Interfacing to the DARPA Robotics Challenge simulator:
Wyatt Newman
November, 2012

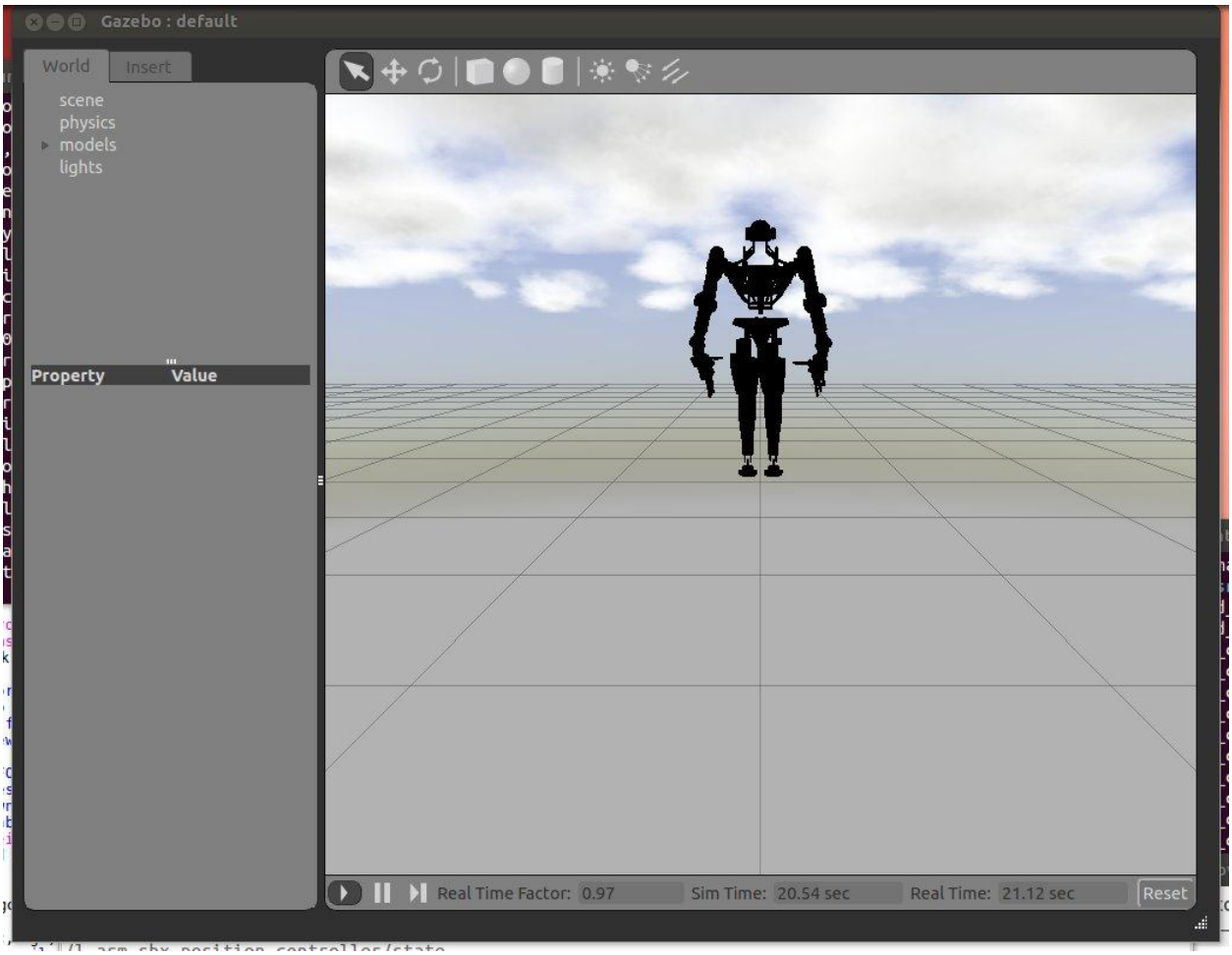
These notes provide a brief introduction to interfacing to the DRC gazebo simulator via ROS.

A simulator has been written in "gazebo" for the Boston Dynamics "Atlas" robot. This simulator is the basis of the first DARPA challenge: the virtual robotics challenge. For this simulator, ROS publisher and subscriber nodes have already been written. By launching the simulator in "gazebo" (which automatically starts up roscore as well), one can examine all of the topics that are available. Sensor nodes publish sensor information (currently including joint angles and angular velocities and color camera data, with more sensor publishers anticipated). Corresponding joint-control topics are subscribed to by joint controllers. It is up to the user to publish to these pre-defined topics in order to move the simulated robot.

It is the intent that the Boston Dynamics "Atlas" robot will have the same ROS interfaces, and thus the same sensor and control topics will be available. Ideally, code written for the simulator will be immediately applicable to the Atlas robot, merely by launching Atlas hardware-interface ROS nodes instead of gazebo simulator ROS nodes for I/O.

The simulator was written by the Open Robotics Software Foundation (ORSF) with its first release in October, 2012. (see http://gazebosim.org/ and http://gazebosim.org/wiki/InstallDRC). This section will focus on rudimentary examples of how to interface user-written ROS publisher and subscriber nodes with the ORSF DRC simulator.

1) Running the Gazebo Simulator of Atlas: The DRC (DARPA Robotics Challenge) simulator may be installed by following the instructions at the preceding links. After installation, the simulator can be launched by entering: **roslaunch drc_robot_utils drc_robot.launch** A "gazebo" graphical window will appear, as shown in the example below. Note that "gazebo" will only run with a limited set of software and hardware configurations. Notably, the host computer must have a compatible graphics card and drivers. On my laptop, the DRC simulator crashes frequently.



Gazebo display of DRC simulator

With Gazebo running, **rostopic list** shows that there are nearly 150 active topics. Most of these correspond to publications of joint states or to controller nodes that have subscribed to topics for joint motion commands. We can interpret topics with the help of ROS commands.

Entering **rostopic hz /l_arm_ely_position_controller/state** in a terminal window shows that the state of the left-arm elbow is published at a rate of 100 Hz.

Running **rostopic info /l_arm_ely_position_controller/state** shows that this topic is published by the node "/gazebo" and that there are (at this point) no subscribers.

Entering: **rostopic type /l_arm_ely_position_controller/state** indicates that the message type published on this topic is defined in "pr2_controllers_msgs/JointControllerState."

rostopic bw /l_arm_ely_position_controller/state shows that this topic consumes 9.7KB/s of communications bandwidth.

rostopic echo /l_arm_ely_position_controller/state prints out data available from this topic, which includes a message ID, a time stamp, a joint angle, a joint velocity, PID values used by the joint controller, the current setpoint used by the controller, the joint effort exerted by the PID controller, and the sample period of the controller (1msec).

The counterpart topic, **/l_arm_ely_position_controller/command**, is subscribed to by gazebo, but there are no active publishers (at this point). Rostopic also shows that the message type for this command is std_msgs/Float64 (which we have already seen in our minimal example). By writing a code for a node that publishes to this topic (with this message type) one can compose joint trajectories to be commanded to the gazebo joint controller.

Topics are available to sense the state and control angles of all joints of the robot, all joints of both hands of the robot, and components of the robot's head. For actual (physical) Atlas robot, the intention is that there will be corresponding nodes that subscribe to these same named topics, and that code written for the gazebo simulator may be used verbatim to command the Atlas robot joints and to receive feedback of robot joint states from the robot.

With no additional (user) nodes running, the gazebo simulator shows that it is running the simulation in real time (i.e., time step of the Open Dynamics Engine physics simulation is equal to actual clock time increments). Examining the system monitor shows that one of the CPU cores is fully saturated running gazebo.

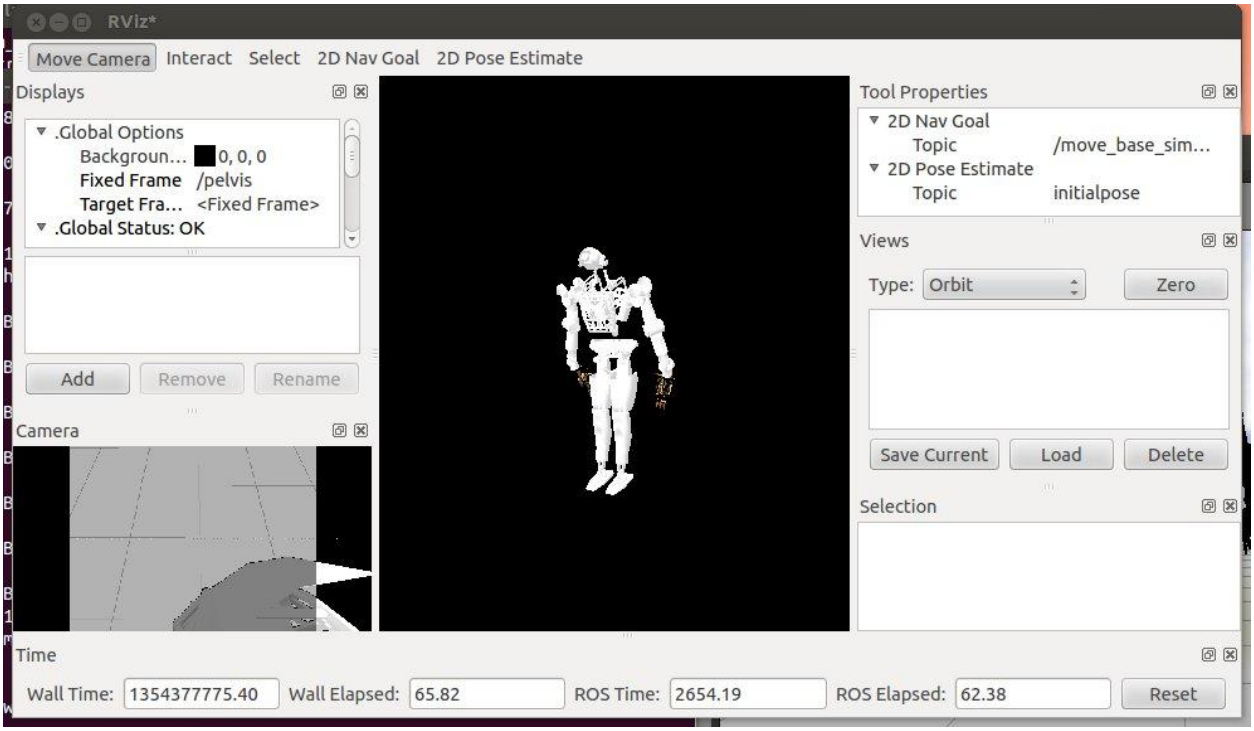
In its default condition, the simulator has the robot's pelvis "pinned." The robot does not have a balance controller. Its joints are controlled to the start-up angles using relatively soft PD controllers. (The integral error gain is set to 0 by default). The pelvis can be unpinned upon command, but it is useful to start working with joint controls in the initial pinned state, avoiding the need for considering environment interaction forces and balance and locomotion control.

One of the topics published by Gazebo is "/right_eye/image_raw" which is synthetic camera data from the robot's right eye. Messages on topic "/right_eye/image_raw" are of type "sensor_msgs/Image." Rostopic shows that this is updated at 13.7 Hz and that it uses 3.2MB/s of bandwidth. Options exist for setting various parameters, including update rate, field of view and compressing settings. One may construct a node to perform image processing on such images, or run "rosbag record" to save the synthetic images for offline development of image-processing code.

Another available topic is "scan", to which gazebo publishes messages of type "sensor_msgs/LaserScan". This topic contains synthetic data from a simulated Hokuyo LIDAR in the robot's sensor head. As model objects are added to the environment, the camera and scan images become more interesting.

As with the joint-state publishers and the joint-command subscribers, it is the intent that the simulated sensors be interchangeable with the real sensors. Data from the physical robot sensor head should be published on topics of the same names and same message formats as the synthetic data published by the gazebo DRC simulator. User nodes that process the sensor data should be verbatim compatible with the real sensor data, requiring no user changes to switch between the simulator and the physical robot.

2) Visualizing data with rviz: Another useful ROS tool is "rviz", which offers a variety of visualization options. In an available terminal, type: **rosrun rviz rviz** This executes a node called "rviz" from a package of the same name. Rviz has options to be set by the user (what data to display), and these options can be saved in a configuration file and re-used the next time rviz is launched. The screenshot below shows rviz running with the configuration settings described here: http://gazebosim.org/wiki/Tutorials/drcsim/visualizing_logging_data.



rviz screenshot of DRC simulator

This view shows the same model of the DRC robot as displayed in the "gazebo" window, but it can be viewed from alternative viewpoints more easily. Further, the rviz can display display selected sensor data. In the example screenshot, one B/W camera view and the LIDAR scan are displayed. These are updated dynamically as the robot moves.

3) Moving joints manually with* rostopic pub*: With the gazebo DRC simulator running, one can move joints by publishing commands to topics. As an example, in an available terminal, enter:

rostopic pub –r 10 /r_arm_shx_position_controller/command std_msgs/Float64 0.0

This command publishes the value "0.0" as a message of type "Float64" on the topic corresponding to right-arm, shoulder x-rotation command. The option –r 10 causes the command to be refreshed at 10Hz. Upon entering this command, the gazebo display will show the robot's right arm elevate to the side, due to a rotation of the right shoulder. Each of the robot's joints can be tested in this manner to identify the meaning, home poses and defined positive motions of each joint.

4) Moving joints under program control:

The following code constitutes a node capable of both subscribing to a gazebo joint state topic and publishing joint-angle commands to gazebo command topics. In this example, the right leg is moved forward and back at the hip sinusoidally via commands published to the respective command topic. The actual angle of the right leg is read from the respective ROS joint state topic. This value is negated and used as a command to the left leg. As a result, the legs are coordinated to swing in opposite directions (although there are errors due to the low-gain controllers).

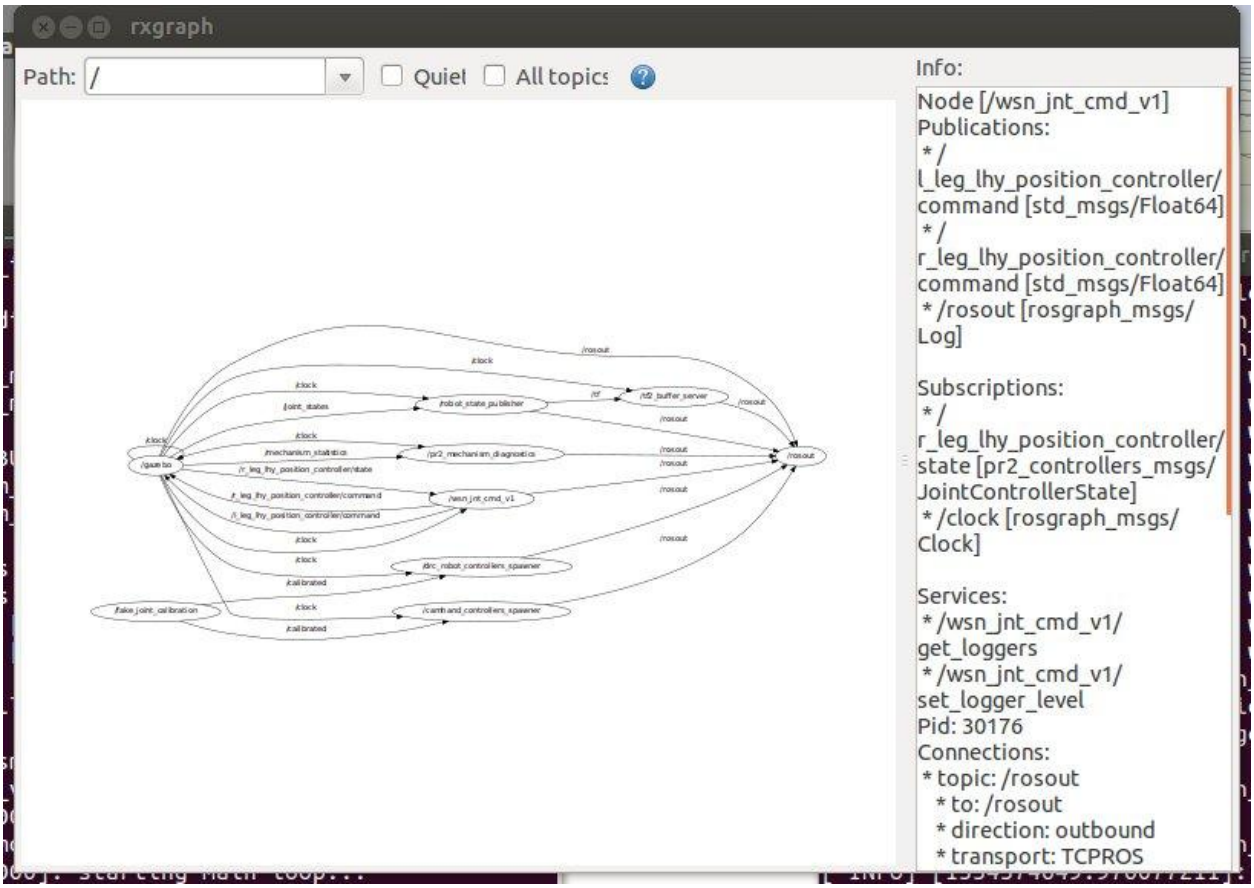
```
// wsn example of both command publishing and state subscribing interfacing to DRC simulator
//test of interfacing to DRC simulator through cpp
#include<ros/ros.h>
#include<iostream>
#include<std_msgs/Float64.h>
#include<geometry_msgs/Twist.h> //data type for velocities
#include<pr2_controllers_msgs/JointControllerState.h>
#include<math.h>
using namespace std;
//global variables, to pass info from callback to main:
std_msgs::Float64 sensed_r_leg_lhy;
std_msgs::Float64 sensed_l_leg_lhy;
//read joint state from Gazebo drc_sim
//specifically, right-leg hip flexor
void stateCallback_r_leg_lhy(const pr2_controllers_msgs::JointControllerState::ConstPtr& jntState)
{
//ROS_INFO("received a state packet, ang = %lf",jntState->process_value);
sensed_r_leg_lhy.data=jntState->process_value; //put this in global var
}
int main(int argc,char **argv)
{
ros::init(argc,argv,"wsn_jnt_cmd_v1");//name of this node
//define node handles for various command publications and state subscriptions:
//nhp means node-handle for publisher
ros::NodeHandle nhp_r_leg_lhy; // node handle for command publisher, right leg hip flexor
(fwd/back motion)
ros::NodeHandle nhp_l_leg_lhy; // node handle for command publisher, left leg hip flexor
//nhs means node-handler for subscriber
ros::NodeHandle nhs_r_leg_lhy; //node handle for state subscriber, right leg hip flexor
//set up a subscriber to published joint state, and a pair of joint-command publishers
//subscription to right leg hip flexor
ros::Subscriber sub_r_leg_lhy = nhp_r_leg_lhy.subscribe("/r_leg_lhy_position_controller/
state",1,stateCallback_r_leg_lhy);
//publishers for right and left hip flexor commands
ros::Publisher pub_r_leg_lhy = nhp_r_leg_lhy.advertise<std_msgs::Float64>
("r_leg_lhy_position_controller/command",1);
ros::Publisher pub_l_leg_lhy = nhp_l_leg_lhy.advertise<std_msgs::Float64>
("l_leg_lhy_position_controller/command",1);
// debug... if start spinning here, will get listener data, but won't go any farther with code in main
// (i.e. no command publications)
//ros::spin(); //infinite loop suspends here, if uncomment this line
//return 0;
// for test, define simple harmonic motion for right hip
double freq = 0.5; // motion frequency, in Hz
double phase=0; // argument for sin/cos, derived by integrating frequency
double amp = 1.0; // height of kick
double sample_rate = 10.0; // set sample rate of this program, e.g. to 10Hz
double dt = 1/sample_rate; // corresponding sample period
//send angle command, reiterated at 10Hz. Need a ROS "rate" object to enforce a rate
std_msgs::Float64 cmd_r_leg_lhy; // consistent datatype for angular motion commands
std_msgs::Float64 cmd_l_leg_lhy;
```

```
cmd_r_leg_lhy.data = 0.0; // this datatype has a ".data" component; init to zero
cmd_l_leg_lhy.data = 0.0;
ros::Duration run_duration(10.0); // specify desired duration of this command segment to be N
seconds
ros::Duration elapsed_time; // define a variable to hold elapsed time
ros::Rate naptime(sample_rate); //will perform sleeps to enforce loop rate of "10" Hz

while (!ros::Time::isValid()) {} // simulation time sometimes initializes slowly.
// Wait until ros::Time::now() will be valid
ros::Time birthday= ros::Time::now(); // get the current time, which defines our start time,
called "birthday"
//ROS_INFO("birthday started as %f", birthday.toSec());
ROS_INFO("starting main loop...");
while (ros::ok()) // do work here
{
elapsed_time= ros::Time::now()-birthday;
//ROS_INFO("elapsed time is %f", elapsed_time.toSec());
if ( elapsed_time < run_duration)
{
// send out new command appropriate for this instant;
phase += freq*dt*2*3.14159265; //increment the desired angle
cmd_r_leg_lhy.data = amp*sin(phase); //desired angle of right hip flexion
pub_r_leg_lhy.publish(cmd_r_leg_lhy); // this action causes the right-leg
command to be published
// note: value in sensed_r_leg_lhy, used below, is populated by callback
function, which
// wakes up with new messages from topic /r_leg_lhy_position_controller/
state
cmd_l_leg_lhy.data = -sensed_r_leg_lhy.data;// make left leg swing opposite,
//based on sensed value of right leg--proves if state sensing is working
pub_l_leg_lhy.publish(cmd_l_leg_lhy); //publish the above value as left-leg
command
}
else
{
// done with N seconds of commands; now send out zeros indefinitely to halt
the robot
cmd_r_leg_lhy.data = 0.0;
cmd_l_leg_lhy.data = 0.0;
pub_r_leg_lhy.publish(cmd_r_leg_lhy);
pub_l_leg_lhy.publish(cmd_l_leg_lhy);
}
ros::spinOnce(); // need this to receive callbacks when interleaving subscribe and
publish actions
naptime.sleep(); // this will cause the loop to sleep for balance of time of desired
(100ms) period
//thus enforcing that we achieve the desired update rate (10Hz)
}
return 0; // this code will only get here if this node was told to shut down, which is
// reflected in ros::ok() is false
}
```

The main program invokes: ros::spinOnce() each cycle to allow the callback to receive data. The callback routine receives messages and puts the resulting data into a global variable, accessible to the main program. The main program has the legs kick for 10 seconds then commands angles of 0.0 thereafter.

When this compiled node is run, rxgraph shows the connectivity. The screenshot below (while poor resolution) shows that the above code, with node name "wsn_jnt_cmd_v1", subscribes to published joint states of the right hip, and that it publishes commands to both the left and right hip.


rxgraph
Path: /
Quiet
All topics
Info:
Node [/wsn_jnt_cmd_v1]
Publications:
* /l_leg_lhy_position_controller/command [std_msgs/Float64]
* /r_leg_lhy_position_controller/command [std_msgs/Float64]
* /rosout [rosgraph_msgs/Log]
Subscriptions:
* /r_leg_lhy_position_controller/state [pr2_controllers_msgs/JointControllerState]
* /clock [rosgraph_msgs/Clock]
Services:
* /wsn_jnt_cmd_v1/get_loggers
* /wsn_jnt_cmd_v1/set_logger_level
Pid: 30176
Connections:
* topic: /rosout
* to: /rosout
* direction: outbound
* transport: TCPROS