Introduction to ROS and the Scarab Robots

Amanda Prorok
University of Pennsylvania, GRASP Lab.

TerraSwarm - Programming the Swarm Workshop
UC Berkeley, May 28, 2015
Overview

1. ROS: concepts
   • computation graph
   • topic negotiation
   • network configuration
   • publisher / subscriber example

2. Using ROS with the Scarab robot
   • setup
   • publisher / subscriber example
   • using roslaunch

3. Application:
   • sending waypoints to the Scarab (movies)
   • simulator (demo)

4. Interfacing ROS with Swarmlets
   • 2 suggestions
What is ROS?

ROS is an open-source, meta-operating system, see [www.ros.org](http://www.ros.org):

“The Robot Operating System (ROS) is a set of software libraries and tools that help you build robot applications. From drivers to state-of-the-art algorithms, and with powerful developer tools, ROS has what you need for your next robotics project.”

ROS provides:
- a collection of package management and software building tools
- implementation of commonly-used functionality
- architecture for distributed inter-process and inter-machine communication and configuration
- hardware abstraction & low-level device control
## ROS Filesystem

ROS resources that you encounter on disk:

<table>
<thead>
<tr>
<th><strong>packages:</strong></th>
<th>Packages are the main unit for organizing software in ROS. A package may contain ROS runtime processes (nodes), a ROS-dependent library, datasets, configuration files.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>repositories:</strong></td>
<td>A collection of packages which share a common version control system.</td>
</tr>
<tr>
<td><strong>package manifests:</strong></td>
<td>Manifests (package.xml) provide metadata about a package</td>
</tr>
<tr>
<td><strong>message types:</strong></td>
<td>Message descriptions, stored in my_package/msg/MyMessageType.msg, define the data structures for messages sent in ROS</td>
</tr>
<tr>
<td><strong>service types:</strong></td>
<td>Service descriptions, stored in my_package/srv/MyServiceType.srv, define the request and response data structures for services in ROS</td>
</tr>
</tbody>
</table>

Navigating the filesystem:

**roscd, rospack, ros1s**
ROS Graph Concepts

ROS runtime **graph** is a peer-to-peer network of processes that are loosely coupled via the ROS communication infrastructure.

<table>
<thead>
<tr>
<th><strong>nodes:</strong></th>
<th>A node is an executable that uses ROS to communicate with other nodes.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>messages:</strong></td>
<td>ROS data type used when subscribing or publishing to a topic.</td>
</tr>
<tr>
<td><strong>topics:</strong></td>
<td>Nodes can publish messages to a topic as well as subscribe to a topic to receive messages.</td>
</tr>
<tr>
<td><strong>master:</strong></td>
<td>Name service for ROS: name registration and lookup; negotiates connections</td>
</tr>
<tr>
<td><strong>rosout:</strong></td>
<td>ROS equivalent of stdout/stderr</td>
</tr>
<tr>
<td><strong>parameter server:</strong></td>
<td>Stores persistent configuration parameters</td>
</tr>
<tr>
<td><strong>roscore:</strong></td>
<td>3 programs: master + rosout + parameter server</td>
</tr>
</tbody>
</table>
Nodes:
- Nodes are processes that perform computation.
- A ROS node is written with the use of a ROS client library: `roscpp` or `rospy`

Topics:
- Messages are routed via a transport system with publish / subscribe semantics
- A node sends out a message by publishing it to a given topic
- The topic is a name that is used to identify the content of the message
- Analogy: topic as a strongly typed message bus (according to .msg spec)
- Asynchronous ‘stream-like’ communication
- TCP/IP or UDP transport
Message Types

ROS uses a simplified Message description language for describing the data values (Messages) that ROS nodes publish. Message descriptions are stored in `.msg` files in the `msg/` subdirectory of a ROS package.

The format of a Message description is a list of data field descriptions and constant definitions on separate lines:

```
int32 x
int32 y
```

Field types can be:
1. a built-in type
2. names of custom or built-in Message descriptions
3. fixed- or variable-length arrays (lists) of the above
4. the special Header type, which maps to std_msgs/Header
1. I am publishing on /topic, reach me at HOSTNAME
2. I am subscribing to /topic
3. Contact “publisher” at HOSTNAME
4. I am subscribing to /topic
5. Data on /topic
ROS TCP Topics

advertise("scan")
ROS TCP Topics

- master
  - topic: scan
- laser
- visualizer
ROS TCP Topics

- master
  - topic: scan
    - subscribe("scan")

laser

visualizer
ROS TCP Topics

master

topic: scan

subscribe("scan")

laser

visualizer
ROS TCP Topics

- master
- topic: scan

laser \(\xrightarrow{\text{scan(TCP)}}\) visualizer
ROS TCP Topics

master

topic: scan

laser

publish(scans)

visualizer
**Network Configuration**

ROS_MASTER_URI is a required setting that tells nodes where they can locate the master. It should be set to the XML-RPC URI of the master.

When a ROS node advertises a topic, it provides a hostname:port combination (a URI) that other nodes will contact when they want to subscribe to that topic. It is important that the hostname that a node provides can be used by all other nodes to contact it.

**Single machine configuration:**

```bash
$ export ROS_HOSTNAME=localhost
$ export ROS_MASTER_URI=http://localhost:11311
```

**Multiple machines configuration:**

```bash
$ export ROS_HOSTNAME=scarab42.wifi
$ export ROS_MASTER_URI=http://scarab44.wifi:11311
```
Useful Commands

$ rosnode list
$ rosnode info [node_name]
$ rosrun [package_name] [node_name]
$ rostopic echo [topic_name]
$ rostopic type [topic]
$ rostopic pub [topic] [msg_type] [args]
$ rospack find [package_name]

Useful links:
http://wiki.ros.org/ROS/Tutorials/UnderstandingTopics
http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes
http://wiki.ros.org/ROS/NetworkSetup
Using ROS with Scarabs
Setup

1. Install Ubuntu 14.04 (on binary compatible machine)
2. Install the full desktop version of ROS Indigo
3. Configure ROS environment, see instructions:
   http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment
4. Install dependencies (see Appendix)
5. Download (and compile) Ben Charrow’s *scarab* repository
   https://github.com/bcharrow/scarab/tree/bcharrow-devel
6. Configure your .bashrc file on all machines (including robots):
   $ export ROS_HOSTNAME=myhostname
   $ export ROS_MASTER_URI=http://masterhostname:11311
Scarab Packages

Repository:
https://github.com/bcharrow/scarab/tree/bcharrow-devel

Main packages:
• roboclaw: motor driver
• laser_odom: odometry estimate
• scarab: launch files
• scarab_twist: keyboard driving
• hfn: human friendly navigation

Also used from ROS repository (included in Indigo installation):
• gmapping: occupancy grid mapping
Publisher / Subscriber: Example

On computer

On robot
Launching Multiple Nodes

ROS provides a way to configure and launch a collection of ROS nodes. Use the `roslaunch` program with a `.launch` XML file that configures system:

```
roslaunch [packagename] [file.launch]
```

Allows us to:
- easily run multiple ROS nodes locally and remotely via SSH
- set parameters on the parameter server
- automatically re-spawn nodes if they die
- change node names, namespaces, topic names without recompiling

Examples:
```
$ roslaunch scarab scarab.launch
$ roslaunch scarab gmapping.launch
$ roslaunch scarab hfn.launch
```
Example: Create Map
Example: Navigate to Goal
The `actionlib` package provides tools to create servers that execute long-running goals that can be preempted. It also provides a client interface in order to send requests to the server.

Action specification:
1. **Goal**: sent to an ActionServer by an ActionClient.
2. **Feedback**: provides server implementers a way to tell an ActionClient about the incremental progress of a goal
3. **Result**: sent from the ActionServer to the ActionClient upon completion of the goal (sent just once).

The action specification is defined using a `.action` file

Useful link:
http://wiki.ros.org/actionlib
1. Create a node that publishes messages to goal topic.
2. Create a node that subscribes to goal topic:
   .../scarab/hfn/scripts/goal_to_action.py

This python program receives goal points from publisher node, and calls a callback function to transform it to an action:

```python
rospy.Subscriber("goal", PoseStamped, callback)
```

The goal is processed in callback function, and sent to a MoveAction client:

```python
# create client in main
client = actionlib.SimpleActionClient('move', MoveAction)

# send goal to action server
client.send_goal(goal)
```
Using an Existing Map

We can create a map once and use it subsequently when operating in that environment (for localization navigation purposes, etc). Use map_server package to save and load maps.

```
rosrun map_server map_saver -f mymapfilename map:=mymaptopic
```

Remap the topic name to match the topic name used by your subscriber.

Useful links:
http://wiki.ros.org/map_server#map_saver
http://wiki.ros.org/Remapping%20Arguments
For testing purposes, we can run code for the Scarab robot in a kinematic simulator. The code is identical to the one we would run on the actual hardware.

Steps:
1. run ROS with a local master URI
2. roslaunch scarab simulation.launch
3. rosrunt rviz rviz
4. choose the appropriate displays to view (i.e., robot frame, map, laser)
5. roslaunch scarab hfn.launch
6. send waypoints to /goal topic
Idea:
Send a waypoint (or vector of waypoints) to the robot using the hfn package.

Suggestion 1:
Adapt the script .../scarab/hfn/scripts/goal_to_action.py so that goal points are received from a swarmlet (via accessors).

Instead of using a ROS publisher of /goal topic, grab the goal data from an accessor (from within this Python script).
Replace lines
rospy.Subscriber("goal", PoseStamped, callback)
rospy.spin()

with something like the following:
while AccessorDataAvailable():
    data = GetAccessorData()
    callback(data)
Idea:
Send a waypoint (or vector of waypoints) to the robot using the hfn package.

Suggestion 2:
Use the script ../scarab/hfn/scripts/goal_to_action.py as it is.

Create your own ROS publisher node that sends messages to /goal topic using client libraries (c++ or python). This node represents the accessor-to-ROS interface.
Appendix
## Dependencies

<table>
<thead>
<tr>
<th>accache</th>
<th>strace</th>
</tr>
</thead>
<tbody>
<tr>
<td>openssl-server</td>
<td>ipython</td>
</tr>
<tr>
<td>ttf-dejavu</td>
<td>python</td>
</tr>
<tr>
<td>libnncurses-dev</td>
<td>python-serial</td>
</tr>
<tr>
<td>libnl-dev</td>
<td>git-core</td>
</tr>
<tr>
<td>libarmadillo-dev</td>
<td>build-essential</td>
</tr>
<tr>
<td>libnncurses5-dev</td>
<td>python-yaml</td>
</tr>
<tr>
<td>libcgal-dev</td>
<td>cmake</td>
</tr>
<tr>
<td>ros-indigo-desktop-full</td>
<td>minicom</td>
</tr>
<tr>
<td>ros-indigo-openni2-launch</td>
<td>iputils-arping</td>
</tr>
<tr>
<td>ros-indigo-hokuyo-node</td>
<td>iputils-tracepath</td>
</tr>
<tr>
<td>ros-indigo-joystick-drivers</td>
<td>iputils-clockdiff</td>
</tr>
<tr>
<td>ros-indigo-navigation</td>
<td>cfengine2</td>
</tr>
<tr>
<td>ros-indigo-octomap-mapping</td>
<td>console-common</td>
</tr>
<tr>
<td>ros-indigo-gmapping</td>
<td>acpid</td>
</tr>
<tr>
<td>ros-indigo-octomap-rviz-plugins</td>
<td>ifplugd</td>
</tr>
<tr>
<td>python-rosinstall</td>
<td>batctl</td>
</tr>
<tr>
<td>python-rosdep</td>
<td>batctl-dbg</td>
</tr>
<tr>
<td>iperf</td>
<td>traceroute</td>
</tr>
<tr>
<td>python-pip</td>
<td>olsrd</td>
</tr>
<tr>
<td>libgsl0-dev</td>
<td>olsrd-plugins</td>
</tr>
<tr>
<td>libgsl0-dev</td>
<td>nfs-common</td>
</tr>
<tr>
<td>libgsl0-dbg</td>
<td>rsync</td>
</tr>
<tr>
<td>libgsl0-dbl</td>
<td>subversion</td>
</tr>
</tbody>
</table>
Topics should be used for continuous data streams (sensor data, robot state, ...).

Services should be used for remote procedure calls that terminate quickly, e.g. for querying the state of a node or doing a quick calculation. They should never be used for longer running processes, in particular processes that might be required to be preempted if exceptional situations occur. These processes should never change or depend on state to avoid unwanted side effects for other nodes.

Actions should be used for everything that moves the robot or that runs for a longer time such as perception routines that are triggered by some node and need a couple of seconds to terminate. The most important property of actions is that they can be preempted, and preemption should always be implemented cleanly by action servers. Another nice property of actions is that they can keep state for the lifetime of a goal, i.e. if executing two action goals in parallel on the same server, for each client a separate state instance can be kept since the goal is uniquely identified by its ID.
**Topics:** Continuous data flow. Data might be published and subscribed at any time independent of any senders/receivers. Many to many connection. Callbacks receive data once it is available. The publisher decides when data is sent.

**Service/Actionlib:** On-demand connection for one specific task. Service calls/Actionlib tasks are processed when the client decides to request so. Tasks take time to complete.

**Service/Actionlib** are thus very similar and can actually be used interchangeably in their functionality. However, they serve different purposes:

**Service:** Simple blocking call. Mostly used for comparably fast tasks as requesting specific data. Semantically for processing requests.

**Actionlib:** More complex non-blocking background processing. Used for longer tasks like execution of robot actions. Semantically for real-world actions.