), ros::NodeHandle and ros::Publisher refer to objects defined in the ROS distribution.

The argument to ros::init() is the user's choice of what this node's name will be once it is launched. The present node will be known as "minimal_publisher1" when it is launched. The node name is required,

and every node in the system must have a unique name. ROS tools are provided that take advantage of the node names, e.g. to monitor which nodes are active and which nodes are publishing or subscribing to which topics.

The ros::Publisher instantiation creates an object that I have chosen here to be called "my_publisher_object" (the name is the programmer's choice). In instantiating this object, roscore is informed that the current node (which we called "minimal_publisher1") intends to publish messages of type std_msgs::Float64 on a topic named "topic1".

The program then creates an object of type std_msgs:Float64 and calls it "input_float." One must consult the message definition in "std_msgs" to understand how to use this object. The object is defined as having a member called "data". The program initializes input_float.data to the value 0.0.

An infinite loop is then entered, though this loop can be terminated by detecting that roscore has terminated by the line: while(ros::ok())… This can be convenient for shutting down a collection of nodes by merely halting roscore.

Inside the "while" loop, the value of input_float.data is incremented by 0.001 each iteration. It is then published by invoking "my_publisher_object.publish(input_float);". It was previously established via ros::Publisher that the object "my_publisher_object" would publish messages of type std_msg::Float64 to the topic called "topic1".

3.3) Compiling and running the minimal publisher node: ROS nodes are compiled by running "rosmake package_name" where "package_name" in this case is "minimal_nodes". However, we have to inform "rosmake" of the existence of our new source code, "minimal_publisher.cpp". To do so, edit the file "CMakeLists.txt", which was created for us in the minimal_nodes directory when we ran roscreate-pkg. A copy of this file is shown below.

File: /home/wyatt/fuerte_workspace/…x/minimal_nodes/CMakeLists.txt Page 1 of 1

```
cmake_minimum_required(VERSION 2.4.6)
include($ENV{ROS_ROOT}/core/rosbuild/rosbuild.cmake)
# Set the build type. Options are:
# Coverage : w/ debug symbols, w/o optimization, w/ code-coverage
# Debug : w/ debug symbols, w/o optimization
# Release : w/o debug symbols, w/ optimization
# RelWithDebInfo : w/ debug symbols, w/ optimization
# MinSizeRel : w/o debug symbols, w/ optimization, stripped binaries
#set(ROS_BUILD_TYPE RelWithDebInfo)
rosbuild_init()
#set the default path for built executables to the "bin" directory
set(EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)
#set the default path for built libraries to the "lib" directory
set(LIBRARY_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/lib)
#uncomment if you have defined messages
#rosbuild_genmsg()
#uncomment if you have defined services
#rosbuild_gensrv()
#common commands for building c++ executables and libraries
#rosbuild_add_library(${PROJECT_NAME} src/example.cpp)
#target_link_libraries(${PROJECT_NAME} another_library)
#rosbuild_add_boost_directories()
#rosbuild_link_boost(${PROJECT_NAME} thread)
#rosbuild_add_executable(example examples/example.cpp)
```

rosbuild_add_executable(minimal_publisher src/minimal_publisher.cpp) #wsn added this line
#target_link_libraries(example ${PROJECT_NAME})

Note that almost all of the lines in the file are commented out (and thus appear in blue). I have added a single line: rosbuild_add_executable(minimal_publisher src/minimal_publisher.cpp). This should be done for each new node in your package. The first argument to "rosbuild_add_executable" is a chosen name for the executable to be created (I chose to call this "minimal_publisher"—which happens to be the same root name as the source code). The second argument is where to find the source code, relative to the package directory. Our source code is in the "src" subdirectory and it is called "minimal_publisher.cpp".

Having edited the CMakeLists.txt file, we can compile our new code. To do so, enter:

rosmake minimal_nodes

This will invoke the C++ compiler to build the package "minimal_nodes", recognizing hooks to the packages on which it depends. This will result in verbose output, which should conclude by saying "with 0 failures" once all is well. Otherwise, look for lines complaining of errors in your source code and make the necessary fixes (case sensitivity, use of semicolons, balanced braces, etc).

Assuming compilation success, change to the "bin" subdirectory within the "minimal_nodes" directory. You will see a new file there named "minimal_publisher." This is the name that was chosen for the output file in the edit of CMakeLists.txt.

To launch the new node, recall that "roscore" must be running first. In a new terminal window, enter:
roscore

This should respond with a page of text concluding with "started core services." Leave this window alone now. We can start and stop nodes at any time now without having to restart roscore.

Next, launch the new publisher node by entering: rosrun minimal_nodes minimal_publisher

The arguments to the command "rosrun" are the package name ("minimal_nodes") and the executable name ("minimal_publisher"). The "rosrun" command can seem confusing at times due to reuse of names. For example, if we wanted to make a LIDAR publisher node, we might have a package called "lidar_publisher", a source file called "lidar_publisher.cpp", an executable called "lidar_publisher", and a node name (declared within the source code) of "lidar_publisher". To run this node, we would type:

rosrun lidar_publisher lidar_publisher

This may seem redundant, but it still follows the format: rosrun package_name executable name. Once this executable has been launched, the ROS system will recognize a new node by the name of "lidar_publisher." This name reuse may seem to lead to confusion, but in many instances, there is no need to invent new names for the package, the source code, the executable name and the node name. In fact, this can help simplify recognizing named entities—as long as the context is clear (package, source code, executable, node name).

3.4) Examining the running minimal publisher node: After entering: rosrun minimal_nodes minimal_publisher, the result may seem disappointing. The window that launched this node seems to

hang and provides no feedback to the user. Still, "minimal_publisher" is running. To see this, we can invoke some ROS tools.

Open up a new terminal and enter: rostopic

You will get the following response:

This shows that the command "rostopic" has 8 options. If we type: rostopic list the result is:

We see that there are three active topics—two that ROS created on its own and the topic created by our publisher, "topic1."

Entering: rostopic hz topic1 results in the following output:

```
😣😣😕  wyatt@wyatt-OptiPlex-760: ~
wyatt@wyatt-OptiPlex-760:~$ rostopic hz topic1
subscribed to [/topic1]
average rate: 17583.703
        min: 0.000s max: 0.018s std dev: 0.00020s window: 17274
average rate: 16185.050
        min: 0.000s max: 0.018s std dev: 0.00022s window: 32309
average rate: 15903.701
        min: 0.000s max: 0.020s std dev: 0.00025s window: 47905
average rate: 15907.482
        min: 0.000s max: 0.035s std dev: 0.00036s window: 50000
average rate: 22091.800
        min: 0.000s max: 0.035s std dev: 0.00032s window: 50000
average rate: 31343.601
        min: 0.000s max: 0.013s std dev: 0.00011s window: 50000
average rate: 30233.324
        min: 0.000s max: 0.019s std dev: 0.00014s window: 50000
average rate: 27933.117
        min: 0.000s max: 0.019s std dev: 0.00014s window: 50000
^Caverage rate: 24677.205
        min: 0.000s max: 0.013s std dev: 0.00016s window: 50000
wyatt@wyatt-OptiPlex-760:~$ 
```

This output shows that our minimal publisher is publishing its data at nearly 30kHz (with some jitter). Viewing the system monitor would show that one CPU core is fully saturated running the minimal publisher. This is because the while-loop within our ROS node has no pauses in it. It is publishing as fast as it can.

Entering: rostopic bw topic1 yields the following output:

```
⊗⊖⊙   wyatt@wyatt-OptiPlex-760: ~
wyatt@wyatt-OptiPlex-760:~$ rostopic bw topic1
subscribed to [/topic1]
average: 163.34KB/s
        mean: 0.01KB min: 0.01KB max: 0.01KB window: 100
average: 110.53KB/s
        mean: 0.01KB min: 0.01KB max: 0.01KB window: 100
average: 94.45KB/s
        mean: 0.01KB min: 0.01KB max: 0.01KB window: 100
average: 177.74KB/s
        mean: 0.01KB min: 0.01KB max: 0.01KB window: 100
average: 69.11KB/s
        mean: 0.01KB min: 0.01KB max: 0.01KB window: 100
average: 47.03KB/s
        mean: 0.01KB min: 0.01KB max: 0.01KB window: 100
average: 94.41KB/s
        mean: 0.01KB min: 0.01KB max: 0.01KB window: 100
average: 163.23KB/s
        mean: 0.01KB min: 0.01KB max: 0.01KB window: 100
^Caverage: 6.53KB/s
        mean: 0.01KB min: 0.01KB max: 0.01KB window: 100
wyatt@wyatt-OptiPlex-760:~$ █
```

This display shows how much our available communications bandwidth is being consumed by our minimal publisher (nominally 160 KB/s). This rostopic option can be useful for identifying nodes that are over-consuming communications resources.

Entering: rostopic info topic1 yields:

```
⊗⊖⊙   wyatt@wyatt-OptiPlex-760: ~
wyatt@wyatt-OptiPlex-760:~$ rostopic info topic1
Type: std_msgs/Float64

Publishers:
 * /minimal_publisher1 (http://wyatt-OptiPlex-760:47480/)

Subscribers: None

wyatt@wyatt-OptiPlex-760:~$ ▯
```
1—15 of 52.019

This tells us that the topic called "topic1" involves messages of type "std_msgs/Float64". At present, there is a single publisher to this topic (which is the norm), and that publisher has a node name of
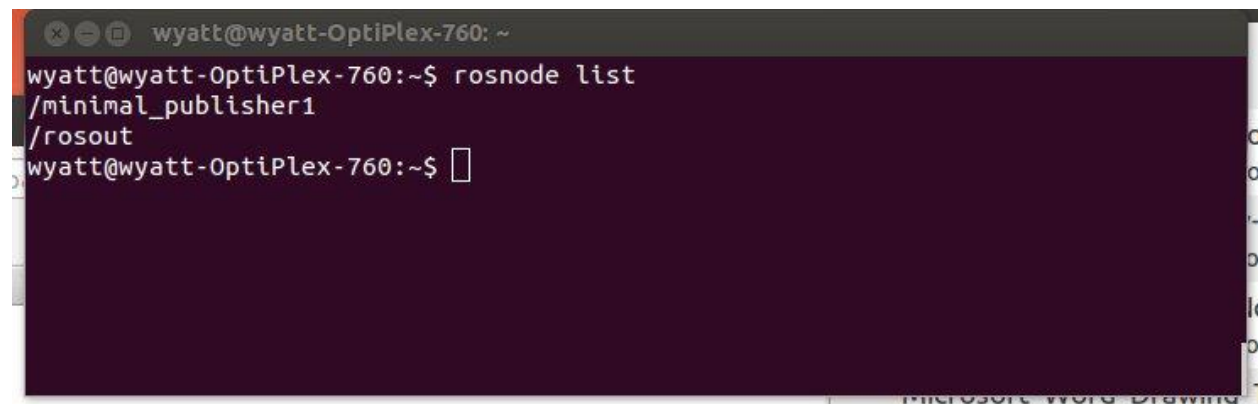
11

"minimal_publisher1". As noted above, this is the name we assigned to the node within the source code with the line: ros::init(argc,argv,"minimal_publisher1");

Entering: rostopic echo topic1 cases rostopic to try to print out everything published on "topic1". In this case, the display has no hope of keeping up with the publishing rate, and most of the messages are dropped between lines of display. If the echo could keep up, we would expect to see values that increase by increments of 0.001, which is the increment used in the while-loop of our source code.

The "rostopic" command tells us much about the status of our running system, even though there are no nodes receiving the messages sent out by our minimal publisher. More handy ROS commands are summarized in the "ROS cheat sheet"

(see: http://mirror.umd.edu/roswiki/attachments/Documentation/ROScheatsheet.pdf)

For example, entering: rosnode list results in the following output:

We see that there are two nodes running: rosout (a generic process used for nodes to display text to a terminal, launched by default by roscore) and minimal_publisher1 (our node).

Entering: rxgraph results in a graphical display of our system, shown below.
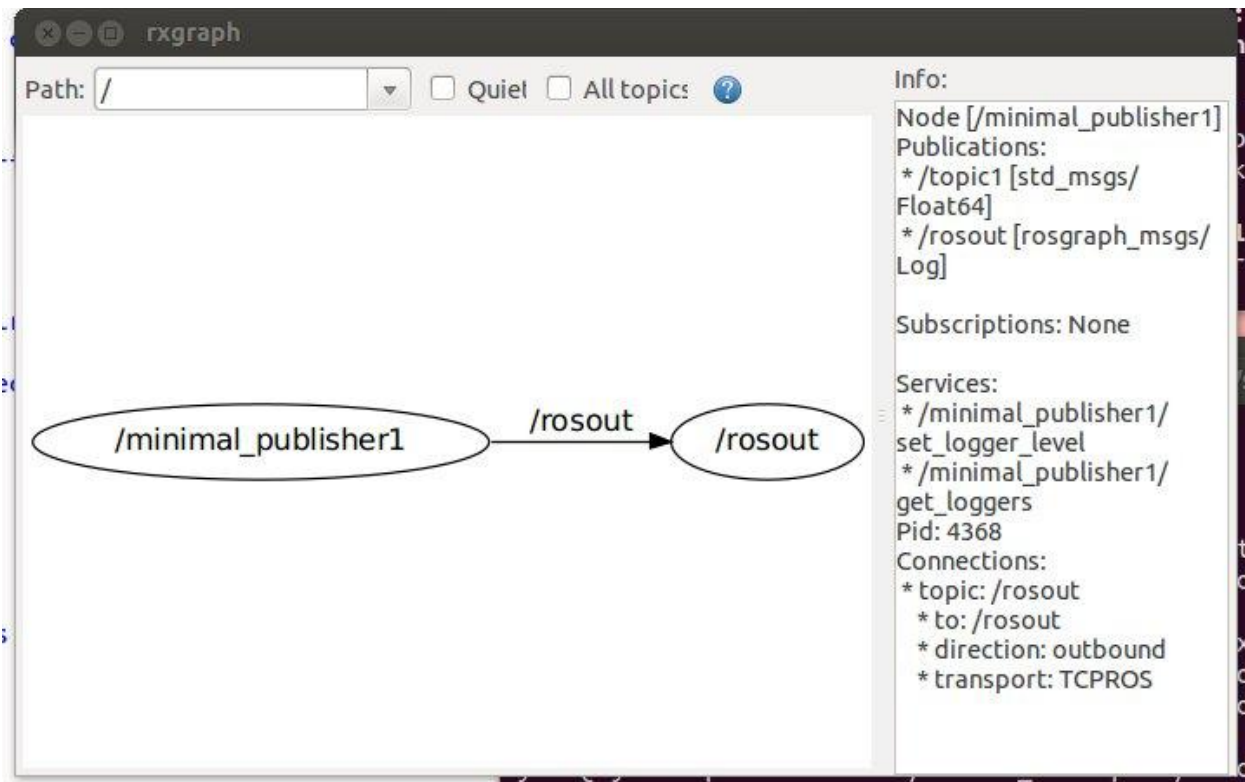
This shows the connectivity of the running nodes, which is quite sparse in the present case with only 2 nodes running. Although minimal_publisher1 does not take advantage of the capability of displaying output to its terminal, the link is nonetheless available through the topic "rosout", which would get processed by the display node "rosout." Using rosout can be helpful, since one's code does not get slowed down by output (e.g. cout) operations. Rather, messages get sent rapidly by publishing the output to the rosout topic, and a separate node (rosout) is responsible for user display. This can be important, e.g., in time-critical code where some monitoring is desired, but not at the expense of slowing down the time-critical node.

3.5) Scheduling the publisher: We have seen that our example publisher is abusive of both CPU capacity and communications bandwidth. In fact, it would be unusual for a node within a robotic system to require updates at 30kHz. A more reasonable update rate for even time-critical, low-level nodes is 1kHz. In the present example, we will slow our publisher down to 1Hz using a ROS timer.

The modify source code for minimal_publisher.cpp is shown below:

```cpp
#include <ros/ros.h>
#include <std_msgs/Float64.h>
int main(int argc, char **argv)
{
ros::init(argc,argv,"minimal_publisher1"); // name of this node will be "minimal_publisher1"
ros::NodeHandle n; // two lines to create a publisher object that can talk to ROS
ros::Publisher my_publisher_object = n.advertise<std_msgs::Float64>("topic1",1);
//"topic1" is the name of the topic to which we will publish
// the "1" argument says to use a buffer size of 1; could make larger, if expect network backups
std_msgs::Float64 input_float; //create a variable of type "Float64", as defined in: /opt/ros/fuerte/
share/std_msgs
// any message published on a ROS topic must have a pre-defined format, so subscribers know how to
// interpret the serialized data transmission
ros::Rate naptime(1); //set the sleep timer for 1Hz repetition rate
input_float.data=0.0;
while(ros::ok()) // do work here in infinite loop (desired for this example), but terminate if
detect ROS has faulted
{
// this loop has not sleep timer, and thus it will consume excessive CPU time
// expect one core to be 100% dedicated to this small task
input_float.data = input_float.data+0.001; //increment by 0.001 each iteration
my_publisher_object.publish(input_float); // publish the value--of type Float64-- to the topic
"topic1"
naptime.sleep(); //this will cause the loop to sleep for the balance of the desired period to
achieve specified frequency
}
}
```

There are only two new lines in the above program:

```cpp
ros::Rate naptime(1); //set the sleep timer for 1Hz repetition rate
and:   naptime.sleep();
```

The ROS class "Rate" is invoked to create a Rate object that I named "naptime". In doing so, naptime is initialized with the value "1", which is a specification of the desired frequency (1Hz). After creating this object, it is used within the while-loop, invoking the member function "sleep()". This causes the node to suspend (thus ceasing to consume CPU time) until the balance of the desired period (1 second) has expired.

After re-compiling (with "rosmake") and running (with rosrun) our modified code, rostopic hz topic1 shows the display below, which indicates that the topic "topic1" is being updated at 1Hz with excellent precision and very low jitter. Further, and inspection of the system monitor shows that there is negligible CPU time being consumed by our modified publisher node.

Entering: rostopic echo topic1 results in the output below:

Each message sent by the publisher is displayed by rostopic echo, as evidenced by the increments of 0.001 between messages. This display is updated once per second, since that is the rate new data is now published.

3.6) Writing a minimal ROS subscriber: The complement to the publisher is a subscriber (a listener node). We will create this node in the same package, "minimal_nodes." The source code will go in the subdirectory "src."

Open an editor and create the file "minimal_subscriber.cpp". Enter the following code:

```cpp
#include<ros/ros.h>
#include<std_msgs/Float64.h>
void myCallback(const std_msgs::Float64& message_holder)
{
// the real work is done in this callback function
// it wakes up every time a new message is published on "topic1"
// Since this function is prompted by a message event, it does not consume CPU time polling for new
data
// the ROS_INFO() function is like a printf() function, except
// it publishes its output to the default rosout topic, which prevents
// slowing down this function for display calls, and it makes the
// data available for viewing and logging purposes
ROS_INFO("received value is: %f",message_holder.data);
//really could do something interesting here with the received data..but all we do is print it
}
int main(int argc, char **argv)
{
ros::init(argc,argv,"minimal_subscriber"); //name this node
// when this compiled code is run, ROS will recognize it as a node called "minimal_subscriber"
ros::NodeHandle n;
//create a Subscriber object and have it subscribe to the topic "topic1"
// the function "myCallback" will wake up whenever a new message is published to topic1
// the real work is done inside the callback function
ros::Subscriber my_subscriber_object= n.subscribe("topic1",1,myCallback);
ros::spin(); //this is essentially a "while(1)" statement, except it
// forces refreshing wakeups upon new data arrival
// main program essentially hangs here, but it must stay alive to keep the callback function alive
return 0; // should never get here, unless roscore dies
}
```

The ROS subscriber is more complex than the publisher, since it requires a callback function. A callback function, "myCallback()" is defined that has an argument of a reference pointer to an object of type std_msgs::Float64. This is the message type associated with the topic "topic1", as published by our minimal publisher. Within the callback function, the only action taken is to display the received data. This is done using ROS_OUT() instead of cout or printf. Using ROS_OUT() uses message publishing, which avoids slowing down time-critical code for display driving. Also, using ROS_OUT() makes the data available for logging or monitoring. The effect, though, as viewed from the terminal from which this node is run is indistinguishable from using cout or printf.

In the main program, the use of ros::init is as before. Within the source code, the node defines its own name (in this case "minimal_subscriber"). This is the name ROS will use to identify this node when it is executed.

The use of ros::Subscriber is similar to the use of ros::Publisher earlier. An object is substantiated of type "Subscriber" use a class that exists within the ROS distribution. There are three arguments used in instantiating the subscriber object. The first argument is the topic name. "topic1" is chosen here, which is the topic to which our minimal publisher publishes. The second argument is the queue size. If the callback function has trouble keeping up with published data, the data may be queued up. In the present case, the queue size is set to one. If the callback function cannot keep up with publications, messages will be lost. (Recall that rostopic echo topic1 could not keep up with the 30kHz rate of the original minimal publisher. Values displayed skipped many intermediate messages). For control purposes, typically only the most recent sensor value published is of interest. If a sensor publishes faster than the callback

function can respond, there is no hard done in dropping messages, as only the most recent message would be needed. In this (and many cases) a queue size of 1 message is all that is needed.

The third argument for instantiating the Subscriber object is the name of the callback function that is to receive data from topic1. This argument has been set to "myCallback", which is the name of our callback function, described earlier.

The ros::spin(); command is non-obvious but essential. The callback routine is actually called from within ros::spin(). This function is blocking (never returns) unless the node is shut down. It's job is to keep the callback function alive.

3.7) Compiling and running the minimal subscriber: For our new node to get compiled, we must include reference to it in CMakeLists.txt. This requires adding a single line, very similar to what we did to enable compiling the minimal publisher. A snippet of CMakeLists.txt is shown below, which includes the single new line of rosbuild_add_executable(). The arguments are the desired executable name (chosen to be identical to the source-code name and, in this case, also identical to the node name) "minimal_subscriber." The relative path to the source code is the second argument: src/minimal_subscriber.cpp
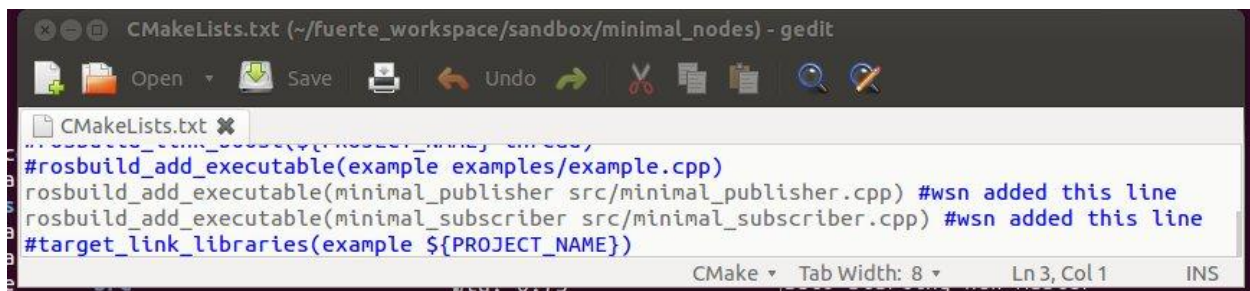
After updating CMakeLists.txt, the code is newly compiled with the command:

rosmake minimal_nodes

The code example should compile without error, after which a new executable, "minimal_subscriber" will appear in the "bin" subdirectory.

We now have two nodes to run. In one terminal, enter: rosrun minimal_nodes minimal_publisher and in a second terminal enter: rosrun minimal_nodes minimal_subscriber. It does not matter which is launched first. An example of the display in the terminal of the minimal_subscriber node is shown below:

This display updated once per second, since the publisher published to topic1 at 1Hz. The messages received increment by 0.001, as programmed in the publisher code.

In another terminal, entering: rosnode list results in the output below.

This shows that we now have 3 nodes running: the default rosout, our minimal publisher (which we named node "minimal_publisher1") and our minimal subscriber (who's node name is identical to the executable code name).

In an available terminal, typing rxgraph produces a graphical display of the running system, shown below:
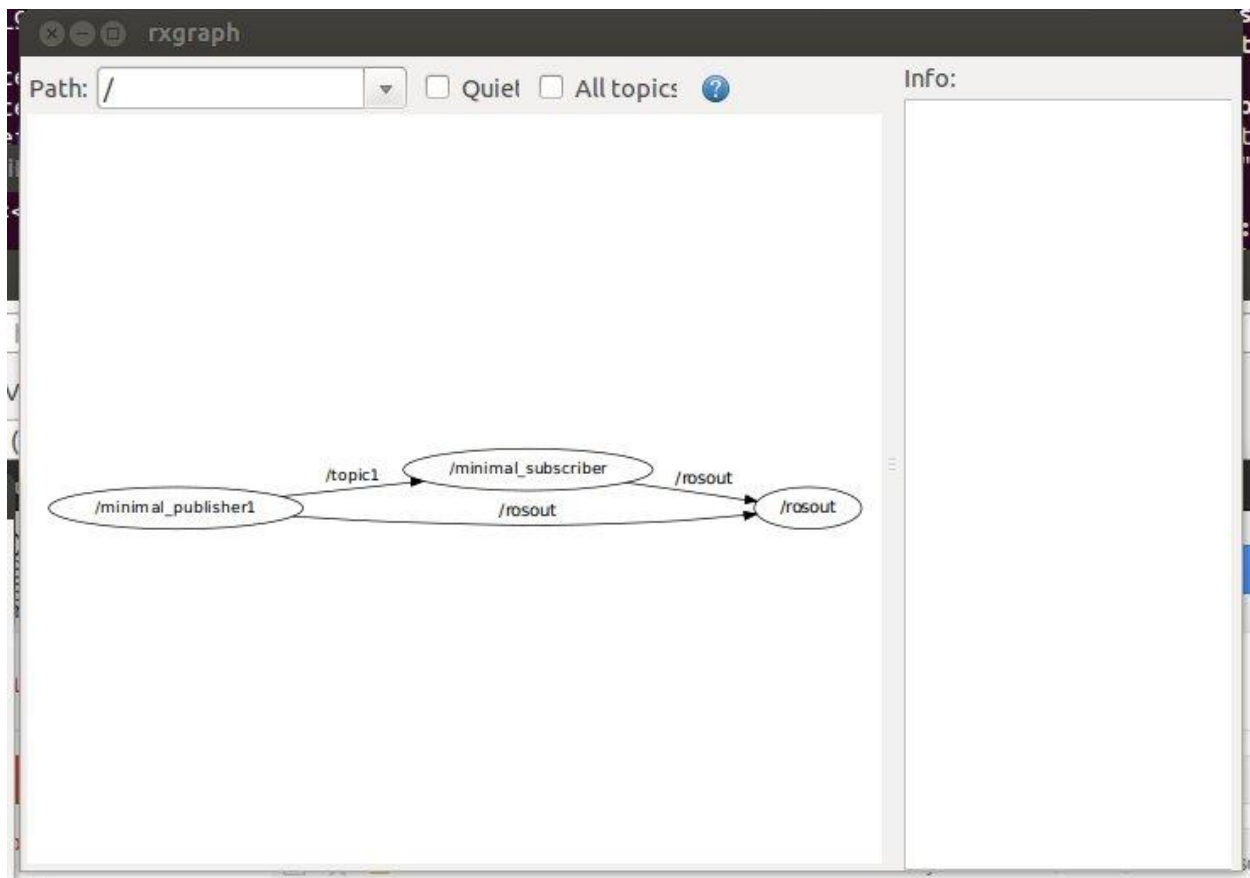
The graphical display shows 3 nodes. Our minimal publisher node is shown publishing to topic "topic1" and our minimal subscriber is shown subscribing to this topic. Both of our nodes have access to publish to the "rosout" topic, which is serviced by the node "rosout" subscribing to this topic.

3.7) Minimal subscriber and publisher node summary: We have seen the basics for how to create our own publisher and subscriber nodes. A single node can subscribe to multiple topics by replicating the corresponding lines of code to create additional subscriber objects and callback functions. Similarly, a node can publish to multiple topics by instantiating multiple Publisher objects. A node can also be both a subscriber and a publisher.

In a ROS-controlled robot system, custom nodes must be designed that are device drivers that can read and publish sensor information and one or more nodes that subscribe to actuator (or controller setpoint) commands and impose these on actuators. Fortunately, there is already a large body of existing ROS drivers for common sensors, including LIDARs, cameras, the Kinect sensor, inertial measurement units, encoders, etc. These may be imported as packages and used as-is in your system (though perhaps requiring some tweaking to reference your com ports, etc). There are also some packages for driving some servos (i.e. hobby-servo style). However, the user may need to design their own subscriber node to interface to custom actuators. Further, hard real-time, high-speed servo loops may require a non-ROS, dedicated controller (although this might be as simple as an Arduino microcontroller). The user then assumes the responsibility for designing the hard-real-time controller and writing a ROS-compatible subscriber interface to run on the control computer.

4) Some more ROS tools: roslaunch, rxconsole and rosbag

Having introduced minimal ROS talkers (publishers) and listeners (subscribers), one can already begin to appreciate the value of some additional ROS tools. Three of these tools are introduced here.

4.1) roslaunch: In our minimal example, we launched two nodes: one publisher and one subscriber. To do so, we opened two separate terminals and typed in two "rosrun" commands. Since a complex system may have hundreds of nodes running, we need a more convenient way to bring up a system. This can be done using "launch" files and the command "roslaunch." (see http://ros.org/wiki/roslaunch for more details and additional capabilities, such as setting parameters).

A launch file has the suffix ".launch". It is conventionally named the same as the package name (though this is not required). A launch file can also invoke other launch files to bring up multiple packages. However, we will start with a minimal launch file.   We may name our launch file "minimal_nodes.launch" and enter the following lines:

```
<launch>
  <node name= "minimal_publisher"   pkg= "minimal_nodes"   type= "minimal_publisher" />
  <node name= "minimal_subscriber"   pkg= "minimal_nodes"   type= "minimal_publisher" />
</launch>
```

Assuming "roscore" is still running (and if it is not, open a terminal and start it), we can execute the launch file by typing: **roslaunch   minimal_nodes   minimal_nodes.launch**

The terminal in which we executed "roslaunch" reports the nodes that it launched (our minimal publisher and our minimal subscriber). Invoking **rosnode list** in another terminal confirms that both nodes are running. Also, **rostopic list** shows that the "topic1" is alive, and the command **rostopic info topic1** shows that our minimal_publisher node is publishing to this topic and our minimal_subscriber node has subscribed to this topic. Clearly, this will be useful when we have many nodes to launch.

One side effect, though, is that we no longer see the output from our subscriber, which formerly appeared in the terminal from which the subscriber was launched. However, since we used "ROS_INFO()" instead of "printf" or "cout", we can still observe this output using the "rxconsole" tool.

4.2) rxconsole: With our two nodes running, type the command: **rxconsole**. An example of the result of this is shown in the following figure. In this instance, rxconsole shows values output by the minimal subscriber from the time rxconsole was started and until rxconsole was paused (using the rxconsole "pause" button). The lines displayed show that the messages are merely informational—not warnings or errors. The console also shows that the node responsible for posting the information is our minimal subscriber. Rxconsole also notes the time-stamp at which the message was sent.
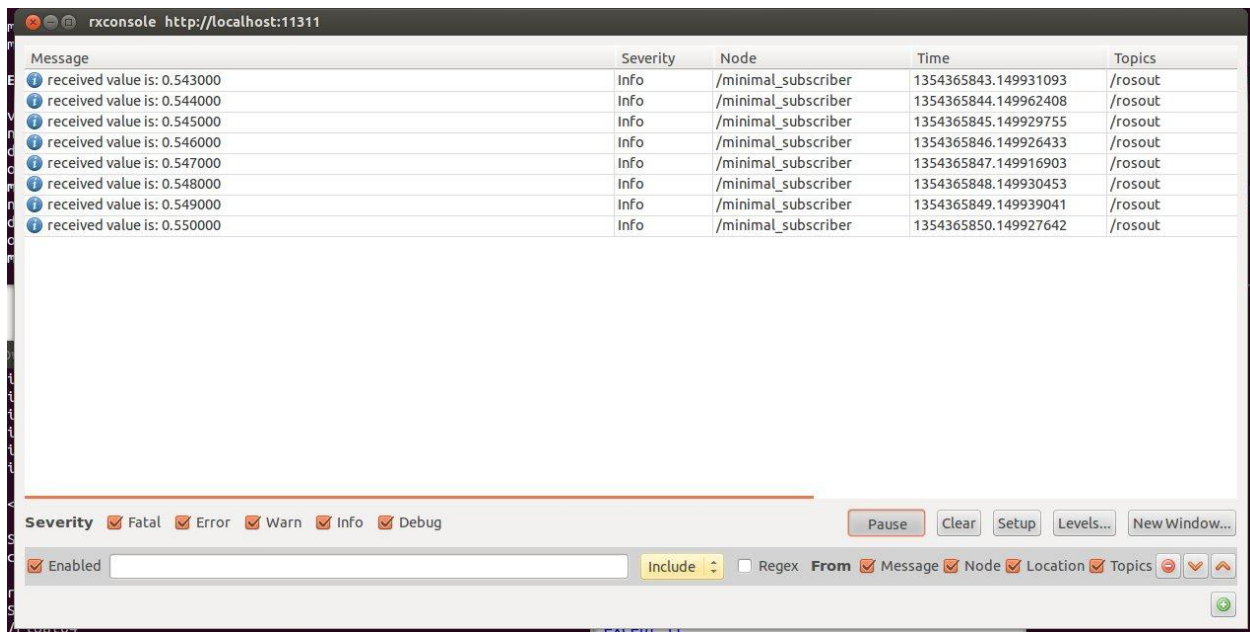
Rxconsole displaying ROS_INFO( ) messages from the minimal_subscriber node.

Multiple nodes using ROS_INFO() may be run simultaneously, and their messages may be viewed with rxconsole. Rxconsole will also not new events, such as starting and stopping new nodes.
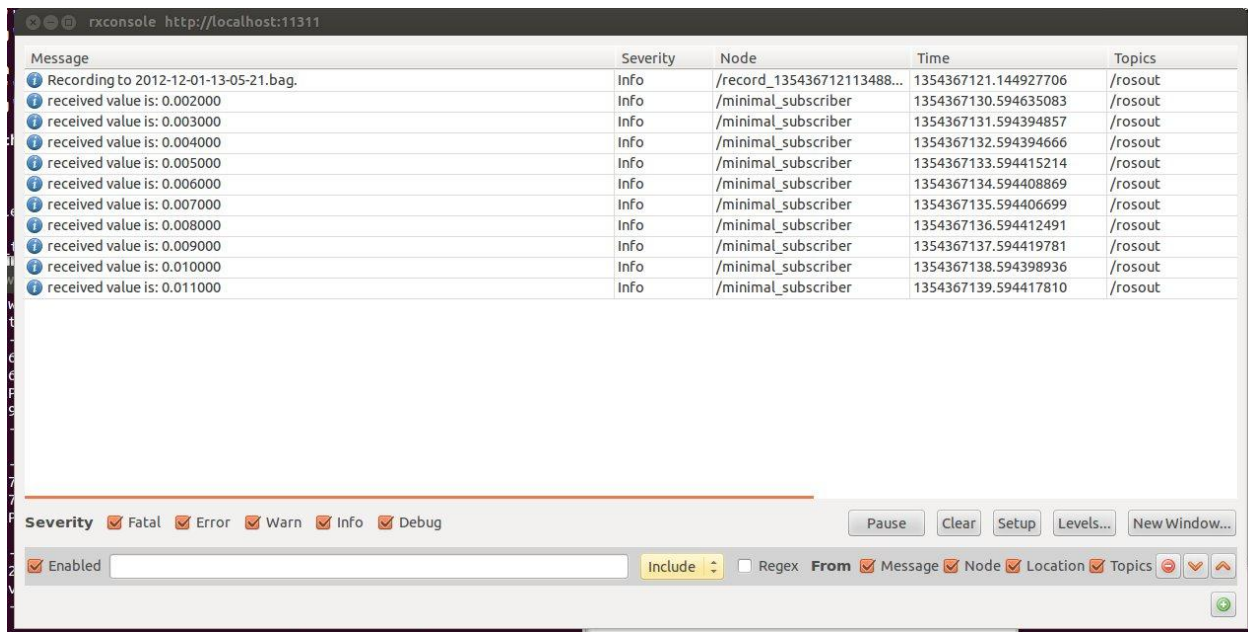
Another advantage of using ROS_INFO() instead of printf() is that the messages output can be logged and run in playback. A facility for doing this is "rosbag."

4.3) Recording and playing back data with "rosbag": The "rosbag" command is extremely useful for debugging complex systems. One can specify a list of topics to record while the system is running, and "rosbag" will subscribe to these topics and record the messages published, along with timestamps in a "bag" file. Rosbag can also be used to play back "bag" files, thus recreating the circumstances of the recorded system. (This playback occurs at the same clock rate at which the original data was published, thus emulating the real-time system).

When running "rosbag", the resulting log (bag) files will be saved in the same directory from which "rosbag" was launched. For our example, move to the "minmal_nodes" directory and create a new subdirectory "bagfiles." Although the order is not important, start without our nodes running (control-C in the launch window, if they are still running). Start up the rxconsole.

In a terminal, navigate to the "bagfile" directory and enter: **rosbag record topic1** . With this command, we have asked to record all messages published on topic1. The rxconsole displays a line showing that recording has begun and names the file to which the data is recorded (by default, labeled using time/date).

In another terminal, launch our two nodes using: **roslaunch   minimal_nodes   minimal_nodes.launch.** Rxconsole begins displaying data from topic1, as reported by our subscriber node using ROS_INFO(). In the screenshot of rxconsole shown below, the nodes were halted after 11 seconds (topic value of 0.011). The bagfile recording was subsequently halted with a control-C in the terminal from which it was launched.

Message	Severity	Node	Time	Topics
Recording to 2012-12-01-13-05-21.bag.	Info	/record_135436712113488...	1354367121.144927706	/rosout
received value is: 0.002000	Info	/minimal_subscriber	1354367130.594635083	/rosout
received value is: 0.003000	Info	/minimal_subscriber	1354367131.594394857	/rosout
received value is: 0.004000	Info	/minimal_subscriber	1354367132.594394666	/rosout
received value is: 0.005000	Info	/minimal_subscriber	1354367133.594415214	/rosout
received value is: 0.006000	Info	/minimal_subscriber	1354367134.594408869	/rosout
received value is: 0.007000	Info	/minimal_subscriber	1354367135.594406699	/rosout
received value is: 0.008000	Info	/minimal_subscriber	1354367136.594412491	/rosout
received value is: 0.009000	Info	/minimal_subscriber	1354367137.594419781	/rosout
received value is: 0.010000	Info	/minimal_subscriber	1354367138.594398936	/rosout
received value is: 0.011000	Info	/minimal_subscriber	1354367139.594417810	/rosout

Severity ☑ Fatal ☑ Error ☑ Warn ☑ Info ☑ Debug   Pause Clear Setup Levels... New Window...

☑ Enabled [   ] Include ↕ ☐ Regex **From** ☑ Message ☑ Node ☑ Location ☑ Topics

Rxconsole display during rosbag recording

Looking in the "bagfiles" directory (from which we launched rosbag), we see there is a new file, named according to the date/time of the recording, and with the suffix ".bag". We can play back the recording using rosbag as well. To do so, merely type: **rosbag play fname.bag** where "fname" is the file name of the recording. The rosbag terminal shows a playbag time incrementing, but there is otherwise no noticeable effect. Rxconsole indicates that the bagfile has been opened, but no other information is displayed. What is happening at this point is that "rosbag" is replaying the recorded data by publishing the recorded values (from topic1) to the topic "topic1", and it is doing so at the same rate as the data was recorded.

To see that this is taking place, do the following. Halt all nodes (including rosbag). Run rxconsole. In a terminal window, start up the subscriber, but not the publisher, using:
rosrun minimal_nodes minimal_subscriber
At this point, this terminal is suspended, because "topic1" is not yet active.

In the "bagfiles" directory, enter**: rosbag play fname.bag** , where "fname" is the name of the bag file that was just created. Rosbag now assumes the role formerly taken by "minimal_publisher", recreating the messages that were formerly published. The minimal_subscriber window reports the recorded data, updating once per second. Rxconsole also shows the data posted by the minimal subscriber. As can be seen from the screenshot below, the playback is identical to the original recording. Dynamically, these values are posted at the original 1Hz rate that minimal_publisher had published them. The rosbag player terminates when it gets to the end of the recorded data.

Rxconsole display from "rosbag play" of recorded file, with minimal_subscriber node running

Note that our subscriber is oblivious to what entity is publishing to topic1. Consequently, playback of previously recorded data is indistinguishable from receiving live data. This is very useful for development. For example, a robot may be teleoperated through an environment of interest while it publishes sensor data from cameras, LIDAR, etc. Using "rosbag", this data may be recorded verbatim. Subsequently, sensor processing may be peformed on the recorded data to test, e.g., machine vision algorithms. Once a sensory-interpretation node is shown to be effective on the recorded data, the same node may be tried out verbatim on the robot system. Note that no changes to the developed node are needed. In live experiments, this node would merely be receiving messages published by the real-time system instead of by "rosbag playback."

5) A Minimal Simulator and Controller Example:

Concluding this introduction, we consider a pair of nodes that both publish and subscribe. One of these nodes is a minimal simulator, and the other is a minimal controller. The minimal simulator simulates F=ma by integrating acceleration to update velocities. The input force is published to the topic "force_cmd" by some entity (eventually, the controller). The resulting system state (the velocity) is published to the topic "velocity" by the minimal simulator.

The simulator code follows. Note that the callback routine checks for new data on the topic "force_cmd." The callback routine posts data, as it is received, in the internal global variable "g_force," so that the main program has access to it.

Note the use of a new ROS function: ros::spinOnce(). This function, executed within the 100Hz loop of the simulator, checks for new force commands each simulation cycle. If a new command is received, it is stored in g_force by the callback function. However, since this simulator is designed to run 10x faster than the example controller counterpart, 9 out of 10 times there will be no new messages on this topic.

23

The callback function will not block, but it will allow the main loop to repeat its iterations at a higher rate than the controller.

File: /home/wyatt/ros/minimal_nodes_test/src/minimal_simulator.cpp Page 1 of 1

```cpp
// minimal_simulator node:
// wsn example node that both subscribes and publishes
// does trivial system simulation, F=ma, to update velocity given F specified on topic "force_cmd"
// publishes velocity on topic "velocity"
#include<ros/ros.h>
#include<std_msgs/Float64.h>
std_msgs::Float64 g_velocity;
std_msgs::Float64 g_force;
void myCallback(const std_msgs::Float64& message_holder)
{
// checks for messages on topic "force_cmd"
ROS_INFO("received force value is: %f",message_holder.data);
g_force.data = message_holder.data; // post the received data in a global var for access by
// main prog.
}
int main(int argc, char **argv)
{
ros::init(argc,argv,"minimal_simulator"); //name this node
// when this compiled code is run, ROS will recognize it as a node called "minimal_simulator"
ros::NodeHandle ns; //subscriber node handle
ros::NodeHandle np; // publisher node handle
//create a Subscriber object and have it subscribe to the topic "force_cmd"
ros::Subscriber my_subscriber_object= ns.subscribe("force_cmd",1,myCallback);
//simulate accelerations and publish the resulting velocity;
ros::Publisher my_publisher_object = np.advertise<std_msgs::Float64>("velocity",1);
double mass=1.0;
double dt = 0.01; //10ms integration time step
double sample_rate = 1.0/dt; // compute the corresponding update frequency
ros::Rate naptime(sample_rate);
g_velocity.data=0.0; //initialize velocity to zero
g_force.data=0.0; // initialize force to 0; will get updated by callback
while(ros::ok())
{
g_velocity.data = g_velocity.data + (g_force.data/mass)*dt; // Euler integration of
//acceleration
my_publisher_object.publish(g_velocity); // publish the system state (trivial--1-D)
ROS_INFO("velocity = %f",g_velocity.data);
ros::spinOnce(); //allow data update from callback
naptime.sleep(); // wait for remainder of specified period; this loop rate is faster than
// the update rate of the 10Hz controller that specifies force_cmd
// however, simulator must advance each 10ms regardless
}
return 0; // should never get here, unless roscore dies
}
```

The following code is a compatible minimal controller node. It subscribes to 2 topics (velocity and vel_cmd) and it publishes to the topic "force_cmd". Each control cycle (set to 10Hz), the controller checks for the latest system state (velocity), checks for any updates to the commanded velocity, and it computes a proportional-error feedback to derive (and publish) a force command. This simple controller attempts to drive the simulated system to the user-commanded velocity setpoint. Again, the ros::spinOnce() function is used to prevent blocking in the timed, main loop. Callback functions put received message data in the global variables g_velocity and g_vel_cmd.

File: /home/wyatt/ros/minimal_nodes…est/src/minimal_controller.cpp

```cpp
// minimal_controller node:
// wsn example node that both subscribes and publishes--counterpart to minimal_simulator
// subscribes to "velocity" and publishes "force_cmd"
// subscribes to "vel_cmd"
#include<ros/ros.h>
#include<std_msgs/Float64.h>
//global variables for callback functions to populate for use in main program
std_msgs::Float64 g_velocity;
std_msgs::Float64 g_vel_cmd;
std_msgs::Float64 g_force; // this one does not need to be global...
void myCallbackVelocity(const std_msgs::Float64& message_holder)
{
// check for data on topic "velocity"
ROS_INFO("received velocity value is: %f",message_holder.data);
g_velocity.data = message_holder.data; // post the received data in a global var for access by
//main prog.
}
void myCallbackVelCmd(const std_msgs::Float64& message_holder)
{
// check for data on topic "vel_cmd"
ROS_INFO("received velocity command value is: %f",message_holder.data);
g_vel_cmd.data = message_holder.data; // post the received data in a global var for access by
//main prog.
}
int main(int argc, char **argv)
{
ros::init(argc,argv,"minimal_controller"); //name this node
// when this compiled code is run, ROS will recognize it as a node called "minimal_controller"
ros::NodeHandle ns1; //subscriber1 node handle
ros::NodeHandle ns2; //subscriber2 node handle
ros::NodeHandle np; // publisher node handle
//create 2 subscribers: one for state sensing (velocity) and one for velocity commands
ros::Subscriber my_subscriber_object1= ns1.subscribe("velocity",1,myCallbackVelocity);
ros::Subscriber my_subscriber_object2= ns2.subscribe("vel_cmd",1,myCallbackVelCmd);
//publish a force command computed by this controller;
ros::Publisher my_publisher_object = np.advertise<std_msgs::Float64>("force_cmd",1);
double Kv=1.0; // velocity feedback gain
double dt_controller = 0.1; //specify 10Hz controller sample rate (pretty slow, but
//illustrative)
double sample_rate = 1.0/dt_controller; // compute the corresponding update frequency
ros::Rate naptime(sample_rate); // use to regulate loop rate
g_velocity.data=0.0; //initialize velocity to zero
g_force.data=0.0; // initialize force to 0; will get updated by callback
g_vel_cmd.data=0.0; // init velocity command to zero
double vel_err=0.0; // velocity error
// enter the main loop: get velocity state and velocity commands
// compute command force to get system velocity to match velocity command
// publish this force for use by the complementary simulator
while(ros::ok())
{
```

```
vel_err = g_vel_cmd.data - g_velocity.data; // compute error btwn desired and actual
//velocities
g_force.data = Kv*vel_err; //proportional-only velocity-error feedback defines commanded
//force
my_publisher_object.publish(g_force); // publish the control effort computed by this
//controller
ROS_INFO("force command = %f",g_force.data);
ros::spinOnce(); //allow data update from callback;
naptime.sleep(); // wait for remainder of specified period;
}
return 0; // should never get here, unless roscore dies
}
```

Once the two nodes are compiled with "rosmake" (which requires editing CMakeLists.txt to add these executables to the package), they can be run (with "rosrun") from separate terminal windows (assuming "roscore" is running). Running "rxconsole" reveals that the force command is updated once for every 10 updates of velocity (as expected for the simulator at 100Hz and the controller at 10Hz).

The velocity command may be input from another terminal using a command line, e.g.:

rostopic pub –r 10 vel_cmd std_msgs/Float64 1.0 publishes the value "1.0" to the topic "vel_cmd" repeatedly at a rate of 10Hz. Watching the output on rxconsole, the velocity can be seen to converge exponentially on the desired value of vel_cmd.

The result can be visualized graphically with the ROS tool **rxplot**. To do so, use command-line arguments for the values to be plotted, i.e.: **rxplot vel_cmd/data,velocity/data**,**force_cmd/data** will plot the velocity command, the actual velocity and the force commands on the same plot vs time.

For the minimal simulator and minimal controller, the velocity command was initially set to 1.0 via rostopic pub. Subsequently, the command was set to 0.0. A snippet of the resulting rxplot is shown below. The control effort (in green) reacts with a braking force, which causes the system velocity (in blue) to decrease. As the system velocity gets closer to the goal of 0.0, the control effort decreases. Ultimately, the system velocity converges on the goal value and the required control effort decreases to zero.
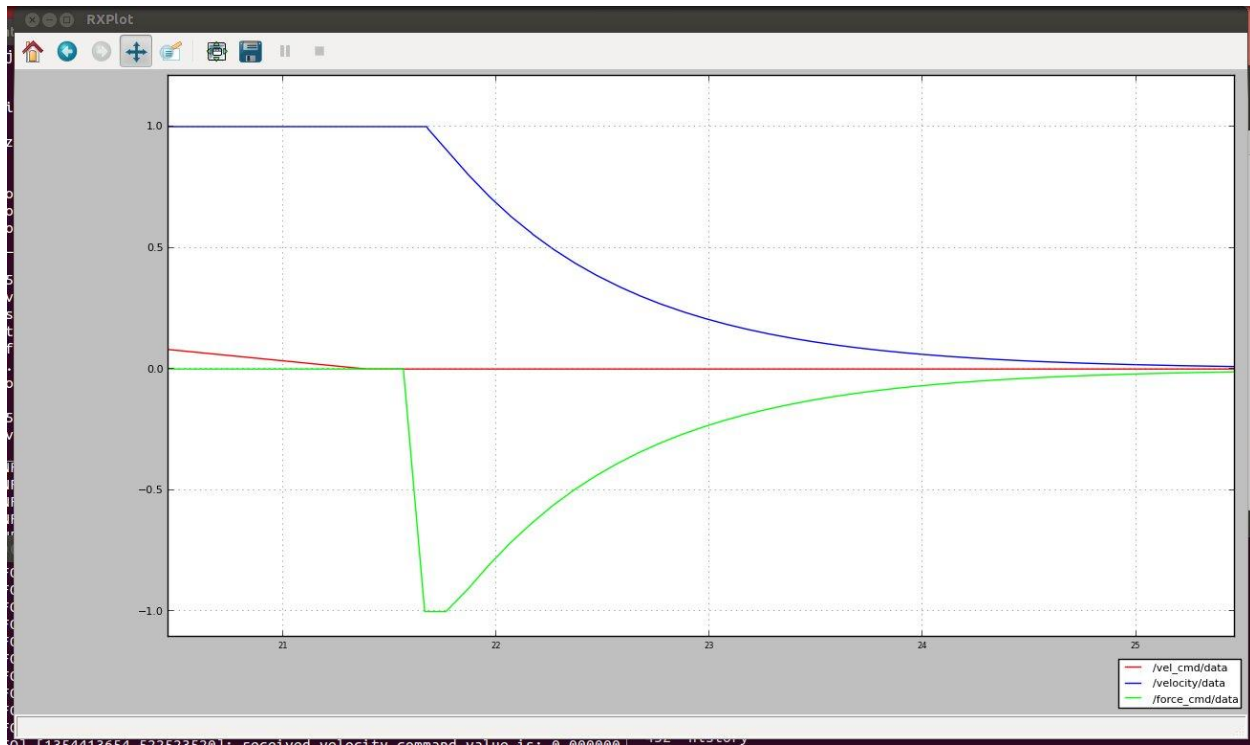
6) What Else?

This introduction has only scratched the surface, although it already provides enough information to interact usefully with simulators and real robots and sensors.

Some topics to explore further include: the *parameter server* (for setting values in a system at run time, instead of embedded in compiled code), *services* (like remote procedure calls, perform a function on demand and return a result), and coordinate *transforms* (tf), which are ubiquitous in robotics. Further, to understand how a robot is specified to the system for simulation, it is useful to understand *URDF* (universal robot descriptor file) syntax. The tool "*rviz*" is valuable for visualizing data (whether real time or from playback of bagged data).

Beyond mastering ROS tools and tricks, there are many valuable packages contributed as open-source code. Some of these are: SLAM, localization, path planning, trajectory generation, Kinect point-cloud processing, natural-language interfaces, and many more. Learning how to utilize 3rd-party packages requires acquiring the packages (typically using git, hg or svn), incorporating the necessary header files in your code, and updating dependencies in your manifest file.

On-line ROS tutorials and the ROS wiki are valuable resources for learning more. This introduction hopefully will help to make these resources more understandable.