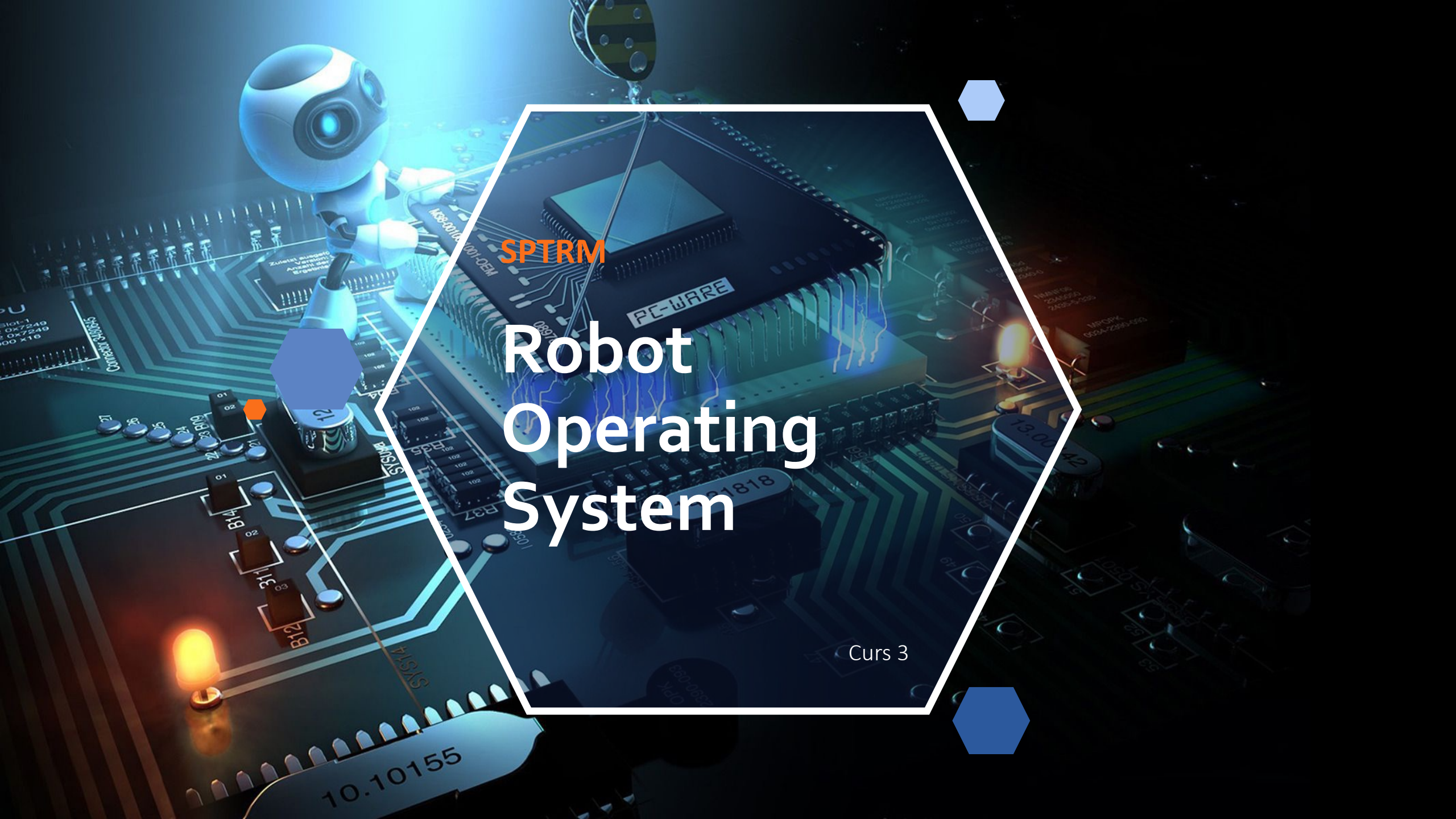


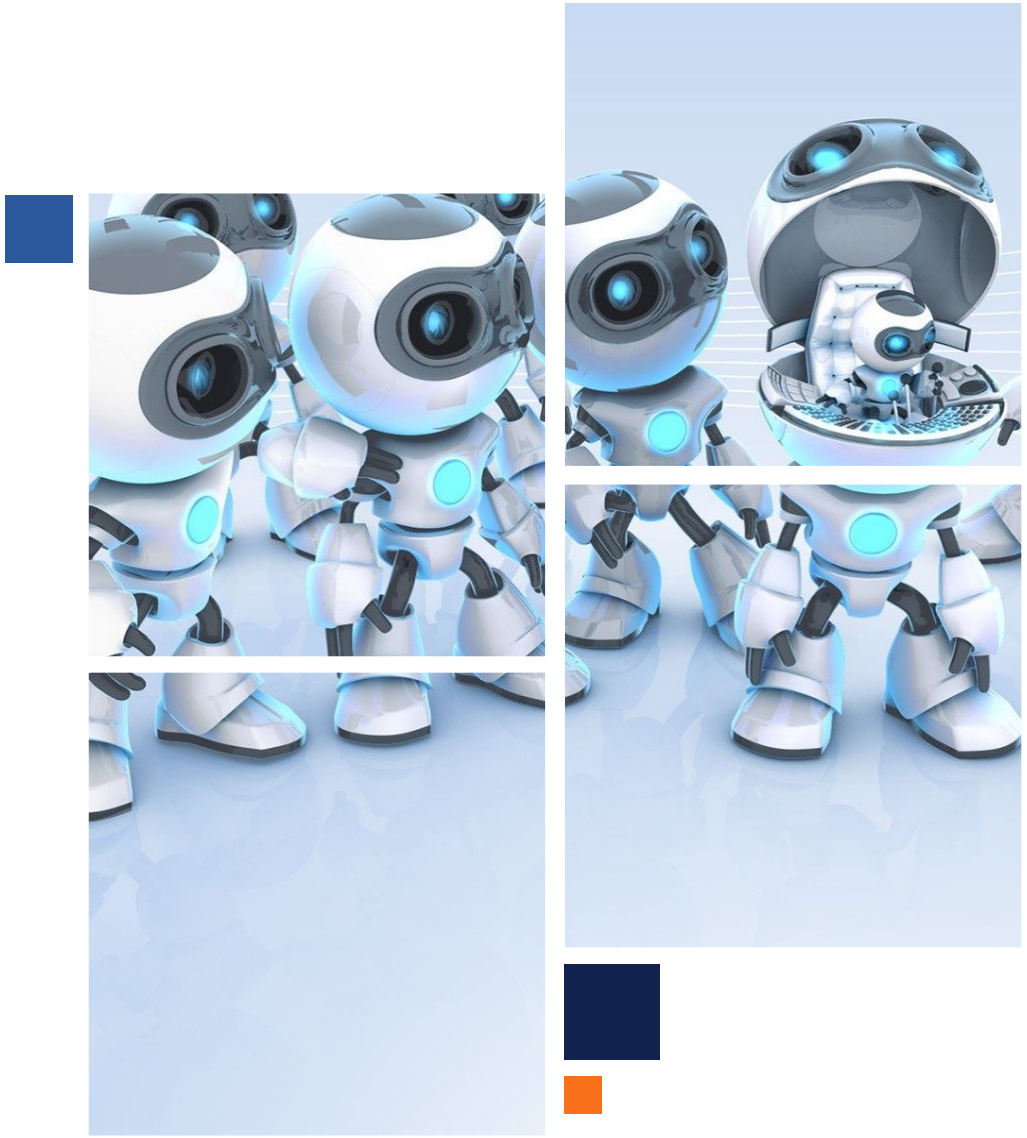
SPTRM

Robot Operating System

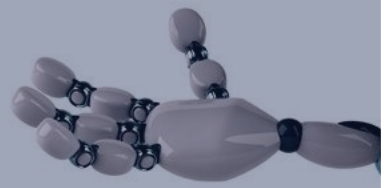
Curs 3

Agenda

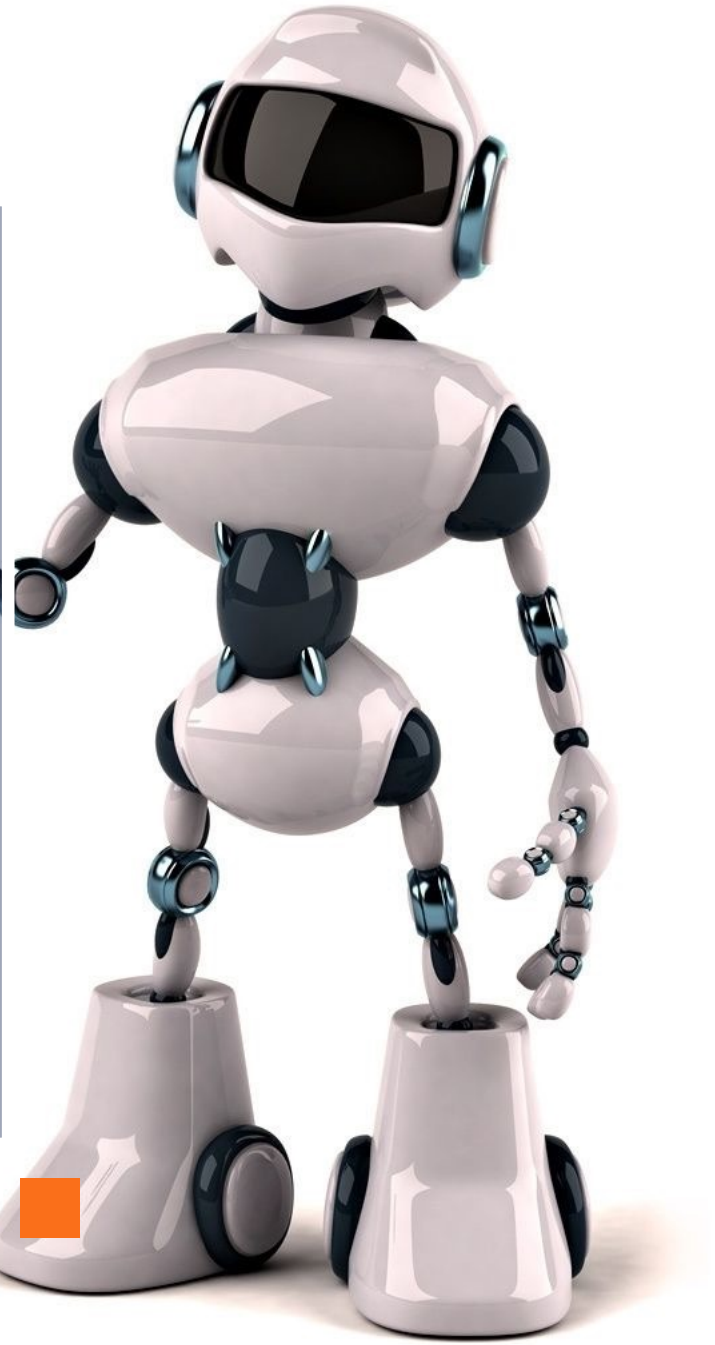
- Recap
- Communication in ROS continue
  - Services
  - Actions
- Time and bags
  - Time and Duration
  - Simulated Time
  - ROS bags
- Debugging in ROS



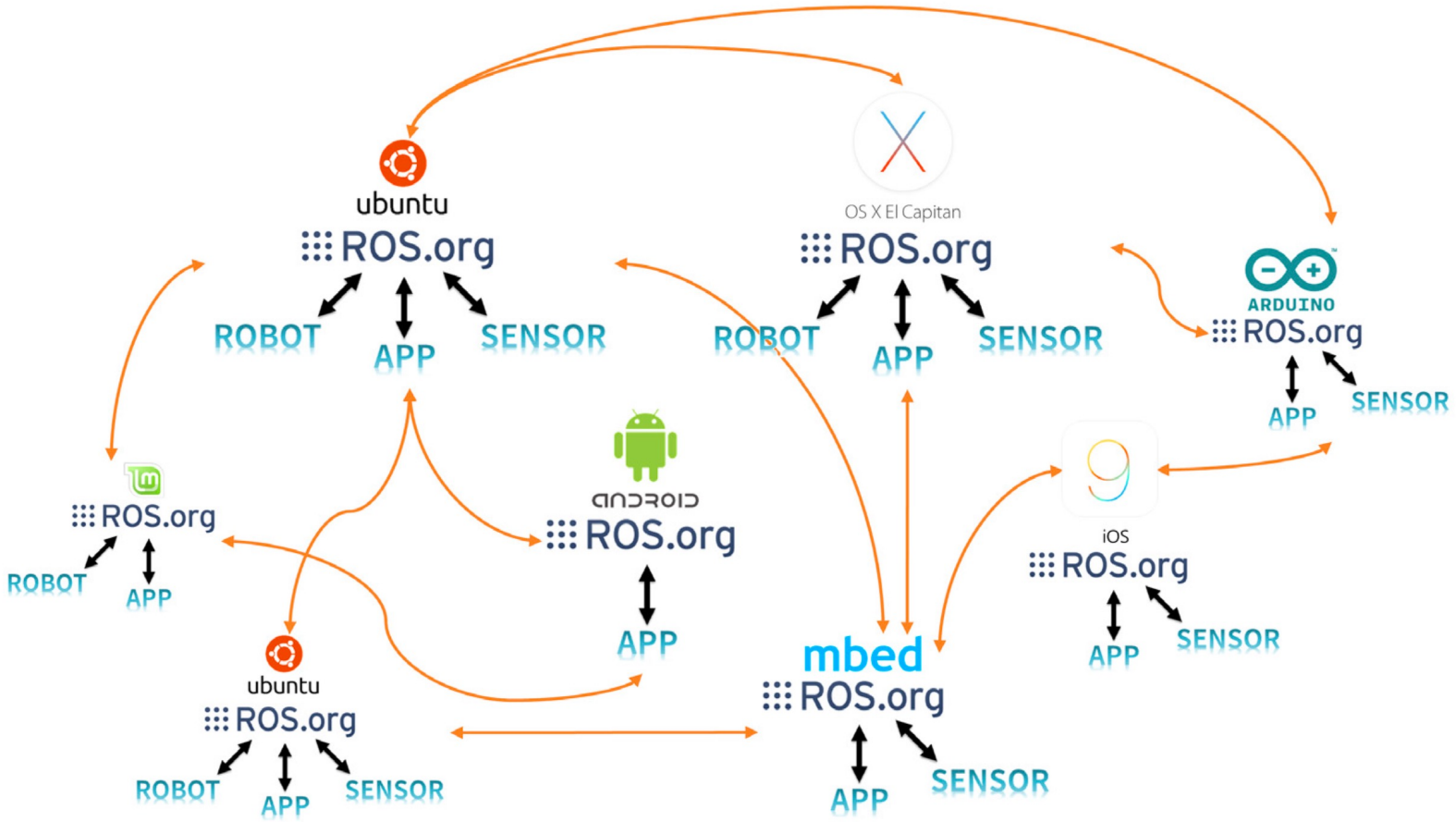
Recap



From the last episode…



Heterogenous communication in ROS



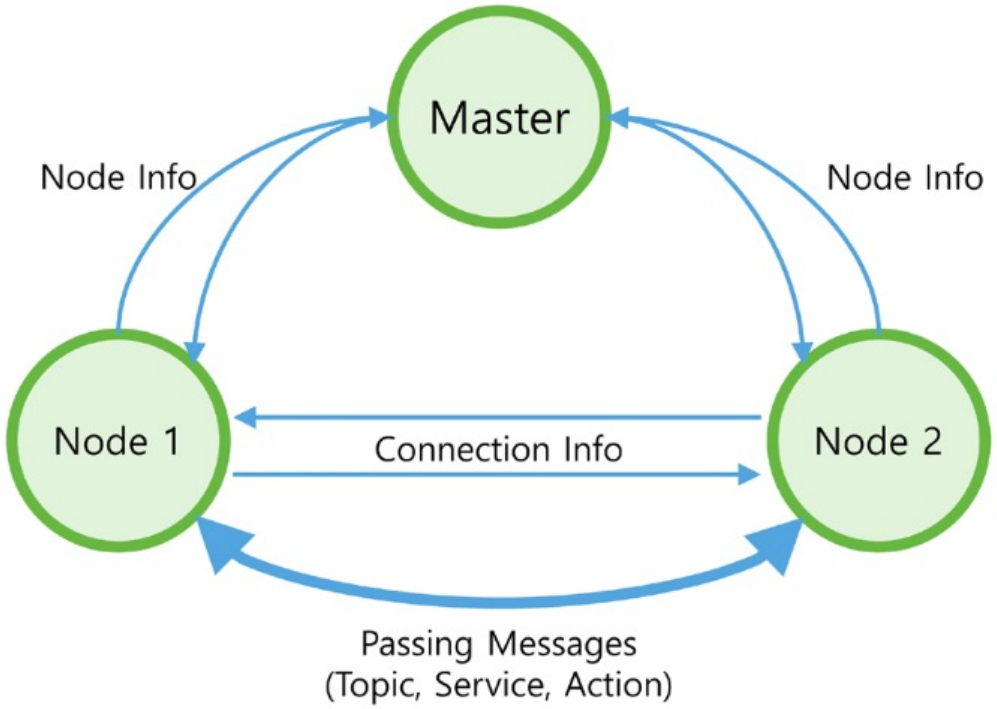
Communication in ROS

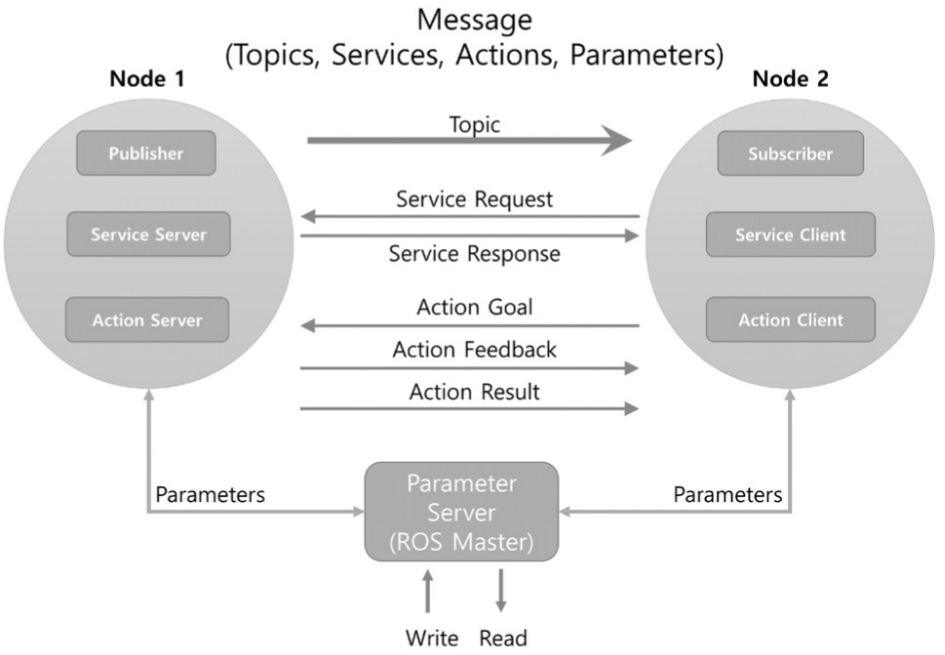
Type	Parameters	Topics	Services	Actions
Description	Global constant parameters	Continuous data streams	Blocking call for processing a request	Nonblocking preemptable goal oriented tasks
Application	Constant settings	One-way continuous data flow	Short triggers and calculations	Task executions and robot actions
Examples	Topic names, camera settings, calibration data, robot setup	Sensor data, robot state	Trigger change, request state, compute quantity	Navigation, grasping, motion execution

ROS Parameter Server

- Shared dictionary accessed via network APIs
- Nodes use this server to store and retrieve parameters at runtime.
- Not designed for high-performance, it is best used for static, non-binary data such as configuration parameters
- Globally viewable so that tools can easily inspect the configuration state of the system and modify if necessary.
- The Parameter Server is implemented using XMLRPC and runs inside of the ROS Master

ROS Communication





XMLRPC (XML-Remote Procedure Call)

RPC protocol

XML as the encoding format

a very simple protocol

very lightweight

supports multiple programming languages

TCPROS

TCPROS is a message format based on TCP/IP (UDPROS is a message format based on UDP)

TCPROS is a transport layer for ROS Messages and Services.

It uses standard TCP/IP sockets for transporting message data. Inbound connections are received via a TCP Server Socket with a header containing message data type and routing information.

Messages

```
fieldtype1 fieldname1
fieldtype2 fieldname2
fieldtype3 fieldname3
```

- Bundle of data to be exchanged between nodes
- Used by topics, services and actions
- Can include basic data type or complex types
- Inheritance
- Inclusion
- Defined in a *.msg file

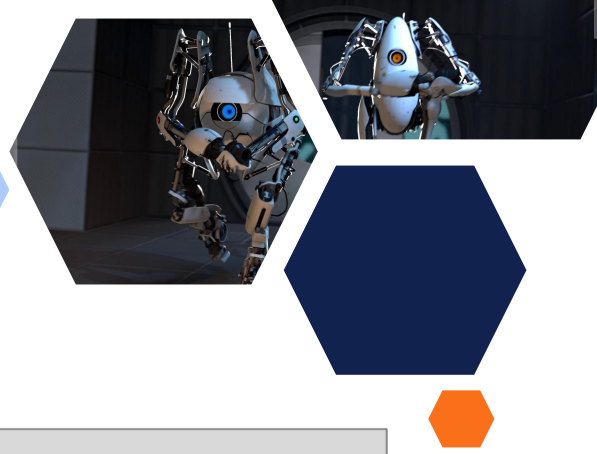
ROS C++ Client Library (roscpp)

hello_world.cpp

```
#include <ros/ros.h>

int main(int argc, char** argv)
{
  ros::init(argc, argv, "hello_world");
  ros::NodeHandle nodeHandle;
  ros::Rate loopRate(10);

  unsigned int count = 0;
  while (ros::ok()) {
    ROS_INFO_STREAM("Hello World " << count);
    ros::spinOnce();
    loopRate.sleep();
    count++;
  }

  return 0;
}
```

- ROS main header file include
- `ros::init(…)` has to be called before calling other ROS functions
- The node handle is the access point for communications with the ROS system (topics, services, parameters)
- `ros::Rate` is a helper class to run loops at a desired frequency
- `ros::ok()` checks if a node should continue running
  Returns `false` if `SIGINT` is received (`Ctrl + C`) or `ros::shutdown()` has been called
- `ROS_INFO()` logs messages to the filesystem
- `ros::spinOnce()` processes incoming messages via callbacks

More info
http://wiki.ros.org/roscpp
http://wiki.ros.org/roscpp/Overview



Publisher

- Create a publisher with help of the node handle

```
ros::Publisher publisher =
nodeHandle.advertise<message_type>(topic,
queue_size);
```

- Create the message contents
- Publish the contents with

```
publisher.publish(message);
```

```
#include <ros/ros.h>
#include <std_msgs/String.h>

int main(int argc, char **argv) {
  ros::init(argc, argv, "talker");
  ros::NodeHandle nh;
  ros::Publisher chatterPublisher =
    nh.advertise<std_msgs::String>("chatter", 1);
  ros::Rate loopRate(10);

  unsigned int count = 0;
  while (ros::ok()) {
    std_msgs::String message;
    message.data = "hello world " + std::to_string(count);
    ROS_INFO_STREAM(message.data);
    chatterPublisher.publish(message);
    ros::spinOnce();
    loopRate.sleep();
    count++;
  }
  return 0;
}
```

More info
http://wiki.ros.org/roscpp/Overview/Publishers%20and%20Subscribers

Subscriber

- Start listening to a topic by calling the method `subscribe()` of the node handle

```
ros::Subscriber subscriber =
nodeHandle.subscribe(topic, queue_size,
                     callback_function);
```

- When a message is received, callback function is called with the contents of the message as argument
- Hold on to the subscriber object until you want to unsubscribe

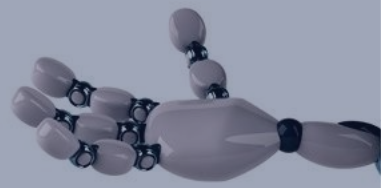
`ros::spin()` processes callbacks and will not return until the node has been shutdown

listener.cpp

```
#include "ros/ros.h"
#include "std_msgs/String.h"

void chatterCallback(const std_msgs::String& msg)
{
  ROS_INFO("I heard: [%s]", msg.data.c_str());
}

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

  ros::Subscriber subscriber =
       nodeHandle.subscribe("chatter",10,chatterCallback);
  ros::spin();
  return 0;
}
```

More info
http://wiki.ros.org/roscpp/Overview/Publishers%20and%20Subscribers

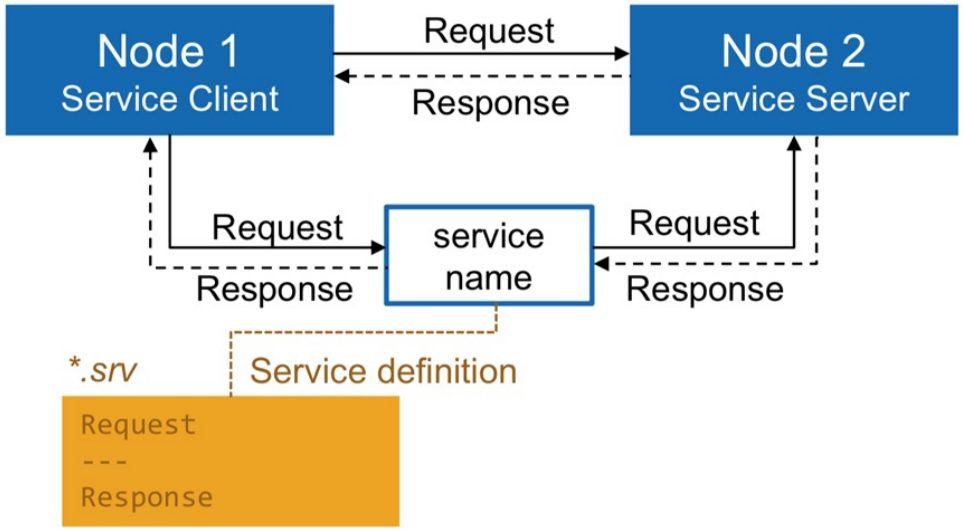
Communication in ROS



ROS Services. ROS Actions. Examples

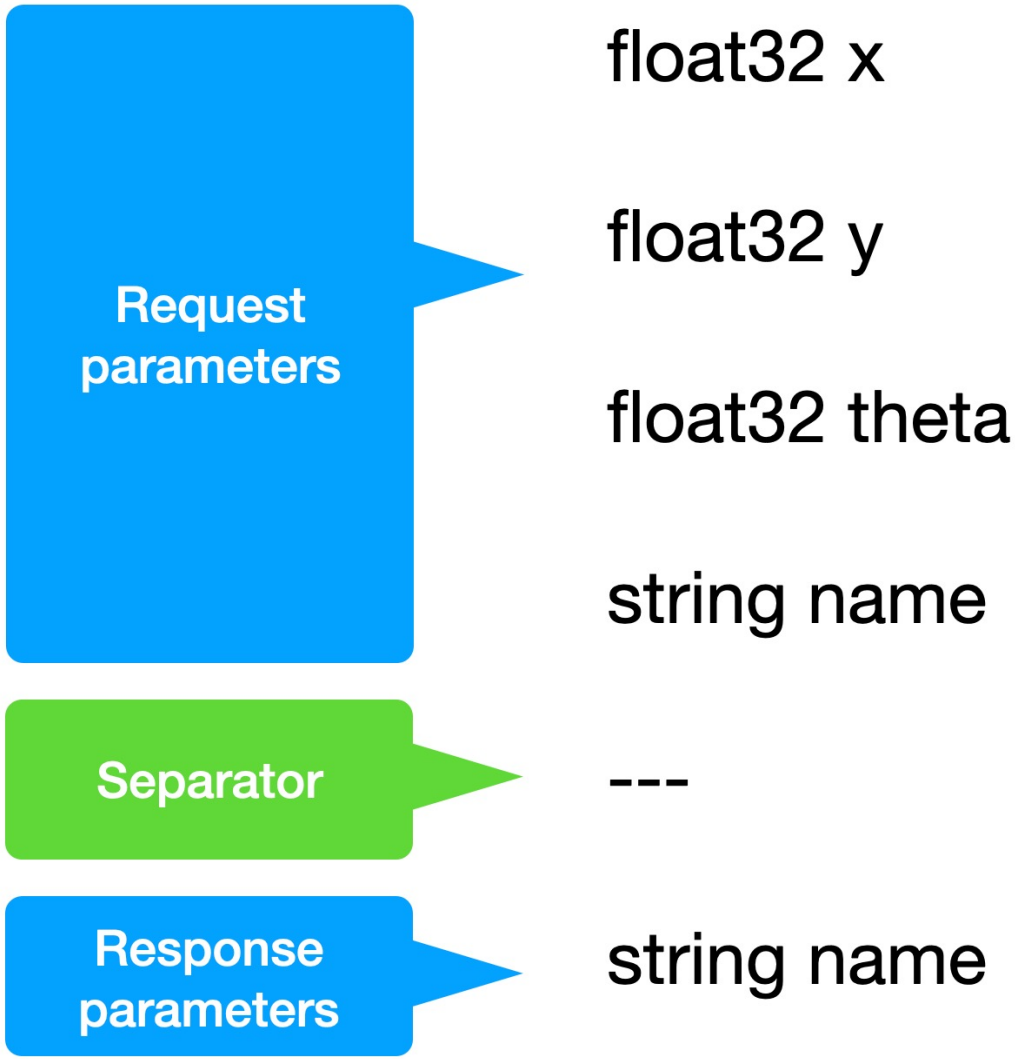
ROS Services

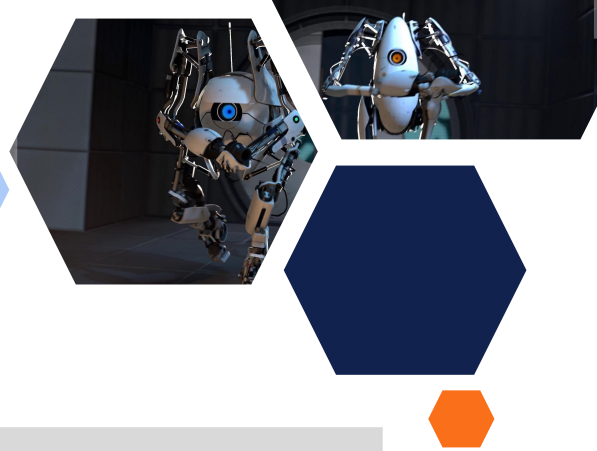
- Services defined in *.srv files
- 2 sections:
  - Request parameters
  - Response parameters
- Similar to ROS messages
- Can contain zero or multiple entries per section (simple date or complex types)



Example

rossrv show turtlesim/Spawn





Service Server

- When a service request is received, 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

- SRV file: TwoInts.srv

```
int64 a
int64 b
---
int64 sum
```

```
#include <ros/ros.h>
#include <roscpp_tutorials/TwoInts.h>

bool add(roscpp_tutorials::TwoInts::Request &request,
         roscpp_tutorials::TwoInts::Response &response)
{
  response.sum = request.a + request.b;
  ROS_INFO("request: x=%ld, y=%ld", (long int)request.a,
                                    (long int)request.b);
  ROS_INFO("  sending back response: [%ld]",
           (long int)response.sum);
  return true;
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "add_two_ints_server");
  ros::NodeHandle nh;
  ros::ServiceServer service =
           nh.advertiseService("add_two_ints", add);
  ros::spin();
  return 0;
}
```

Service Client

- Create a service client
- Create service request contents
- Call service
- Response is stored in service.response

```
#include <ros/ros.h>
#include <roscpp_tutorials/TwoInts.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 nh;
  ros::ServiceClient client =
  nh.serviceClient<roscpp_tutorials::TwoInts>("add_two_ints");
  roscpp_tutorials::TwoInts service;
  service.request.a = atoi(argv[1]);
  service.request.b = atoi(argv[2]);
  if (client.call(service)) {
    ROS_INFO("Sum: %ld", (long int)service.response.sum);
  } else {
    ROS_ERROR("Failed to call service add_two_ints");
    return 1;
  }
  return 0;
}
```

Demo

rosrun roscpp_tutorials add_two_ints_server

rosservice list

```
student@ubuntu:~$ rosservice list
/add_two_ints
/add_two_ints_server/get_loggers
/add_two_ints_server/set_logger_level
/rosout/get_loggers
/rosout/set_logger_level
```

rosservice type /add_two_ints

```
student@ubuntu:~$ rosservice type /add_two_ints
roscpp_tutorials/TwoInts
```

rossrv show roscpp_tutorials/TwoInts

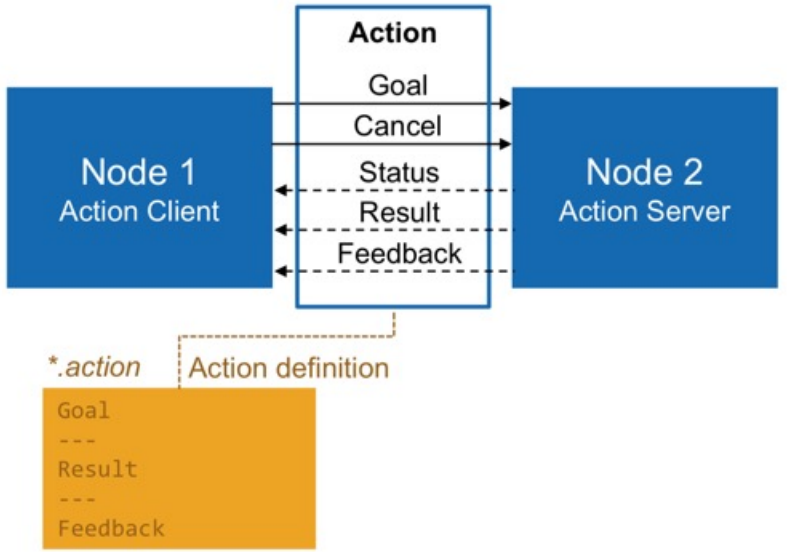
```
student@ubuntu:~$ rossrv show roscpp_tutorials/TwoInts
int64 a
int64 b
---
int64 sum
```

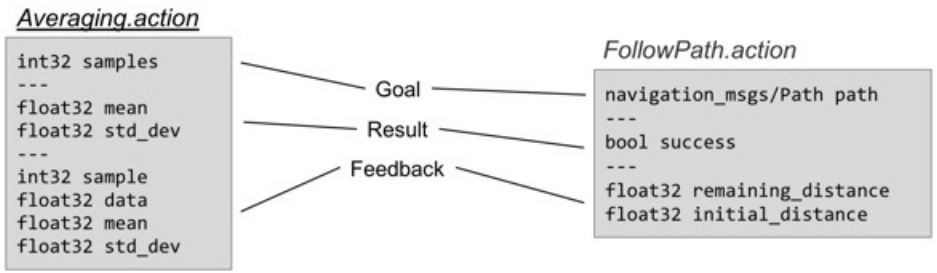
rosservice call /add_two_ints "a: 10 b: 5"

```
student@ubuntu:~$ rosservice call /add_two_ints "a: 10
b: 5"
sum: 15
```

ROS Actions (actionlib)

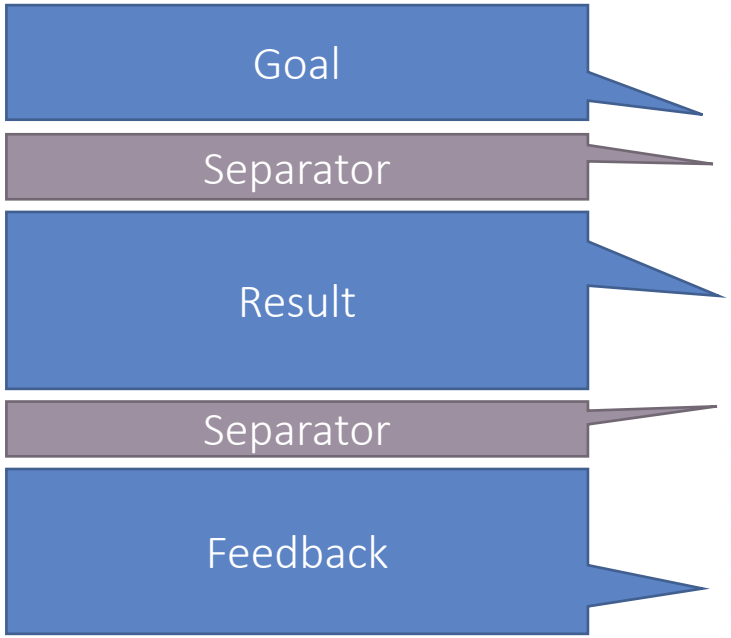
- Similar to service calls
- Allows task cancelling (preempt) and progress feedback
- Best way to implement interfaces to time-extended, goal-oriented behaviors
- Similar in structure to services, action are defined in *.action files
- Internally, actions are implemented with a set of topics





1. Define action file

- Define an action that acts like a timer
- Periodically reports how much time is left
- Signals when the specified time elapsed
- Timer.action




```
# Part 1: the goal, to be sent by the client
#
# The amount of time we want to wait
duration time_to_wait
---
# Part 2: the result, to be sent by the server upon completion
#
# How much time we waited
duration time_elapsed
# How many updates we provided along the way
uint32 updates_sent
---
# Part 3: the feedback, to be sent periodically by the server during
# execution.
#
# The amount of time that has elapsed from the start
duration time_elapsed
# The amount of time remaining until we're done
duration time_remaining
```

2. Implement action server

- Define the action server
- Left – simple version with no feedback
- Right – sends also feedback

```python
#! /usr/bin/env python
import rospy

import time
import actionlib
from basics.msg import TimerAction, TimerGoal, TimerResult

def do_timer(goal):
    start_time = time.time()
    time.sleep(goal.time_to_wait.to_sec())
    result = TimerResult()
    result.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
    result.updates_sent = 0
    server.set_succeeded(result)

rospy.init_node('timer_action_server')
server = actionlib.SimpleActionServer('timer', TimerAction, do_timer, False)
server.start()
rospy.spin()
```

```python
#! /usr/bin/env python
import rospy

import time
import actionlib
from basics.msg import TimerAction, TimerGoal, TimerResult, TimerFeedback

def do_timer(goal):
    start_time = time.time()
    update_count = 0

    if goal.time_to_wait.to_sec() > 60.0:
        result = TimerResult()
        result.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
        result.updates_sent = update_count
        server.set_aborted(result, "Timer aborted due to too-long wait")
        return

    while (time.time() - start_time) < goal.time_to_wait.to_sec():

        if server.is_preempt_requested():
            result = TimerResult()
            result.time_elapsed = \
                rospy.Duration.from_sec(time.time() - start_time)
            result.updates_sent = update_count
            server.set_preempted(result, "Timer preempted")
            return

        feedback = TimerFeedback()
        feedback.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
        feedback.time_remaining = goal.time_to_wait - feedback.time_elapsed
        server.publish_feedback(feedback)
        update_count += 1

        time.sleep(1.0)

    result = TimerResult()
    result.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
    result.updates_sent = update_count
    server.set_succeeded(result, "Timer completed successfully")

rospy.init_node('timer_action_server')
server = actionlib.SimpleActionServer('timer', TimerAction, do_timer, False)
server.start()
rospy.spin()
```

3. Implement action client

- Define the action server
- Left – simple version with no feedback
- Right – receives also feedback

```python
#! /usr/bin/env python
import rospy

import actionlib
from basics.msg import TimerAction, TimerGoal, TimerResult

rospy.init_node('timer_action_client')
client = actionlib.SimpleActionClient('timer', TimerAction)
client.wait_for_server()
goal = TimerGoal()
goal.time_to_wait = rospy.Duration.from_sec(5.0)
client.send_goal(goal)
client.wait_for_result()
print('Time elapsed: %f'%(client.get_result().time_elapsed.to_sec()))
```

```python
#! /usr/bin/env python
import rospy

import time
import actionlib
from basics.msg import TimerAction, TimerGoal, TimerResult, TimerFeedback

def feedback_cb(feedback):
    print('[Feedback] Time elapsed: %f'%(feedback.time_elapsed.to_sec()))
    print('[Feedback] Time remaining: %f'%(feedback.time_remaining.to_sec()))

rospy.init_node('timer_action_client')
client = actionlib.SimpleActionClient('timer', TimerAction)
client.wait_for_server()

goal = TimerGoal()
goal.time_to_wait = rospy.Duration.from_sec(5.0)
# Uncomment this line to test server-side abort:
#goal.time_to_wait = rospy.Duration.from_sec(500.0)
client.send_goal(goal, feedback_cb=feedback_cb)

# Uncomment these lines to test goal preemption:
#time.sleep(3.0)
#client.cancel_goal()

client.wait_for_result()
print('[Result] State: %d'%(client.get_state()))
print('[Result] Status: %s'%(client.get_goal_status_text()))
print('[Result] Time elapsed: %f'%(client.get_result().time_elapsed.to_sec()))
print('[Result] Updates sent: %d'%(client.get_result().updates_sent))
```

4. Start action server

- rosrun basics simple_action_server.py

```
user@hostname$ rostopic list
/rosout
/rosout_agg
/timer/cancel
/timer/feedback
/timer/goal
/timer/result
/timer/status
```

```
user@hostname$ rostopic info /timer/goal
Type: basics/TimerActionGoal

Publishers: None

Subscribers:
 * /timer_action_server (http://localhost:63174/)
```

```
user@hostname$ rosmsg show TimerActionGoal
[basics/TimerActionGoal]:
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
actionlib_msgs/GoalID goal_id
  time stamp
  string id
basics/TimerGoal goal
  duration time_to_wait
```

```
user@hostname$ rosmsg show TimerGoal
[basics/TimerGoal]:
duration time_to_wait
```

5. Run action client

- Test multiple scenarios
  - Basic client
  - Advanced client with feedback
  - Advanced client with cancel request

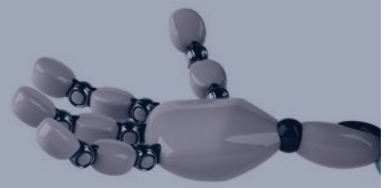
```
user@hostname$ rosrun basics simple_action_client.py
Time elapsed: 5.001044
```

```
user@hostname$ rosrun basics fancy_action_client.py
[Feedback] Time elapsed: 0.000044
[Feedback] Time remaining: 4.999956
[Feedback] Time elapsed: 1.001626
[Feedback] Time remaining: 3.998374
[Feedback] Time elapsed: 2.003189
[Feedback] Time remaining: 2.996811
[Feedback] Time elapsed: 3.004825
[Feedback] Time remaining: 1.995175
[Feedback] Time elapsed: 4.006477
[Feedback] Time remaining: 0.993523
[Result] State: 3
[Result] Status: Timer completed successfully
[Result] Time elapsed: 5.008076
[Result] Updates sent: 5
```

```
user@hostname$ rosrun basics fancy_action_client.py
[Result] State: 4
[Result] Status: Timer aborted due to too-long wait
[Result] Time elapsed: 0.000012
[Result] Updates sent: 0
```

```
user@hostname$ rosrun basics fancy_action_client.py
[Feedback] Time elapsed: 0.000044
[Feedback] Time remaining: 4.999956
[Feedback] Time elapsed: 1.001651
[Feedback] Time remaining: 3.998349
[Feedback] Time elapsed: 2.003297
[Feedback] Time remaining: 2.996703
[Result] State: 2
[Result] Status: Timer preempted
[Result] Time elapsed: 3.004926
[Result] Updates sent: 3
```

Time and bags



ROS Time. ROS Duration. ROS Rate. Simulation Time. Recording and replaying messages

ROS Time and Duration

- Normally, ROS uses the PC's system clock as time source (**wall time**)
- APIs:

  ros::Time, ros::Duration

  ros::WallTime, ros::WallDuration
- Message definition

```
int32 sec
int32 nsec
```

- Get current time

  ros::Time begin = ros::Time::now();

- Arithmetics

  1 hour + 1 hour = 2 hours (*duration + duration = duration*)
  2 hours - 1 hour = 1 hour (*duration - duration = duration*)
  Today + 1 day = tomorrow (*time + duration = time*)
  Today - tomorrow = -1 day (*time - time = duration*)
  Today + tomorrow = *error* (*time + time is undefined*)

- Conversion to floating point secods

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

ros::Duration d(0.5);
secs = d.toSec();
```

ROS Sleeping and Rates

- Sleep for the amount of time specified by the duration

```
ros::Duration(0.5).sleep(); // sleep for half a second
```

- *roslib* provides a **ros::Rate** (**ros::WallRate**) convenience class which makes a best effort at maintaining a particular rate for a loop

```
ros::Rate r(10); // 10 hz
while (ros::ok())
{
  ... do some work ...
  r.sleep();
}
```

Simulated Time in ROS

- For simulations or playback of logged data, it is convenient to work with a simulated time (pause, slow-down etc.)
- Using a simulated clock: rosparam set use_sim_time true
- When using simulated Clock time, now() returns time 0 until first message has been received on /clock, so 0 means essentially that the client does not know clock time yet.
- A value of 0 should therefore be treated differently, such as looping over now() until non-zero is returned.

```
ros::Time a_little_after_the_beginning(0.001);
```

Recording and replaying messages

- One of the primary features for ROS
- Consumers do not care how data is produced
- Producers do not care how data is consumed
- Record one or multiple topics
- Replay the recordings

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 visualisation and analysis
- Bags are saved with start date and time as file name in the current folder (e.g. 2018-03-02- 11-37-23.bag)

ROS Bags

RECORDING BAG FILES

ROSBAG RECORD -O FILENAME.BAG TOPIC-NAMES

INSPECTING BAG FILES

ROSBAG INFO FILENAME.BAG

REPLAYING BAG FILES

ROSBAG PLAY FILENAME.BAG

Bags in launch files

- Bag files can also be used within launch files
- Can either record or play automatically

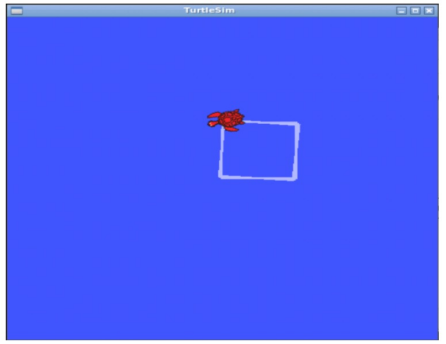
```
<node
    pkg="rosbag"
    name="record"
    type="record"
    args="-O filename.bag topic-names"
/>
```

```
<node
    pkg="rosbag"
    name="play"
    type="play"
    args="filename.bag"
/>
```

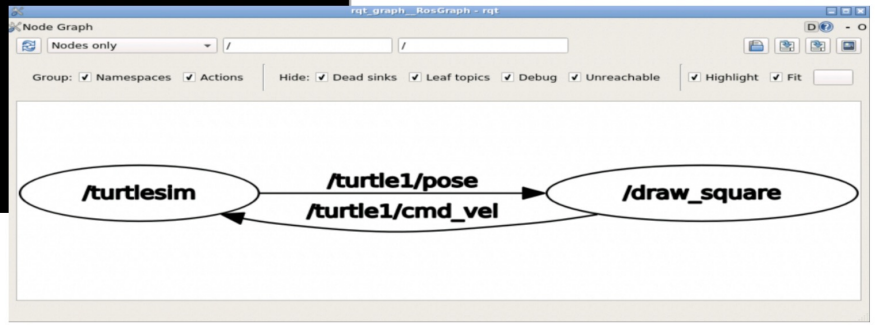
Example

No rosbag activated

```
user:~$ rosrun turtlesim draw_square
[ INFO] [1584701303.815341450]: New goal [7.544445 5.544445, 0.000000]
[ INFO] [1584701305.735351123]: Reached goal
[ INFO] [1584701305.735424077]: New goal [7.448444 5.544445, 1.570796]
[ INFO] [1584701309.671084511]: Reached goal
[ INFO] [1584701309.671162126]: New goal [7.466837 7.544360, 1.561600]
[ INFO] [1584701311.607165170]: Reached goal
[ INFO] [1584701311.607227514]: New goal [7.465954 7.448364, 3.132396]
```



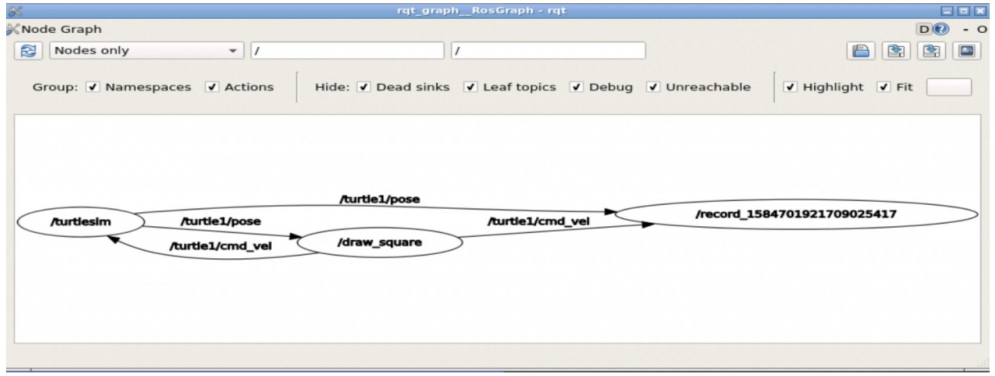

```
user:~$ rostopic echo /turtle1/pose
x: 5.68563461304
y: 5.5736319397
theta: 6.24639987946
linear_velocity: 1.0
angular_velocity: 0.0
---
x: 5.70162343979
y: 5.57304477692
theta: 6.24639987946
linear_velocity: 1.0
angular_velocity: 0.0
---
x: 5.71761274338
y: 5.57245635986
theta: 6.24639987946
linear_velocity: 1.0
angular_velocity: 0.0
```



Example

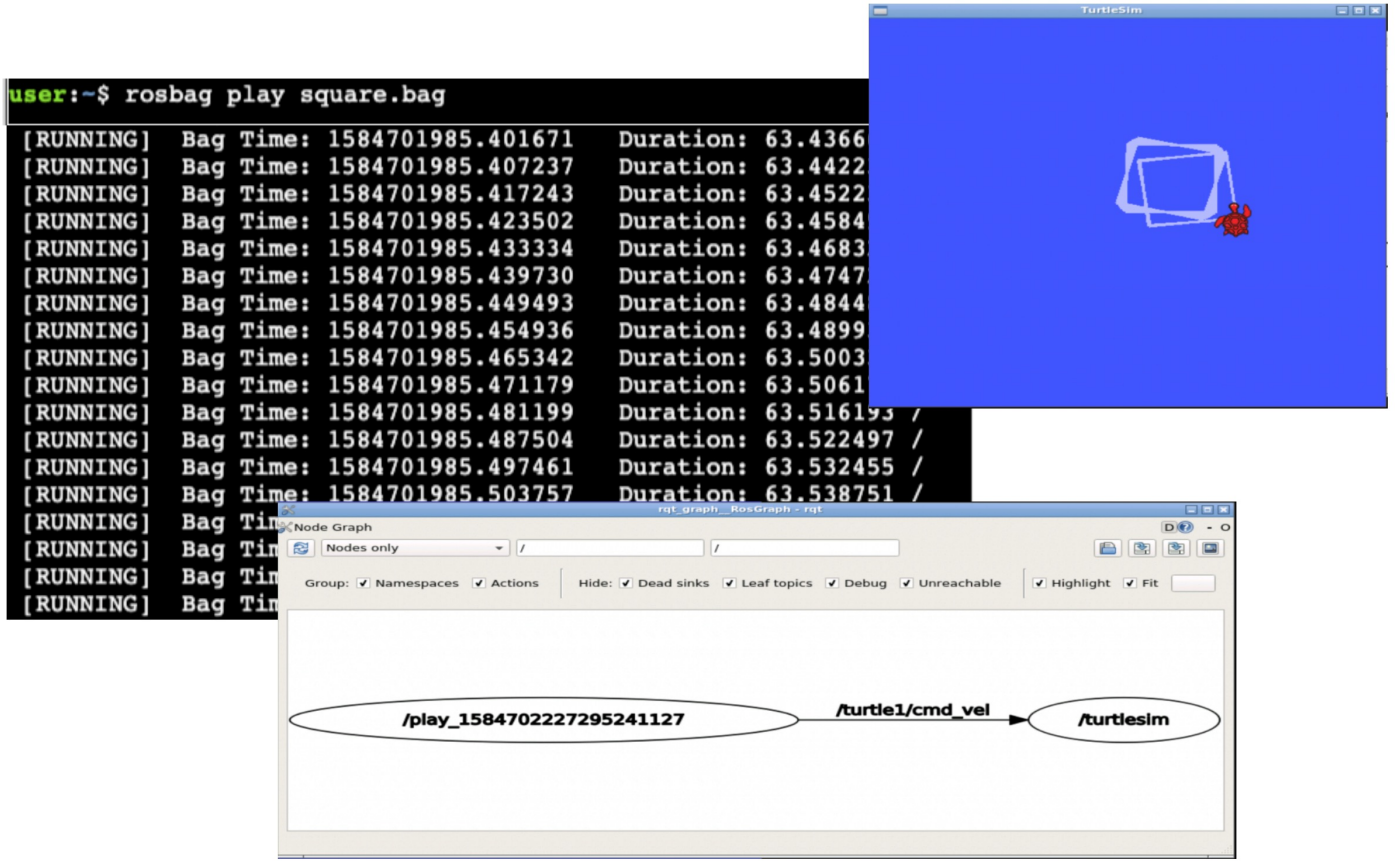
With rosbag

```
user:~$ rosbag record -O square.bag /turtle1/cmd_vel /turtle1/po
se
[ INFO] [1584701921.715508331]: Subscribing to /turtle1/cmd_vel
[ INFO] [1584701921.720126643]: Subscribing to /turtle1/pose
[ INFO] [1584701921.724771714]: Recording to square.bag.
```

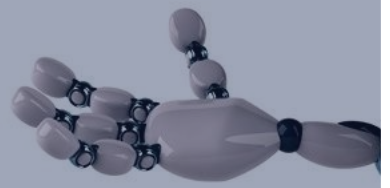


Example

Only rosbag



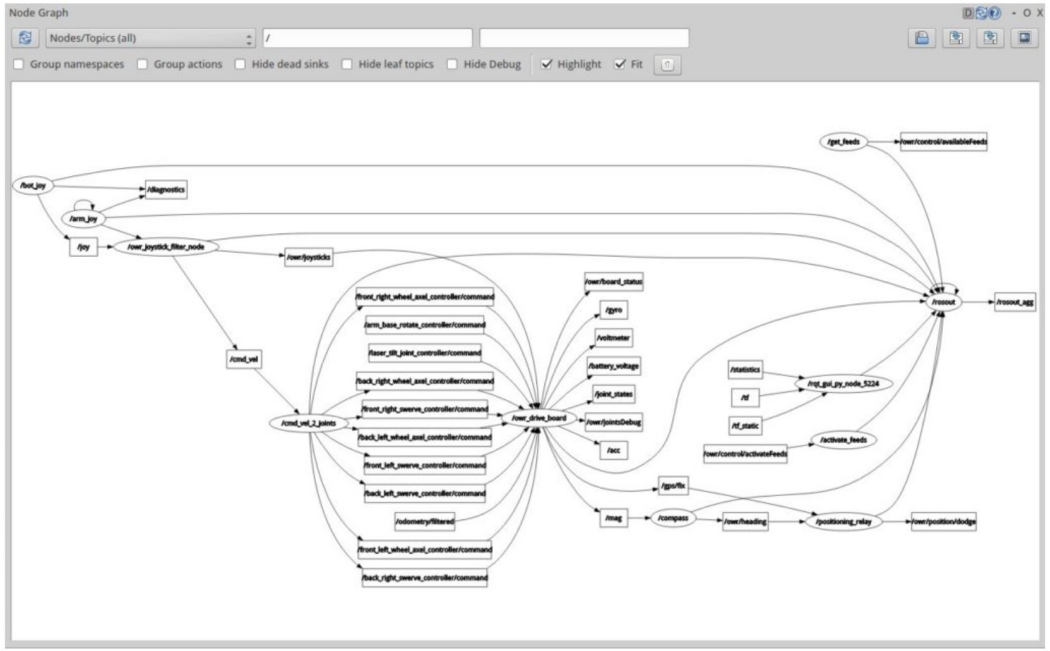
Debugging in ROS



RQT Tools. ROSWTF. Advanced debugging with GDB and Valgrind

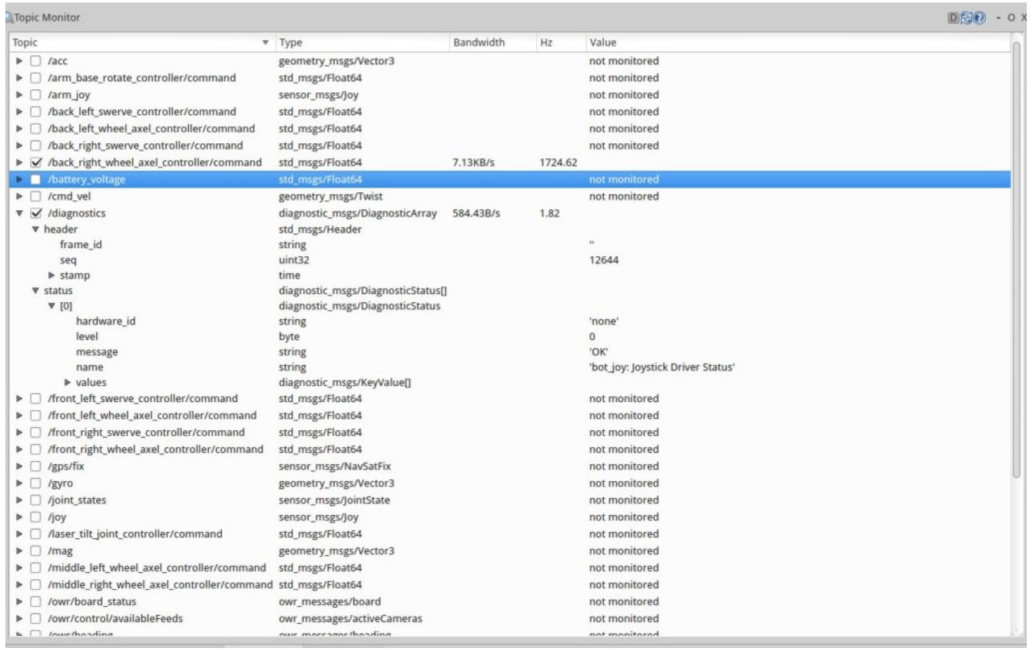
The Node Graph

- Nodes are ovals
- Topics are squares
- Topic advertisement / subscription => arrows
- Check which nodes are running
- Check if nodes are correctly connected



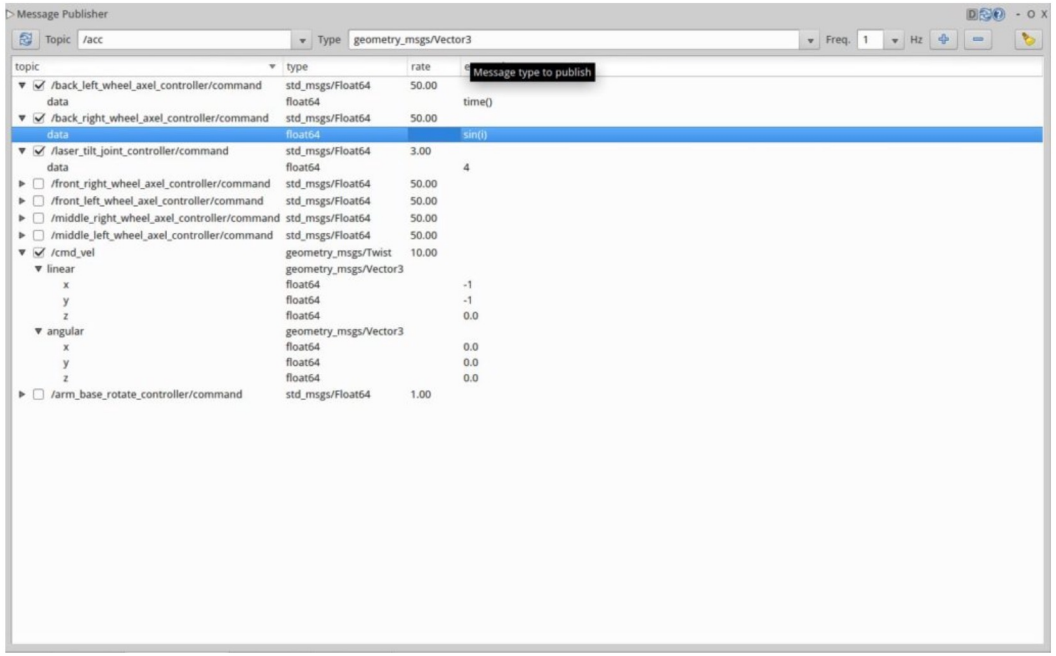
The Topic Monitor

- the younger, better organised sibling of *rostopic echo*
- displays a list of all currently advertised topics and allows them to be monitored



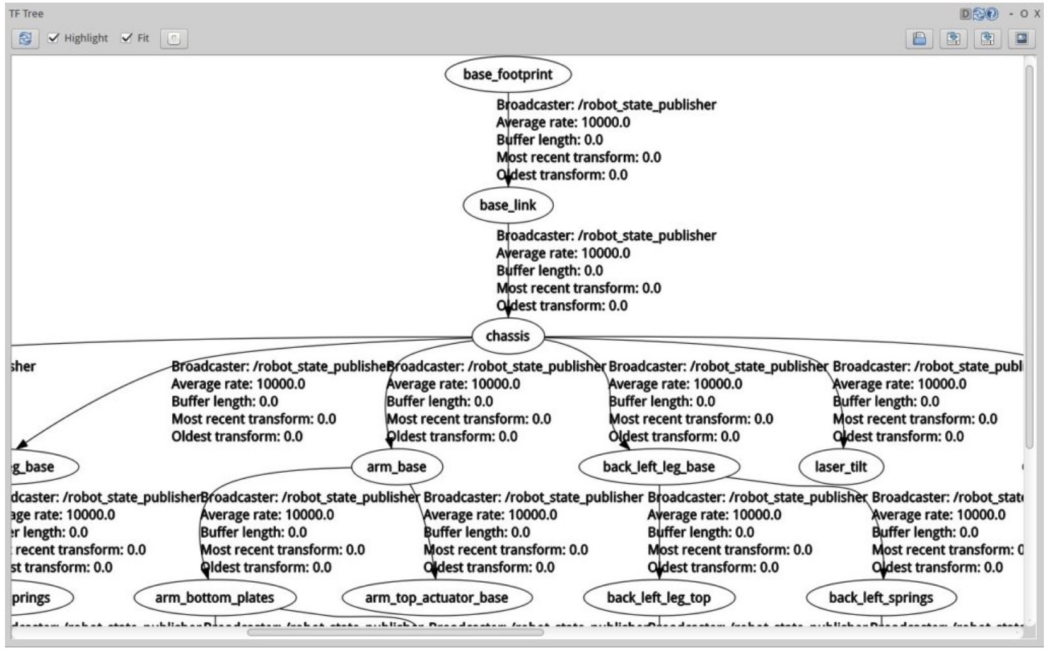
The Message Publisher

- Allows messages to be published equivalent to *rostopic pub*
- Select a topic, message type and frequency and then enter the data to be sent
- Shows a list of topics and a corresponding list of types
- Displays the fields of that message directly
- Shows last sent message with quick resend option



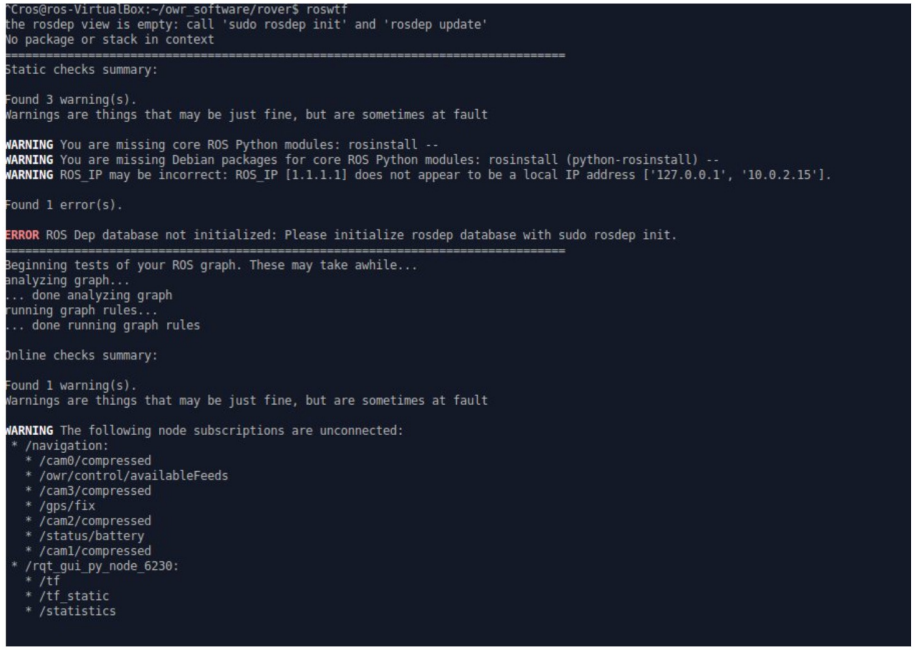
The TF Tree

- the connection structure of the transforms
- which node is publishing a given frame, the last time it was updated, and the oldest transform in the system
- detect gaps in the graph



ROSWTF

- ROS **W**here's **T**he **F**ire
- A debugging tool in ROS which helps you to find the problem
- Network and configuration issues (e.g. launch file problems)



GDB

- GNU Debugger
- Started from the command line in a terminal
- Can be run via IDE
- Requires code to be compiled with symbolic debugging information included
- Supports breakpoint definition
- Once execution breaks, allows stepping around
- Variable examination
- Quick Guide to GDB: http://beej.us/guide/bggdb/

Valgrind

- Valgrind tool suite provides a number of debugging and profiling tools that help you make your programs faster and more correct
- Memcheck = detect many memory-related errors that are common in C and C++ programs
- More information: https://www.valgrind.org/docs/manual/quick-start.html

Advanced Debugging using GDB and Valgrind

- Compile catkin workspace with debugging symbols enabled

```
$ catkin_make -DCMAKE_BUILD_TYPE=Debug

[Catkin Output Goes Here]
```

- Identify the correct executable to be run (cannot do *gdb rosrun …*)
- Run command on correct executable

```
$ gdb devel/lib/[ros_package_name]/[node_name]
```

```
$ valgrind --leak-check=yes devel/lib/[ros_package_name]/[node_name]
```

Advanced Debugging using GDB and Valgrind

- Creating a dedicated launch file

```
<launch>
   <node pkg="examples" type="example1" name="example1" output="screen" launch-prefix="xterm -e gdb --args" />
</launch>

<launch>
   <node pkg="examples" type="example1" name="example1" output="screen" launch-prefix="valgrind" />
</launch>
```

- Enabling core dumps for ROS nodes

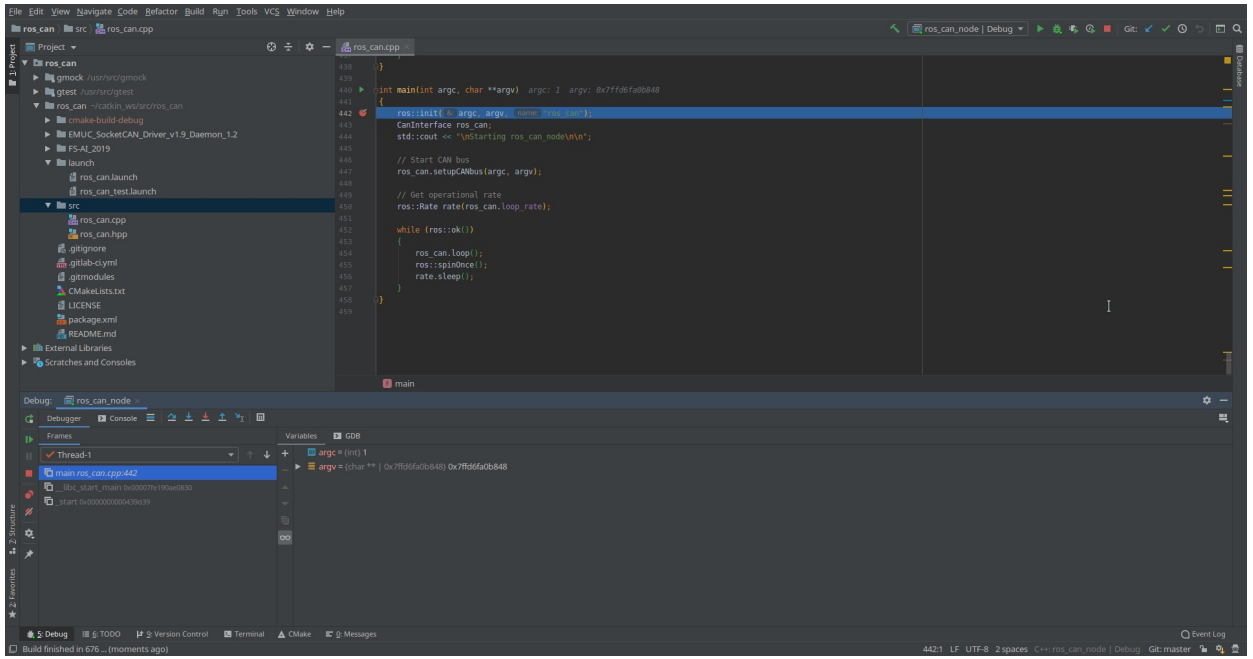
```
$ ulimit -c unlimited
```

- Set the core filename to use the pid process by default

```
$ echo 1 | sudo tee /proc/sys/kernel/core_uses_pid
```

CLion

- A cross-platform IDE for C and C++ (JetBrains)
- http://www.imgeorgiev.com/2020-02-02-ROS-debugging/



Thank you

Question Time