ROS Basics - Topics, Services & Messages

ARRA / AR2A

Advancements for Robotics in Rescue Applications

March 13, 2016

ARRA/AR2A

What do we want?

- ARRA / AR2A aims to improve the current state of technology of robotics in rescue applications.

Who are we?

- A volunteer non-profit organisation of robotic enthusiasts.

How can you help?

- Check us out at `https://github.com/ar2a`

License information

- **CC-BY-SA 4.0**
  `https://creativecommons.org/licenses/by-sa/4.0/`

Section 1

Introdution

Overview

ROS consists of 3 Levels:

- Filesystem
  - Packages
  - Disk-Files
  - Message types
  - ...
- Computation Graph
  - Nodes
  - Topcis
  - Services
  - ...
- Community
  - Repositories
  - Wiki
  - ...

Node

- Process, which performs computation
- Control different parts of the robot (motors, rangefinder)
- Communication via messages/services
- Peer-to-peer network, master only registers nodes, topics, ...
- Identified by the graph resource name (hierarchical naming structure for topics/services/nodes/...)
- Name can be remapped at runtime
- Information about the running nodes can be shown with the tool “rosnode“

Section 2

ROS Topic

Message

- Data structure, which can also be nested
- Standard data-types: int, float, string, time, array
- Special type “Header“: contains timestamp and coordinate frame
- Simple textfiles: each line contains the type and name of a field
- Only useful for unidirectional communication; subscribers are unaware of all the publishers
- Information about messages can be shown with the tool “rosmsg“

Create a custom message (1)

- Create a .msg-file. For example:

```
Header header
string text
int16 number
```

- Naming convention for msg-types: name of the package + '/' + name of the .msg file
- **Example:** the file "std_msgs/msg/String.msg" results in a message type of "std_msgs/String"

Create a custom message (2)

- Enable the generation of source code from the msg-files
- In the **package.xml** uncomment/add:

```
<build_depend>message_generation</build_depend>
<run_depend>message_runtime</run_depend>
```

Create a custom message (3)

- Example **CMakeList.txt** for a package with a message depending on std_msgs

```
# Get information about this package's buildtime dependencies
find_package(catkin REQUIRED COMPONENTS message_generation)

# Declare the message files to be built
add_message_files(FILES MyMessage.msg)

# Generate the language-specific message and service files
generate_messages(DEPENDENCIES std_msgs)

# Declare that this catkin package's runtime dependencies
catkin_package(CATKIN_DEPENDS message_runtime
               std_msgs include_dirs)
```

Topic

- Simple string specified when publishing/subscribing to a message

```
ros::Publisher pub =
        nh.advertise<std_msgs::String>("topic_name", 5);
```

- The topic-type is defined by the message type → only one type per topic
- A subscriber will not accept a message of a different type

Publish to a topic

- Example for publishing a standard message (string) to a topic

```
ros::Publisher pub =
        nh.advertise<std_msgs::String>("topic_name", 5);
std_msgs::String str;
str.data = "helloworld";
pub.publish(str);
```

- Advertise:
  - The first parameter is the topic-name
  - The second parameter specifies the maximum number of outgoing messages queued for delivery

Subscribe to a topic

Function:

- ros::Subscriber subscribe(...)

Parameters:

- **const std::string& topic**: topic name
- **uint32_t queue_size**: queue for incoming messages (0 is infinite, but dangerous)
- <**callback**>: e.g. class method pointer and class instance pointer
- **const ros::TransportHints& transport_hints**: hint for the transport layer (preferred UDP transport, ...)

rostopic (1)

rostopic is a command line based debug tool which allows you to retrieve information about ROS topics.

Available sub-commands:

- rostopic bw: *display bandwidth used by topic*
- rostopic echo: *print messages to screen*
- rostopic hz: *display publishing rate of topic*
- rostopic list: *print information about active topics*
- rostopic pub: *publish data to topic*
- rostopic type: *print topic type*

rostopic (2)

Example: A given turtle demo was used for the following example.
The *list* command prints information about the active topics on the screen.
Command:

```
$ rostopic list -v
```

Output:

```
Published topics:
* /turtle1/color_sensor [turtlesim/Color] 1 publisher
* /turtle1/cmd_vel [geometry_msgs/Twist] 1 publisher
* /rosout [rosgraph_msgs/Log] 2 publishers
* /rosout_agg [rosgraph_msgs/Log] 1 publisher
* /turtle1/pose [turtlesim/Pose] 1 publisher

Subscribed topics:
* /turtle1/cmd_vel [geometry_msgs/Twist] 1 subscriber
* /rosout [rosgraph_msgs/Log] 1 subscriber
```

rostopic (3)

Example: The *echo* command shows the messages from the topics on the screen.

Command:

```
$ rostopic echo /turtle1/cmd_vel
```

Output:

```
linear:
x: 2.0
y: 0.0
z: 0.0
angular:
x: 0.0
y: 0.0
z: 0.0
- - -
```

rosmsg (1)

rosmsg is a command-line tool for displaying information about ROS Message types.

Available sub-commands for rosmsg:

- rosmsg show: Show message description
- rosmsg list: List all messages
- rosmsg md5: Display message md5sum
- rosmsg package: List messages in a package
- rosmsg packages: List packages that contain messages

rosmsg (2)

The *type* command shows what type the service is.

Command:

```
$ rosmsg show sensor_msgs/CameraInfo
```

Output:

```
Header header
uint32 seq
time stamp
string frame_id
uint32 height
uint32 width
RegionOfInterest roi
uint32 x_offset
uint32 y_offset
uint32 height
uint32 width
```

Section 3

ROS Service

Service

- Node offers service under a certain string name
- Provide service with “ros::NodeHandle::advertiseService(string, callback)“ $\rightarrow$ provide a service name and a callback-function
- A client can make a persistent connection to a service, which enhances the performance
- “rosservice“: tool for seeing a list of available services and making queries
- “rossrv“: tool, which displays information about .srv data structures

Create service (1)

- Create .srv-file
- Structure for request and reply (similar to a remote procedure call)

```
string requeststring
---
string responsestring
```

- Service type similar to a message: package name + '/' + the name of the .srv file
  Example: "my_srvs/srv/MyService.srv" results in the service type "my_srvs/MyService"

Create service (2)

Same process as with messages
In the **CMakeLists.txt**:

- Additionally add the created .svg-files to the add_service_files() call to declare the service files to be built

Service: Generated Structure

- Roscpp generates a structure with C++-Classes for a service:

```
namespace my_package {
        struct Foo {
                class Request{ ... };
                class Response{ ... };
                Request request;
                Response response;
        };
}
```

Service: Calling a service

- Call-method from the “ros::service“ namespace
  “foo“ is the request/response structure

```
my_package::Foo foo;
if (ros::service::call("my_service_name", foo)) { ... }
```

- With node handle
  Enable a persistent connection with second parameter

```
ros::ServiceClient client =
nh.serviceClient<my_package::Foo>("my_service_name",true);
my_package::Foo foo;
if (client.call(foo)) { ... }
```

rosservice (1)

rosservice is a commandline based debug tool which allows for an easy testing and debugging of ROS services, e.g. a service can be manually called from the command line.

Available sub-commands:

- rosservice list: *print information about active services*
- rosservice call: *call the service with the provided args*
- rosservice type: *print service type*
- rosservice find: *find services by service type*
- rosservice uri: *print service ROSRPC uri*

rosservice (2)

Example: A given turtle demo was used for the following example. The *list* command shows the services from the turtlesim node demo.

Command:

```
$ rosservice list
```

Output:

```
/clear
/kill
/reset
/rosout/get_loggers
/rosout/set_logger_level
/spawn
/teleop_turtle/get_loggers
/teleop_turtle/set_logger_level
/turtle1/set_pen
...
```

rosservice (3)

Example: The *type* command shows what type the service is.

Command:

```
$ rosservice type clear
```

Output:

```
std_srvs/Empty
```

This service is empty, this means when the service call is made it takes no arguments (i.e. it sends no data when making a request and receives no data when receiving a response).

rosservice (4)

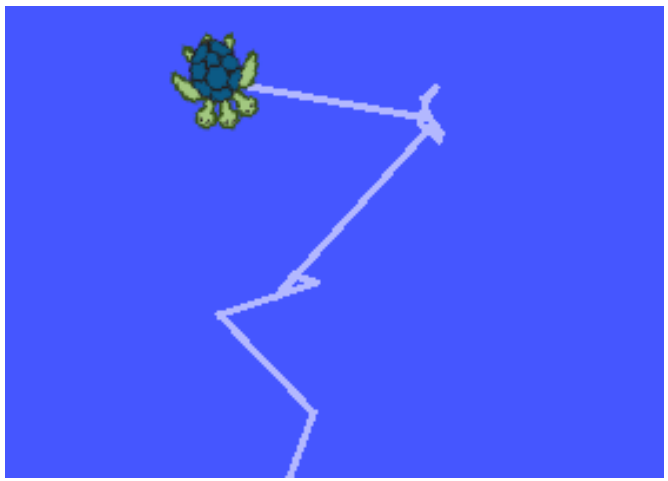
Example: The *call* command perform the used service. **Command:**

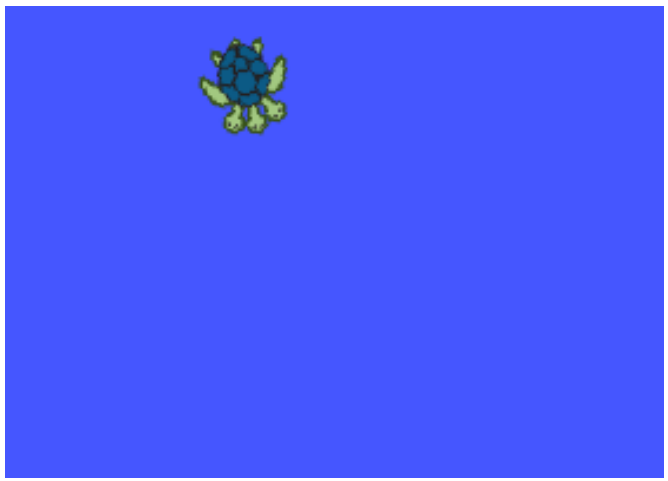
```
$ rosservice call clear
```

There are no further arguments because the service is of type empty. Otherwise the usage is:

```
$ rosservice call [service] [args]
```

This does what expected, it clears the background of the turtlesim_node.





rossrv

rossrv is a command-line based debugging tool for displaying information about ROS Service types.

Available sub-commands for rossrv:

- rossrv show: Show service description
- rossrv list: List all services
- rossrv md5: Display service md5sum
- rossrv package: List services in a package
- rossrv packages: List packages that contain services

The End