Celine Berger and Eric Telmer
Humanoids Final Write-up
May 2013

For our final project, we set out to develop a humanoid robot that would be able to perform object manipulation tasks, specifically to connect a hose to a spigot and open the spigot as outlined in the DARPA Virtual Robotics Challenge. The hose task consists of three parts - putting the hose to the faucet, rotating the hose to connect it, and then opening the spigot to enable fluid flow. As we had no prior knowledge or experience with any robotic simulators, we decided to take on the challenge of learning how Robot Operating System (ROS) works and use it for the challenge task.

To begin learning how to work with ROS, we researched on how it worked overall. ROS is a framework for robot software development; it provides a structured communications layer above the host operating systems of a heterogeneous computer cluster. It provides standard operating system services such as hardware abstraction, low-level device control, implementation of commonly used functionality, message-passing between processes, and package management. It has an emphasis on large-scale integrative robotics research, which makes it useful for the wide variety of tasks in the DARPA challenge. ROS is peer-to-peer, tools-based, multi-lingual, and free and open-source. We thought the system's multilingual capability would serve in our favor, as both team members have varying degrees of knowledge of either C++ or Python, both of which are supported. While having no background in ROS seemed detrimental in completing this task, having basic programming knowledge would help us understand the workings of the system.

After background research on ROS, we then delved into working our way through the tutorial, going step by step. The first section we approached was the ROS filesystem. The basic overview of this section includes the Filesystem tools - rospack, roscd, roscd log, rosls, and tab completion. These commands are built-in to manipulate packages and manifests. Rospack gets information from packages. Roscd, part of the rosbash suite, allows the (cd) directory to be changed directly to a package or stack. Roscd log takes us to the folder where ROS stores log files. Rosls like roscd is part of the rosbach suite. It allows us to ls directly into a package by name rather than by absolute path. Tab completion is what it sounds like – pressing the TAB key allows the command line to be filled out automatically.

After learning how to navigate ROS, we learned how to create and build a package.
Part of setting up a package is ensuring it has the proper dependencies. Dependencies are other packages that any one package needs to work. To build a package, we ran a build command that depends on CMakeLists.txt.

We then began to learn about ROS commandline tools - roscore, rosnode and rosrun. Nodes are executables that use ROS to communicate with each other. Messages are a datatype used to subscribe or publish a topic - multiple nodes can subscribe to a topic that they know that other nodes subscribe to so that they are able to communicate with each other. Messages are the tools for this communication. When a node wants to subscribe to a service, it communicates with the Master that acts as the overall manager and tracks and names services and topics. We used roscore to initialize a Master; this must be done before attempting to run any nodes. We used rosrun to start a node out of a package:

rosrun <package_name> <node_name>

Each topic has a defined message type, which can be found using *rostopic type <topic>*. So every message that is published to the topic has to be of the same data structure. To publish to a topic, we use rostopic pub:

rostopic pub <topic> <msg_type> <args>

For example, a message type of velocity:

float32 linear
float32 angular

A service is a function that you send a request using *rosservice type <service>* to figure out the datatype of both requests and responses. To call a service, we use *rosservice call <service> <args>*. A request consists of the arguments that you provide a function, the response is what it returns. The ROS parameter server is a space where global variables can be stored.

With roslaunch, we can launch packages. A launch file contains instructions on how to run multiple nodes and link them together. Using the launch file and launch command as read by roslaunch runs these nodes and maps inputs and outputs.

Using this basic knowledge of ROS, we attempted to understand how to use the object_manipulator package to grasp and then turn a valve. We were planning on using the PR2 simulator in conjunction with the object_manipulator package. We had difficulty in getting on getting the object_manipulator package to work with PR2. The following is an overview of the steps we would need to take:

- Have it perform a grasp movement, and record standard values for the fields of the pickup message (PickupGoal.msg) that we would use later.
- We would send this same message later, but modify the fields in order to perform the turning of the spigot.

The object_manipulator provides two SimpleActionServers:

/object_manipulator_pickup: requests that an object be picked up.

- action definition:
  - goal: PickupGoal.msg
  - result: PickupResult.msg
  - feedback: no feedback provided

/object_manipulator_place: requests that a previously picked up object be placed somewhere in the environment

- action definition:
  - goal: PlaceGoal.msg

- result: PlaceResult.msg
- feedback: no feedback provided

Our plan was to use the PickupGoal.msg to force the hand to grasp and then turn the identified valve. We were not focusing on how the valve is identified.

A PickupGoal.msg has the following definition:

```
string arm_name
object_manipulation_msgs/GraspableObject target
object_manipulation_msgs/Grasp[] desired_grasps
object_manipulation_msgs/GripperTranslation lift
string collision_object_name
string collision_support_surface_name
bool allow_gripper_support_collision
bool use_reactive_execution
bool use_reactive_lift
bool only_perform_feasibility_test
bool ignore_collisions
arm_navigation_msgs/Constraints path_constraints
        arm_navigation_msgs/OrderedCollisionOperations
                additional_collision_operations
arm_navigation_msgs/LinkPadding[] additional_link_padding
object_manipulation_msgs/GraspableObject[] movable_obstacles
float32 max_contact_force
```

In order to force the robot to not attempt to lift the valve up, we would have had to edit the “lift” message. It has the following definition:

```
geometry_msgs/Vector3Stamped direction
float32 desired_distance
float32 min_distance
```

We would set all of these to 0.

In order to trick the robot into turning the valve we would continuously generate PickupGoal.msg with zero lift and new desired_grasps each time.
desired_grasps has the following definition:

```
    sensor_msgs/JointState pre_grasp_posture
    sensor_msgs/JointState grasp_posture
    geometry_msgs/Pose grasp_pose
    float64 success_probability
    bool cluster_rep
    float32 desired_approach_distance
    float32 min_approach_distance
    object_manipulation_msgs/GraspableObject[] moved_obstacles
```

So each new message would have pre_grasp_posture equal to the previous messages grasp_posture. Each iteration, we would change the grasp_posture and grasp_pose. The desired_approach_distance and min_approach_distance would remain at zero.

grasp_posture:

```
    string[] name
    float64[] position ;in rad
    float64[] velocity ;in rad/s
    float64[] effort   ;in N
```

grasp_pose:

```
    Point position
    Quaternion orientation
```

point:

```
    float64 x
    float64 y
    float64 z
```

Quaternion:

```
    float64 x
    float64 y
    float64 z
    float64 w
```

We would set velocity and effort to the same value each time. Point would also remain the same as we don't want the hand moving, just turning.
This is how we would change position and orientation:

Position = prev_position + velocity;
orientation = prev_orientation rotated by vel, using a rotation matrix.

If we received a PickupResult Message that shows failure, that means that we have rotated the hand as much as possible, in this situation we will un-grasp, retreat the hand , rotate it back to its initial position and the repeat until the valve is open.

By continuously generating Pickup messages with zero lift, and with the grasp slowly rotating we could achieve our goal of slowly turning a valve.

While we couldn't figure out how to actually get this to work, we did effectively get a basic overview of ROS.

File: object_manipulation_msgs/Grasp.msg

Message Definition

```
sensor_msgs/JointState pre_grasp_posture

sensor_msgs/JointState grasp_posture

geometry_msgs/Pose grasp_pose

float64 success_probability

bool cluster_rep

float32 desired_approach_distance

float32 min_approach_distance

object_manipulation_msgs/GraspableObject[] moved_obstacles
```