CSC398 Autonomous Robotic Systems
- Introduction into ROS (4) -

Ubbo Visser

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

September 2024

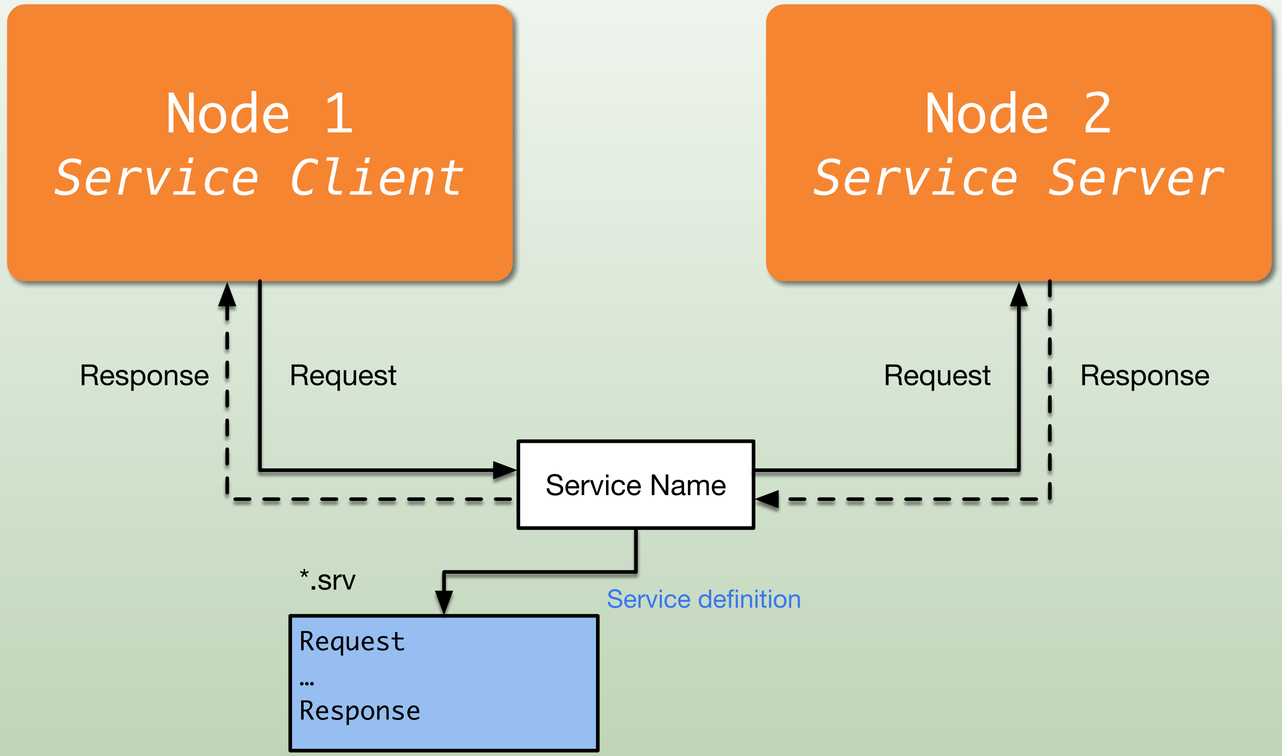
UNIVERSITY OF MIAMI
ROBOCANES

OVERVIEW

- ROS services
- ROS actions (actionlib)
- ROS time
- ROS bags
- Debugging strategies

ROS SERVICES

- 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 **.srv* files



List services:

```
~$rosservice list
```

Show service type:

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

Call service requesting content:

```
~$rosservice call /service_name args
```

Details at: http://wiki.ros.org/Services

ROS SERVICE EXAMPLE

std_srvs/Trigger.srv

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

Request

Response

nav_msgs/GetPlan.srv

Request

Response

```
# 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

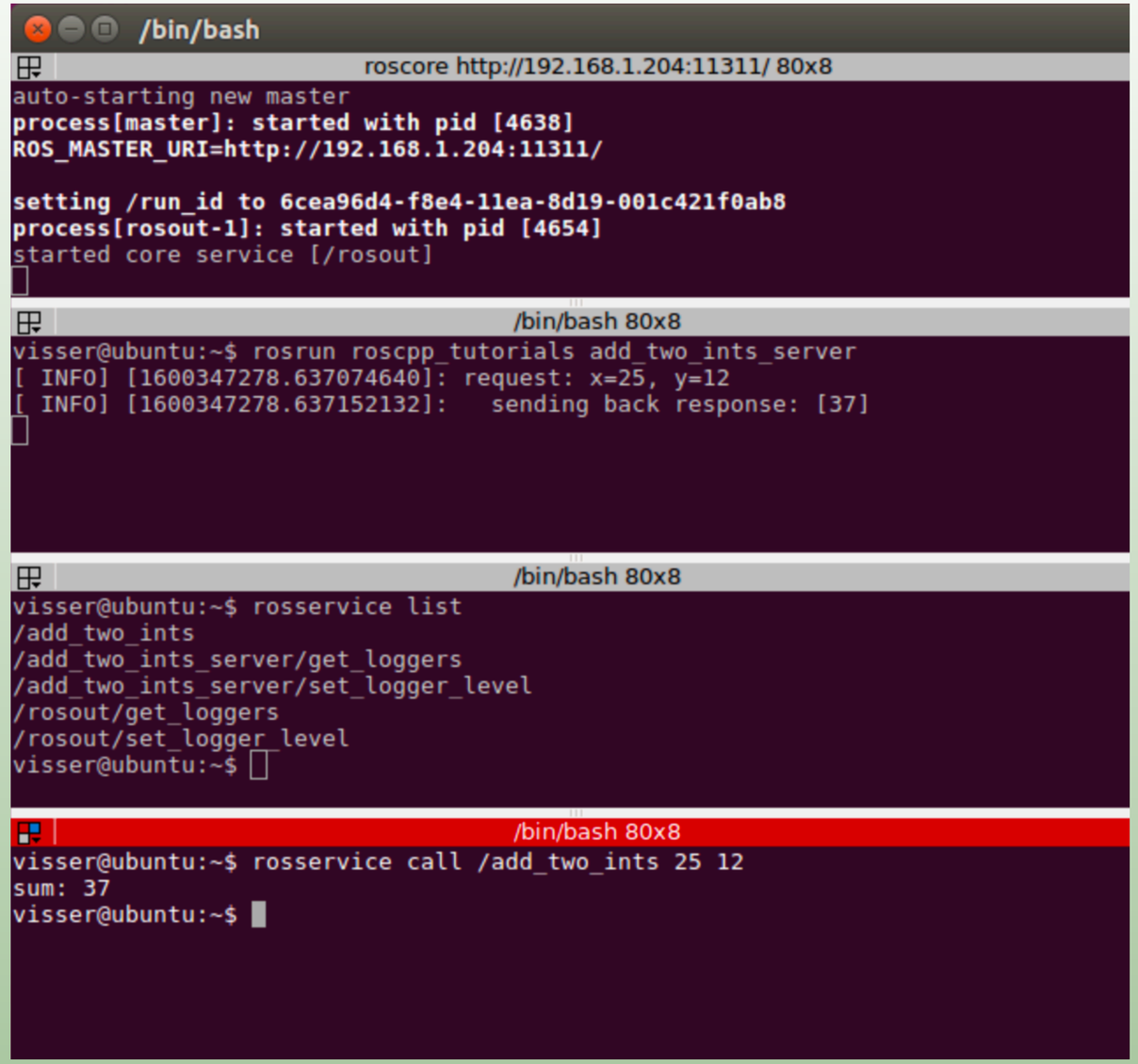
```
~$rosrun roscpp_tutorials add_two_ints_server
```

T3: list services

```
~$rosservice list
```

T3: call service

```
~$rosservice call /add_two_ints 25 12
```



ROS C++ CLIENT LIBRARY - SERVICE SERVER

- Create a service server:

```
ros::ServiceServer service =
    nodeHandle.advertiseService(service_name,
    callback_function);
```

- 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

```
#include "ros/ros.h"
#include "beginner_tutorials/AddTwoInts.h"

bool add(beginner_tutorials::AddTwoInts::Request  &req,
         beginner_tutorials::AddTwoInts::Response &res)
{
  res.sum = 1 << (req.a + req.b);
  ROS_INFO("Request: x = %ld, y = %ld", (long int)req.a, (long int)req.b);
  ROS_INFO("Sending response: [%ld]", (long int)res.sum);
  return true;
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "add_two_ints_server");
  ros::NodeHandle n;

  ros::ServiceServer service = n.advertiseService("add_two_ints", add);
  ROS_INFO("Ready to add two ints.");
  ros::spin();

  return 0;
}
```

Details at: http://wiki.ros.org/roscpp/Overview/Services

ROS C++ CLIENT LIBRARY - SERVICE CLIENT

- Create a service client:

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

- Create service request contents

```
service.request
```

- Call service with

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

- Response in

```
service.response
```

Details at: http://wiki.ros.org/roscpp/Overview/Services

```
#include "ros/ros.h"
#include "beginner_tutorials/AddTwoInts.h"
#include <cstdlib>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "add_two_ints_client");
  if (argc != 3)
  {
    ROS_INFO("usage: add_two_ints_client X Y");
    return 1;
  }

  ros::NodeHandle n;
  ros::ServiceClient client =
       n.serviceClient<beginner_tutorials::AddTwoInts>("add_two_ints");
  beginner_tutorials::AddTwoInts srv;
  srv.request.a = atoll(argv[1]);
  srv.request.b = atoll(argv[2]);
  if (client.call(srv))
  {
    ROS_INFO("Sum: %ld", (long int)srv.response.sum);
  }
  else
  {
    ROS_ERROR("Failed to call service add_two_ints");
    return 1;
  }

  return 0;
}
```

ROSPY CLIENT LIBRARY - SERVICE SERVER

- Create a service server:

```
s = rospy.Service('add_two_ints', AddTwoInts, add_two_ints)
```

- 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

```
NAME = 'add_two_ints_server'

# import the AddTwoInts service
from rospy_tutorials.srv import *
import rospy

def add_two_ints(req):
    print("Returning [%s + %s = %s]" % (req.a, req.b, (req.a + req.b)))
    return AddTwoIntsResponse(req.a + req.b)

def add_two_ints_server():
    rospy.init_node(NAME)
    s = rospy.Service('add_two_ints', AddTwoInts, add_two_ints)

    # spin() keeps Python from exiting until node is shutdown
    rospy.spin()

if __name__ == "__main__":
    add_two_ints_server()
```

Details at: http://wiki.ros.org/rospy/Overview/Services

ROSCP CLIENT LIBRARY - SERVICE CLIENT

- Service proxies

```
add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
```

- Call service with

```
# simplified style
resp1 = add_two_ints(x, y)

# formal style
resp2 = add_two_ints.call(AddTwoIntsRequest(x, y))
```

- Wait for service good practice

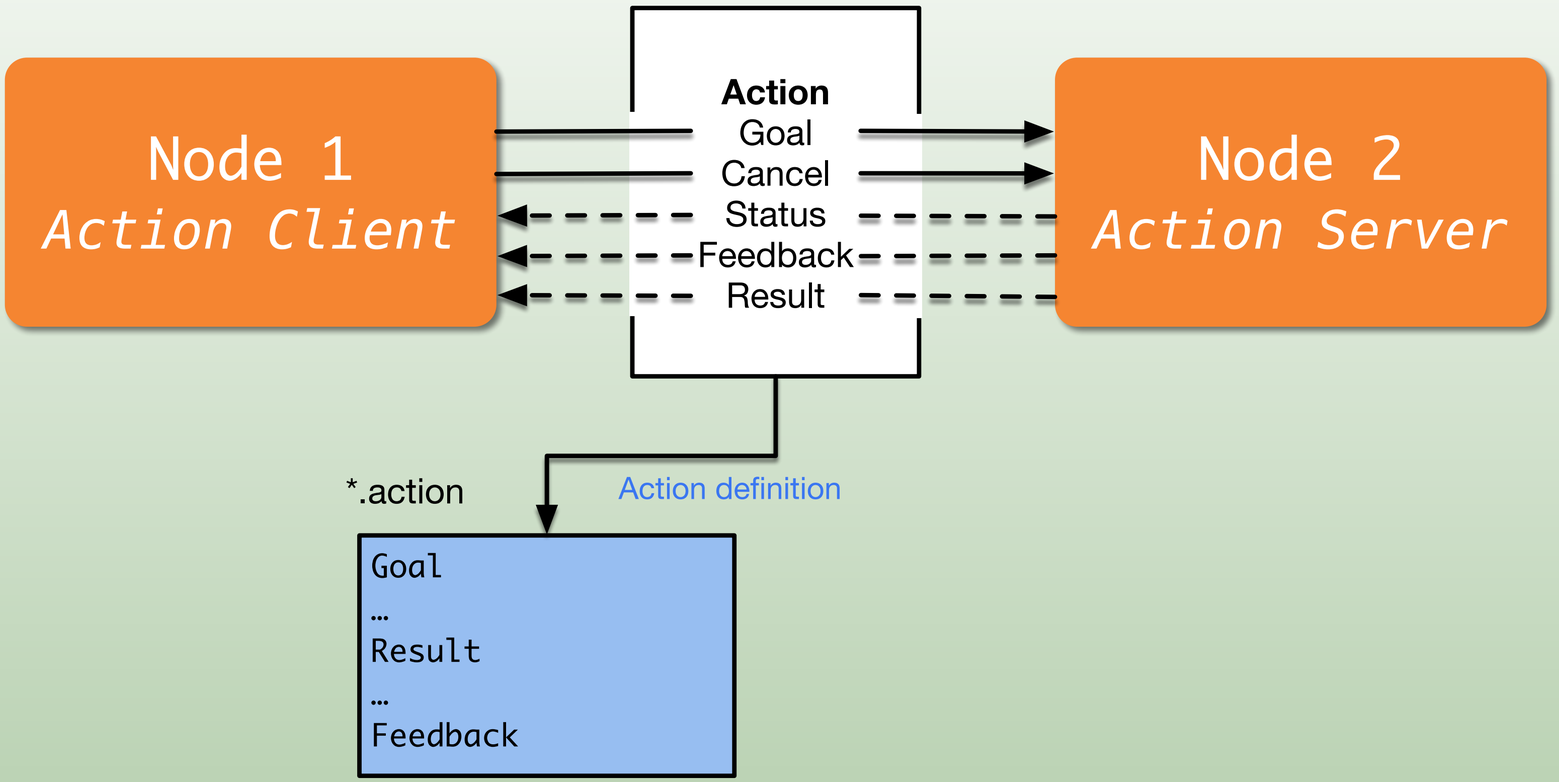
```
rospy.wait_for_service('add_two_ints')
```

Details at: http://wiki.ros.org/rospy/Overview/Services

```
import rospy

# imports the AddTwoInts service
from rospy_tutorials.srv import *

## add two numbers using the add_two_ints service
## @param x int: first number to add
## @param y int: second number to add
def add_two_ints_client(x, y):

    # NOTE: you don't have to call rospy.init_node() to make calls against
    # a service. This is because service clients do not have to be
    # nodes.

    # block until the add_two_ints service is available
    # you can optionally specify a timeout
    rospy.wait_for_service('add_two_ints')

    try:
        # create a handle to the add_two_ints service
        add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)

        print("Requesting %s+%s"%(x, y))

        # simplified style
        resp1 = add_two_ints(x, y)

        # formal style
        resp2 = add_two_ints.call(AddTwoIntsRequest(x, y))

        if not resp1.sum == (x + y):
            raise Exception("test failure, returned sum was %s"%resp1.sum)
        if not resp2.sum == (x + y):
            raise Exception("test failure, returned sum was %s"%resp2.sum)
        return resp1.sum
    except rospy.ServiceException as e:
        print("Service call failed: %s"%e)

def usage():
    return "%s [x y]"%sys.argv[0]      jfaustwg, 15 years ago • Create trunk/branche

if __name__ == "__main__":

    argv = rospy.myargv()
    if len(argv) == 1:
        import random
        x = random.randint(-50000, 50000)
        y = random.randint(-50000, 50000)
    elif len(argv) == 3:
        try:
            x = int(argv[1])
            y = int(argv[2])
        except:
            print(usage())
            sys.exit(1)
    else:
        print(usage())
        sys.exit(1)
    print("%s + %s = %s"%(x, y, add_two_ints_client(x, y)))
```

ROS ACTIONS (ACTIONLIB)

- 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, actions 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

ROS ACTIONS (ACTIONLIB)

Averaging.action

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

Goal

Result

Feedback

FollowPath.action

Goal

Result

Feedback

```
navigation_msgs/Path path
---
bool success
---
float32 remaining_distance
float32 initial_distance
```

COMPARISON OF OPTION

	Parameters	Dynamic reconfiguration	Topics	Services	Actions
Description	Global constant parameters	Local, changeable parameters	Continuous data streams	Blocking call for processing a request	Non-blocking, preempt-able goal oriented tasks
Application	Constant settings	Tuning parameters	One-way continuous data flow	Short triggers or calculations	Task execution and robot actions
Examples	Topic names, camera settings, calibration data, robot setup	Controller parameters	Sensor data, robot state	Trigger change, request state, compute quantity	Navigation, grasping, motion execution

ROS TIME

- 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/roscpp/Overview/Time

ROS BAGS

- 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. 2022-09-29-11-58-14.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, ros_gui, 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

```
~$catkin_make clean --all
```

- 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://www.clearpathrobotics.com/ros-robot-operating-system-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

ACKNOWLEDGEMENT

Material is based on ROS Wiki and ETH Zürich ROS Introduction (https://rsl.ethz.ch/)