Capabilities

As part of the Robotics-in-Concert (ROCON) project, it was determined that the abstractions used to describe different robots were not sufficiently powerful to support truly portable robot applications (apps). For the ROCON project, apps are central to the design as they provide a generic means for interfacing with different robots. In order to improve the portability of apps between robots, the concept of capabilities was developed to provide an interface upon which both robot developers and application developers can build.

The Robot Operating System (ROS), on which the ROCON system builds, provides a good mechanism for defining interfaces between parts of the robot system. ROS includes a message passing system that enforces the interface by which two programs can communicate. Anyone can define new messages, but in order to encourage portability and software reuse, most developers rely on a standard set of commonly used messages. So, for example, programs that produce and consume data from laser scanners can talk with each other by simply agreeing on a name over which to communicate. The contents of the messages are defined in a separate, common place (in the case of laser drivers, this package is called sensor_msgs). In this way the developer who built the laser driver and the developer of the program that uses the laser's data can develop code separately from each other. Connecting the output of one program to the input of the other is still the responsibility of the system integrator.

The capabilities system goes one step further by not only defining the type of data being sent, but also over which named topics the data will be sent, effectively defining how the system components will be connected. ROS concepts other than topics are also covered by capability interfaces, including: services, actions, parameters, and dynamic parameters. By defining the required types and names for each of the ROS concepts, robot developers can set up their robot to send and receive data according to the interface, and app developers can depend on that interface’s being available when designing their applications.

This approach is analogous to how the Android mobile operating system allows Android apps to require and access "capabilities" provided by the mobile device. For example, an app may state that it requires the Internet and a GPS location so that it can perform its task. In a similar way, a robot app can require the ability to navigate and access to an RGB camera to perform its task.

Once these capability interfaces are in place, the ROCON system can do things like: start capabilities on a robot so that an app can be used with it; only run those capabilities that are required by currently running applications (e.g., to save power and/or processing); and determine whether an app can be used on a robot or not based on the capabilities that it requires.

Overview of Capabilities

There are three main concepts surrounding the capabilities system: capability interfaces, semantic capability interfaces, and capability providers. A capability interface is a specification file that defines the ROS topics, services, actions, parameters, and dynamic parameters that are required to provide and/or use a particular capability. For example, a capability interface may define the DifferentialMobileBase capability to be anything that can take linear and angular velocity in a 'Twist' type message over the '/cmd_vel' ROS topic. This file only specifies the interface, not its implementation.

A semantic capability interface redefines a normal capability interface, giving it some semantic meaning, but not changing the interface beyond the ROS names. For example, the 'FrontRGBCamera' semantic capability interface might redefine the standard 'RGBCamera' capability interface, changing only the topic names from '/camera/image' to '/front_camera/image' and implying that it is a front facing camera. The idea here is that an app can prefer a 'FrontRGBCamera' if it is available but will settle for a 'RGBCamera' because the two interfaces are the same except for the topic renaming.

Finally, the capability providers are specifications that define an implementation for a capability interface or semantic capability interface. The specification says which interface it implements and references the resource that must be run to provide the implementation. Additionally, providers can depend on other capability interfaces. In this way, higher level capability providers can build on lower level capabilities. For example, the 'Navigation' capability might have a capability provider that depends on the

'DifferentialMobileBase' capability interface, which makes sense because the robot needs to be able to move before it can be asked to navigate.

State of Current Work

The capabilities system is considered to be mostly complete: the libraries that discover, parse, validate, and serve the capabilities have been implemented and tested. As well, a client API has been implemented and tested in both ROS primitives and Python, allowing us to integrate the capabilities system into the rest of the ROCON project.

The capabilities are served through a ROS Node that uses these libraries called the “capability server”. This server can list the available capability interfaces and capability providers as well as provide a list of capabilities that are currently running. The server can also be used to start and stop capabilities explicitly, and it provides the notion of reference-counted capabilities. The reference counted capabilities allow a user of the capability server to “use” a capability and when it is done, “free” that capability. In this way, multiple agents can use the same capability without of the risk of having some other agent shut down that capability while the others are still using it. This effectively allows for shared ownership of a running capability.

The capability server provides a ROS API that is implemented using ROS messages and services with a ROS topic for event feedback. There is also a Python “client” API that provides an easy-to-use Python interface for interacting with the capability server. Under the hood this client API uses the ROS API so that the client API can be used on a remote computer, which is different from the computer that the capability server is running on.

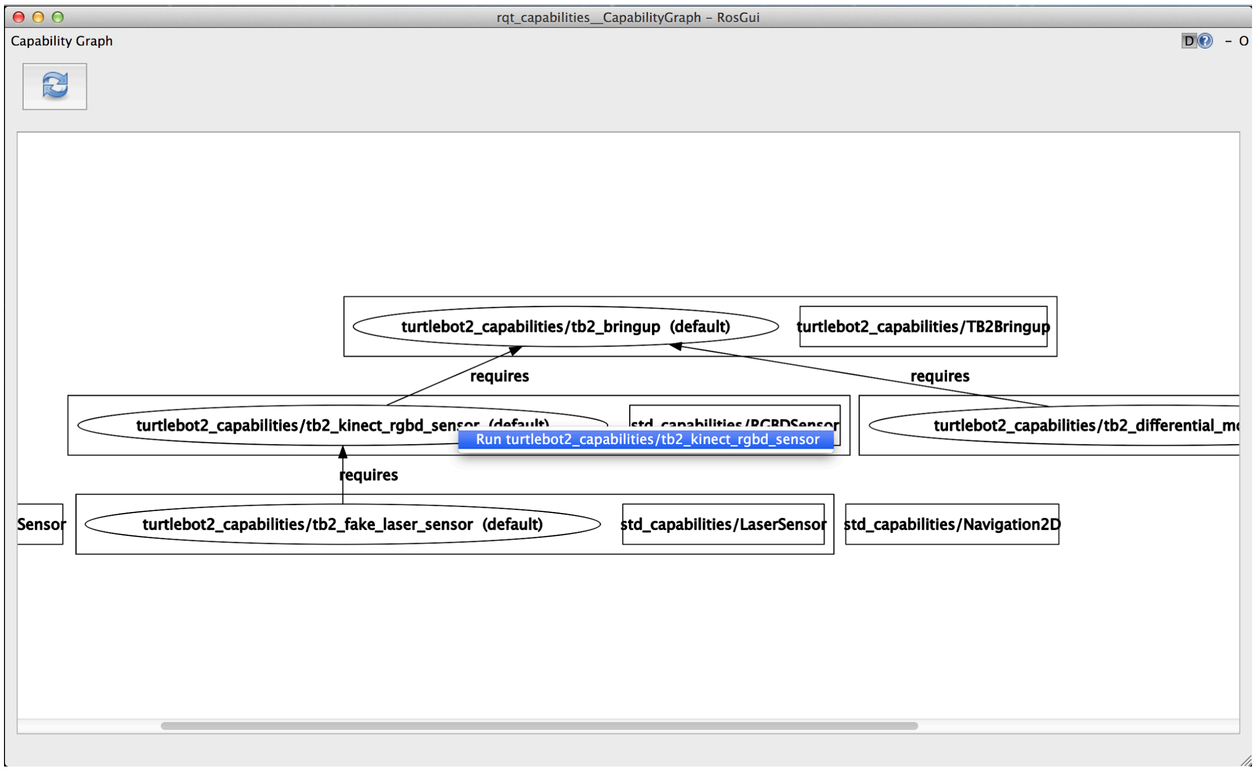
In addition to the capability server we have developed an initial set of standard, robotics-related capability interfaces on which most developers can agree. This allows for more reuse of code and portability between different parts of the community. For example, this is the preliminary ‘DifferentialMobileBase’ capability interface spec:

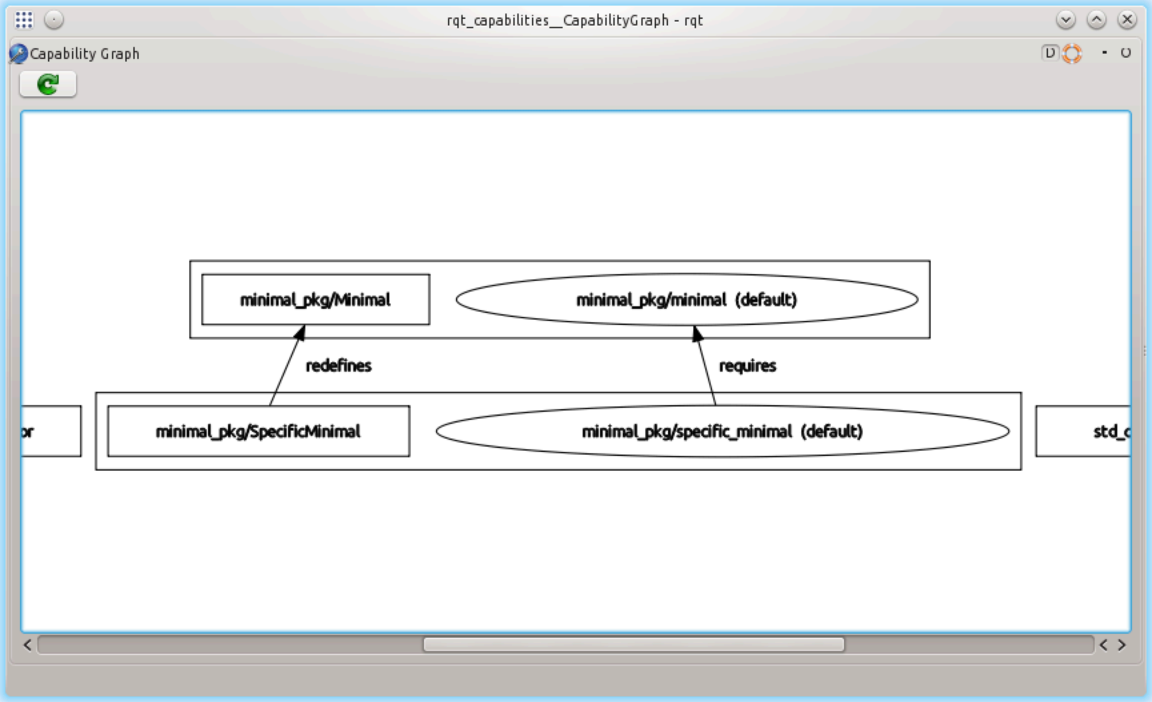
```
%YAML 1.1
```

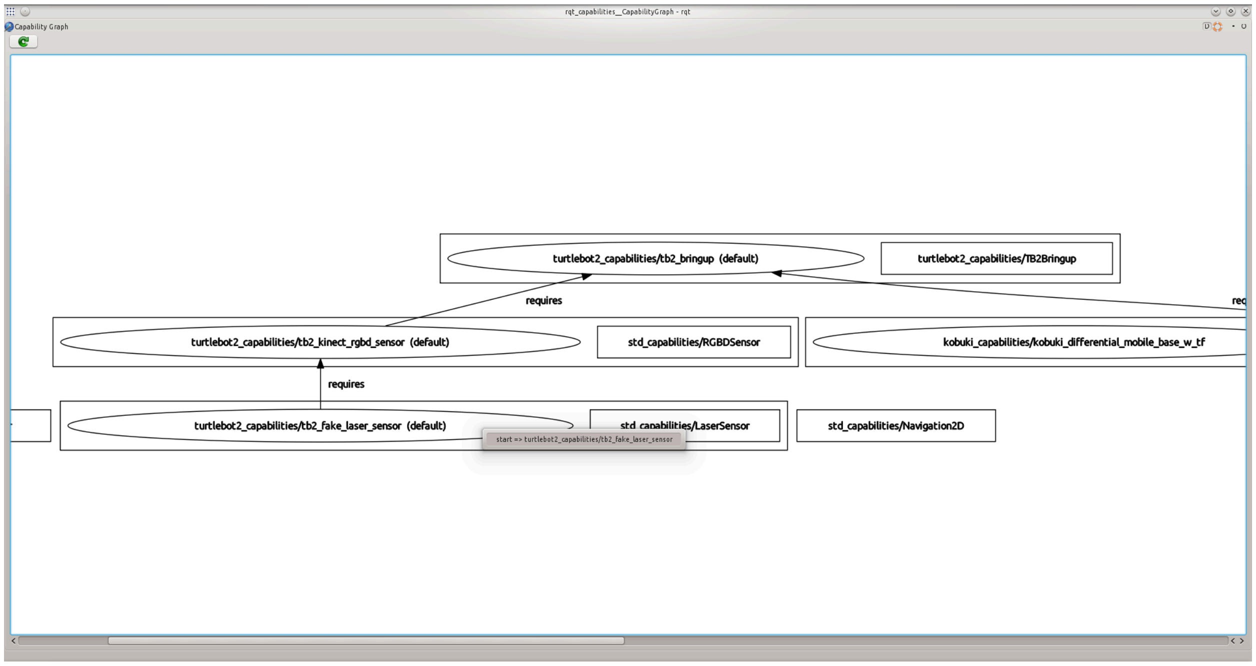
```yaml
---
name: DifferentialMobileBase
spec_version: 1
spec_type: interface
description: 'This interface describes the basic ROS interface for controlling a
differential mobile base'
interface:
  topics:
    requires:
      'cmd_vel':
        type: 'geometry_msgs/Twist'
        description: 'Command Velocities to move the base of the robot, only the x
component of the linear velocity and the z, also known as yaw, component of the
angular velocity are used. The units follow REP-0103 such that the linear velocity
is measured in m/s and the angular rate is measured in radians/s.'
    provides:
      'odom':
        type: 'nav_msgs/Odometry'
        description: 'Odometry of the mobile base. The message should have valid
values for the x and y components of the pose.pose.position as well as the z, also
known as yaw, component of the pose.pose.orientation. If velocities are available
they should be filled into the twist.twist submessage. If the covariance of either
the pose or the twist is known it should also be filled in.'
```

Additionally, as part of our testing and initial roll-out of the capabilities system, we developed capability providers and interfaces for the Turtlebot 2.

To aid with development, a graphical tool was built on top of the ROS Qt framework (rqt). This tool, called 'rqt_capabilities', provides a graph view of capability interfaces and providers and how they are related. From the graphical tool you can view the running capabilities, start capabilities, and stop capabilities.


rqt_capabilities__CapabilityGraph – RosGui
Capability Graph
turtlebot2_capabilities/tb2_bringup (default)
turtlebot2_capabilities/TB2Bringup
requires
requires
turtlebot2_capabilities/tb2_kinect_rgbd_sensor (default)
std_capabilities/RGBDSensor
turtlebot2_capabilities/tb2_differential_m
Run turtlebot2_capabilities/tb2_kinect_rgbd_sensor
requires
Sensor
turtlebot2_capabilities/tb2_fake_laser_sensor (default)
std_capabilities/LaserSensor
std_capabilities/Navigation2D


rqt_capabilities__CapabilityGraph - rqt
Capability Graph
minimal_pkg/Minimal
minimal_pkg/minimal (default)
redefines
requires
minimal_pkg/SpecificMinimal
minimal_pkg/specific_minimal (default)


rqt_capabilities__CapabilityGraph - rqt
Capability Graph
turtlebot2_capabilities/tb2_bringup (default)
turtlebot2_capabilities/TB2Bringup
requires
turtlebot2_capabilities/tb2_kinect_rgbd_sensor (default)
std_capabilities/RGBDSensor
kobuki_capabilities/kobuki_differential_mobile_base_w_tf
requires
turtlebot2_capabilities/tb2_fake_laser_sensor (default)
std_capabilities/LaserSensor
std_capabilities/Navigation2D
start => turtlebot2_capabilities/tb2_fake_laser_sensor

```
Integration with the ROCON project
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Capabilities were designed to help with the “App installation”, “App updating”, “fetching of App information”, and “App execution” goals of the ROCON project. Capabilities are used to capture some of the hardware and software requirements for an App while abstracting the configuration and implementation details. For instance, If you would like to install an App, the process for determining whether installation is possible for a given “concert client” involves comparing which capabilities an App requires against which capabilities are implemented or are available for the current “concert client.” This allows for abstractions on hardware and software capabilities that are not possible by using explicit software dependencies alone.

The capability specs are separated into interfaces and providers, which has allowed for the federated development of standard interfaces with platform specific providers. For instance, the “std_capabilities/LaserSensor” Capability Interface describes the minimum interface for a generic laser scanner based sensor:

```yaml
%YAML 1.1
---
name: LaserSensor
spec_version: 1
```

```
spec_type: interface
description: 'This is the standard laser sensor interface used in ROS'
interface:
  topics:
    requires:
      'scan':
        type: 'sensor_msgs/LaserScan'
        description: 'Standard message format for LaserScanners in ROS. See the
message description for details on how to fill it out:
http://docs.ros.org/api/sensor_msgs/html/msg/LaserScan.html Additionally, see REP-0138
which is related to LaserScan messages: http://www.ros.org/reps/rep-0138.html'
```

And the “turtlebot_capabilities/depthimage_to_laserscan” Capability Provider, which is distributed with the Turtlebot specific code, describes the way that the Turtlebot implements this interface:

```
%YAML 1.1
---
name: depthimage_to_laserscan
spec_version: 1
spec_type: provider
description: 'Implements the standard LaserSensor capability for the TurtleBot by
providing a fake laser scan using the depth image from the 3D sensor.'
implements: std_capabilities/LaserSensor
launch_file: 'launch/depthimage_to_laserscan.launch'
depends_on:
    'std_capabilities/RGBDSensor':
        provider: 'turtlebot_capabilities/kinect_rgbd_sensor'
```

This is a good example of how the capabilities system allows apps to depend on a generic capability defined in a common software package, but can be supported by implementations which are specific to a robot or concert client.

ROCON App Platform
===================

In order to meet these goals for the ROCON project:

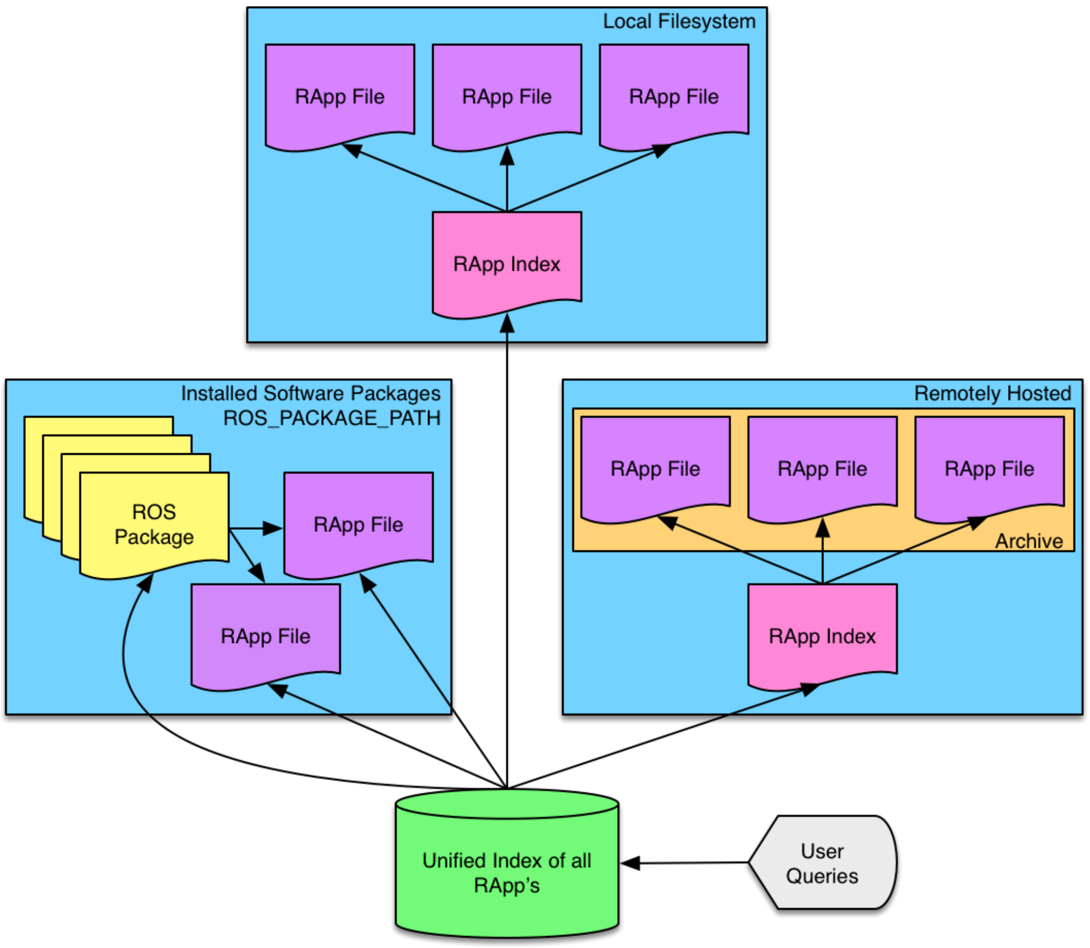
- Download an App from a repository of App’s
- Install an App onto a concert client
- Update an App already on a concert client
- Remove an App which was previously installed
- Download meta information for an App from a repository of App’s

the concept of the ROCON App Platform was developed. The App Platform collects meta information about ROCON App's which includes the direct package dependencies (software packages) of the App, the capabilities that are required for this App to function, and information for how to run the App as well as assets such as graphical icons for a user interface.

Building an App Index

^^^^^^^^^^^^^^^^^^^^^

To meet the goals stated above, a decentralized system of repositories was developed such that App meta information could be drawn from installed software packages, the local user filesystem, or a remote repository of App's bundled with their resources. Users of the ROCON App Platform should be agnostic to the source of the App definitions, allowing them to easily add App's while testing, but also keep up-to-date with the remote App repositories.



Repositories of App’s on the local filesystem and hosted remotely must be indexed in a single file which serves as a single point of entry for discovering App’s and querying their meta information. When discovering App’s and loading App meta information from installed software packages, the App Platform uses the ROS_PACKAGE_PATH to find App’s by parsing the catkin package.xml files to get the relevant meta information for those App’s. So, in this case, the ROS_PACKAGE_PATH and the catkin package.xml files server as the indexing method.

Local repositories of App’s can be indexed using the tools provided by the ROCON App Platform, and can be additionally prepared for remote access by bundling all the relevant resources into an attached archive. Once the index and archive have been prepared from a local repository of App’s, they can be hosted remotely.

Querying App Index
^^^^^^^^^^^^^^^^^^

Once the sources of App’s have been fetched and processed, then the user of the ROCON App Platform can make queries about the App’s without worrying about where the information for those App’s are stored. For instance, a user could ask for a list of App’s are installed, which App’s need updating (there are new dependencies or missing dependencies), or even if it is possible to install.

Different factors are involved in deciding whether an App is installable on a given concert client. For instance, to be considered installed an App must have all of its software package dependencies installed and the concert client must implement all of the capabilities on which the App depends. The indexed App’s and their meta information support these kinds of queries using the ROCON App Platform.

Installation and Removal of Apps
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

ROCON (or robot) apps are lightweight. They are essentially launchers with no code plus a bit of extra information about their public interface, specific configuration, icon and description of their features. The actual logic of each app is implemented in its dependencies, which can be shared among multiple apps. For installation this means, that it is not the app, which gets installed, but its dependencies. The information contained in the app has already been gathered and stored in memory during the previous step, when fetching it from a local or remote repository.

Since using shared dependencies greatly reduces the disk space needed for installed apps, the removal of apps is not a required feature of the app framework for now. If the necessity for app removal arises in the future, a solution will be implemented, which handles the complex management of shared dependencies.