CSC752 Autonomous Robotic Systems
- Introduction into ROS (4) -

*Ubbo Visser*

Department of Computer Science
College of Arts and Sciences
University of Miami

September 2020
OVERVIEW

- ROS services
- ROS actions (actionlib)
- ROS time
- ROS bags
- Debugging strategies
- Request/response communication between nodes is realized with services
- The *service server* advertises the service
- The *service client* accesses this service
- Similar in structure to messages, services are defined in *.*rv files

**List services:**

```
~$ rosservice list
```

**Show service type:**

```
~$ rosservice type /service_name
```

**Call service requesting content:**

```
~$ rosservice call /service_name args
```
ROS SERVICE EXAMPLE

**std_srvs/Trigger.srv**

```c
---
bool success  # indicate successful run of triggered service
string message  # informational, e.g. for error messages
```

**nav_msgs/GetPlan.srv**

```c
# Get a plan from the current position to the goal Pose
# The start pose for the plan
geometry_msgs/PoseStamped start

# The final pose of the goal position
geometry_msgs/PoseStamped goal

# If the goal is obstructed, how many meters the planner can
# relax the constraint in x and y before failing.
float32 tolerance
---
nav_msgs/Path plan
```
ROS SERVICE EXAMPLE

T1: run roscore
~$roscore

T2: run a ros service
~$rosrun roscpp_tutorials add_two_ints_server

T3: list services
~$rosservice list

T3: call service
~$rosservice call /add_two_ints 25 12
- Create a service server:

```cpp
class AddTwoIntsResponse:
    int sum

bool add(const AddTwoIntsRequest &req, AddTwoIntsResponse &res)
```

- When a service request is received, the callback function is called with the request as argument

- Fill in the response to the response argument

- Return to function with true to indicate that it has been executed properly

Details at: [http://wiki.ros.org/roscpp/Overview/Services](http://wiki.ros.org/roscpp/Overview/Services)
Create a service client:

```cpp
ros::ServiceClient client = nodeHandle.serviceClient<service_type>(service_name);
```

Create service request contents

```cpp
service.request
```

Call service with

```cpp
client.call(service)
```

Response in

```cpp
service.response
```

Details at: http://wiki.ros.org/roscpp/Overview/Services
Similar to service calls, but provide possibility to
- Cancel the task (preempt)
- Receive feedback on the progress
- Best way to implement interfaces to time-extended, goal-oriented behaviors
- Similar in structure to services, action are defined in *.action files
- Internally, actions are implemented with a set of topics

Details at:
http://wiki.ros.org/actionlib
http://wiki.ros.org/actionlib/DetailedDescription
Averaging.action

#goal definition
int32 samples
---
#result definition
float32 mean
float32 std_dev
---
#feedback
int32 sample
float32 data
float32 mean
float32 std_dev

FollowPath.action

Goal

Result

Feedback

navigation_msgs/Path path
---
bool success
---
float32 remaining_distance
float32 initial_distance
<table>
<thead>
<tr>
<th>Description</th>
<th>Parameters</th>
<th>Dynamic reconfiguration</th>
<th>Topics</th>
<th>Services</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Global constant parameters</td>
<td>Local, changeable parameters</td>
<td>Continuous data streams</td>
<td>Blocking call for processing a request</td>
<td>Non-blocking, preempt-able goal oriented tasks</td>
</tr>
<tr>
<td>Application</td>
<td>Constant settings</td>
<td>Tuning parameters</td>
<td>One-way continuous data flow</td>
<td>Short triggers or calculations</td>
<td>Task execution and robot actions</td>
</tr>
<tr>
<td>Examples</td>
<td>Topic names, camera settings, calibration data, robot setup</td>
<td>Controller parameters</td>
<td>Sensor data, robot state</td>
<td>Trigger change, request state, compute quantity</td>
<td>Navigation, grasping, motion execution</td>
</tr>
</tbody>
</table>
ROS uses the PC’s system clock as time source (wall time)

For simulations or playback of logged data, it is convenient to work with a simulated time (pause, slow-down etc.)

To work with a simulated clock:

- Set the `/use_sim_time` parameter

  `$ rosparam set use_sim_time true`

- Publish the time on the topic `/clock` from
  - Gazebo (enabled by default)
  - ROS bag (use option `--clock`)

To take advantage of the simulated time, you should always use the ROS Time APIs:

- `ros::Time`

  ```
  ros::Time begin = ros::Time::now();
  double secs = begin.toSec();
  ```

- `ros::Duration`

  ```
  ros::Duration duration(0.5); // 0.5s
  ```

- `ros::Rate`

  ```
  ros::Rate rate(10); // 10Hz
  ```

- If wall time is required, use `ros::WallTime`, `ros::WallDuration`, and `ros::WallRate`

Details at:
- [http://wiki.ros.org/Clock](http://wiki.ros.org/Clock)
- [http://wiki.ros.org/roscpp/Overview/Time](http://wiki.ros.org/roscpp/Overview/Time)
- A bag is a format for storing message data
- Binary format with file extension *.bag
- Suited for logging and recording datasets for later visualization and analysis

**Record topics in a bag**

```
~$rosbag record --all
```

**Record specific topics**

```
~$rosbag record topic_1 topic_2 topic_3
```

Stop recording with Ctrl + C

Bags are saved with start date and time as file name in the current folder (e.g. 2020-09-17-12-58-14.bag)

**Show information about a bag**

```
~$rosbag info bag_name.bag
```

**Read a bag and play/publish its contents**

```
~$rosbag play bag_name.bag
```

Including playback options, e.g.

```
~$rosbag play --rate=0.5 bag_name.bag
```

- **--rate=factor**: Publish rate factor
- **--clock**: Publish clock time (set param use_sim_time to true)
- **--loop**: Look playback

**Details at:**

http://wiki.ros.org/Clock
http://wiki.ros.org/roscpp/Overview/Time
DEBUGGING STRATEGIES

- Compile and run code often to catch bugs early
- Understand compilation and runtime error messages
- Use analysis tools to check data flow (rosnode info, rostopic echo, roswtf, rqt_graph etc.)
- Visualize and plot data (RViz, RQT Multiplot etc.)
- Divide program into smaller steps and check intermediate results (ROS_INFO, ROS_DEBUG etc.)
- Make your code robust with argument and return value checks and catch exceptions
- If things don’t make sense, clean your workspace

- Use breakpoints if you use an IDE
- Maintain code with unit tests and integration tests

Build with DEBUG mode and use GDB and/or Valgrind

~$catkin config --cmake-args -DCMAKE_BUILD_TYPE=Debug

Details at:
ROS UnitTests
http://wiki.ros.org/gtest
http://wiki.ros.org/rostest
http://wiki.ros.org/roslaunch/Tutorials/
Roslaunch%20Nodes%20in%20Valgrind%20or%20GDB
FURTHER REFERENCES

- **ROS Wiki**
  - http://wiki.ros.org/
- **Installation**
  - http://wiki.ros.org/ROS/Installation
- **Tutorials**
  - http://wiki.ros.org/ROS/Tutorials
- **Packages**
  - https://www.ros.org/browse/list.php
- **ROS Cheat Sheet**
  - https://kapeli.com/cheat_sheets/ROS.docset/Contents/Resources/Documents/index
- **ROS Best Practices**
  - https://github.com/leggedrobotics/ros_best_practices/wiki
- **ROS Package Templates**
  - https://github.com/leggedrobotics/ros_best_practices/tree/master/ros_package_template
Material is based on ROS Wiki and ETH Zürich ROS Introduction (https://rsl.ethz.ch/)