CS 545
Introduction to

Some slides have been adapted from
Sachin Chitta and Radu Rusu (Willow Garage)

Overview

?

my new application

OS

web browser

networking

window manager

memory management

process management

scheduler

device drivers

file system

Overview

Standards

Hardware: PCI bus, USB port, FireWire, ...

Software: HTML, JPG, TCP/IP, POSIX, ...

my new application

web browser

networking

window manager

memory management

process management

scheduler

device drivers

file system

OS

Overview

...but what about robots ?

my new application

web browser

networking

window manager

memory management

process management

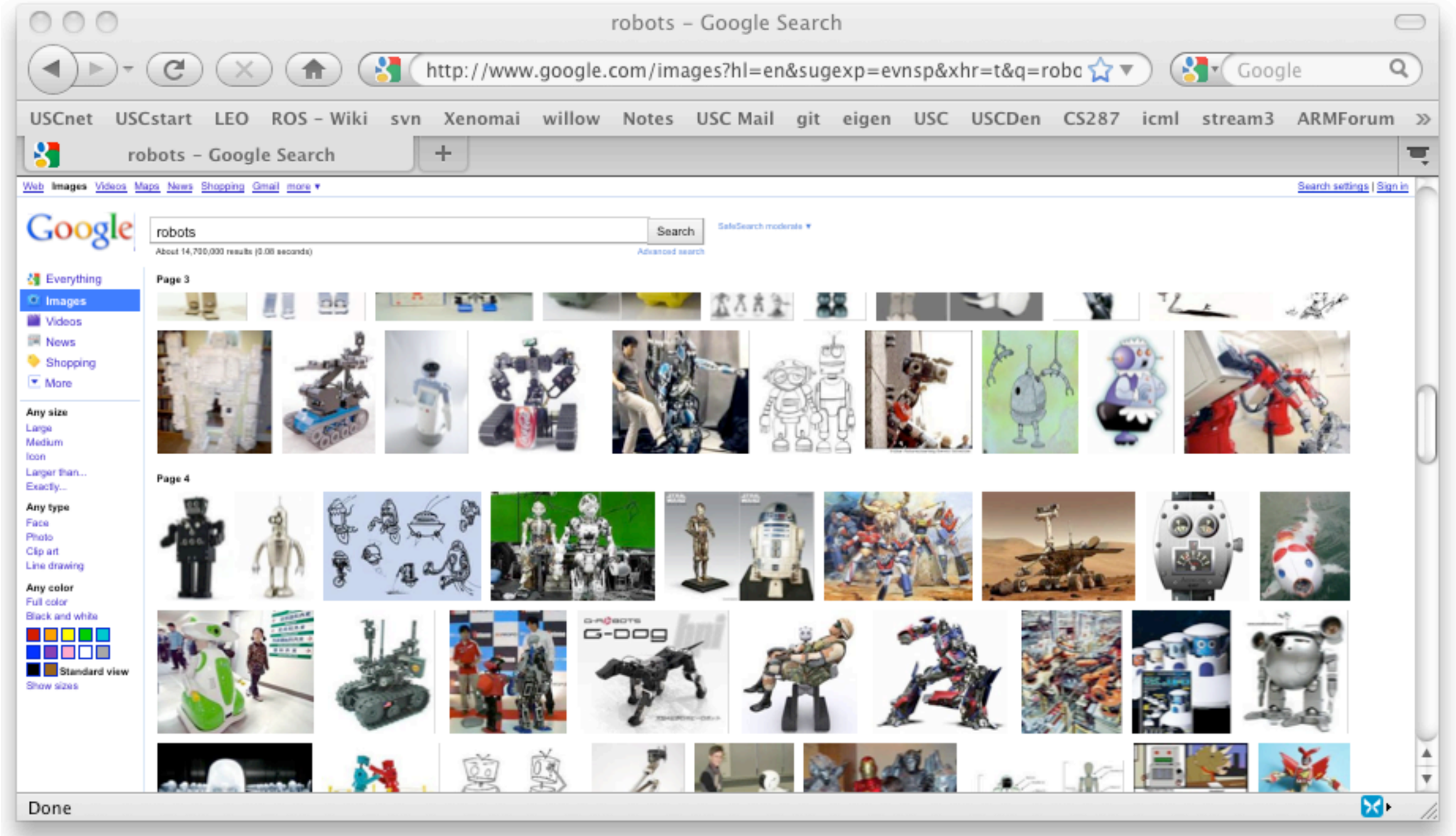
scheduler

device drivers

file system

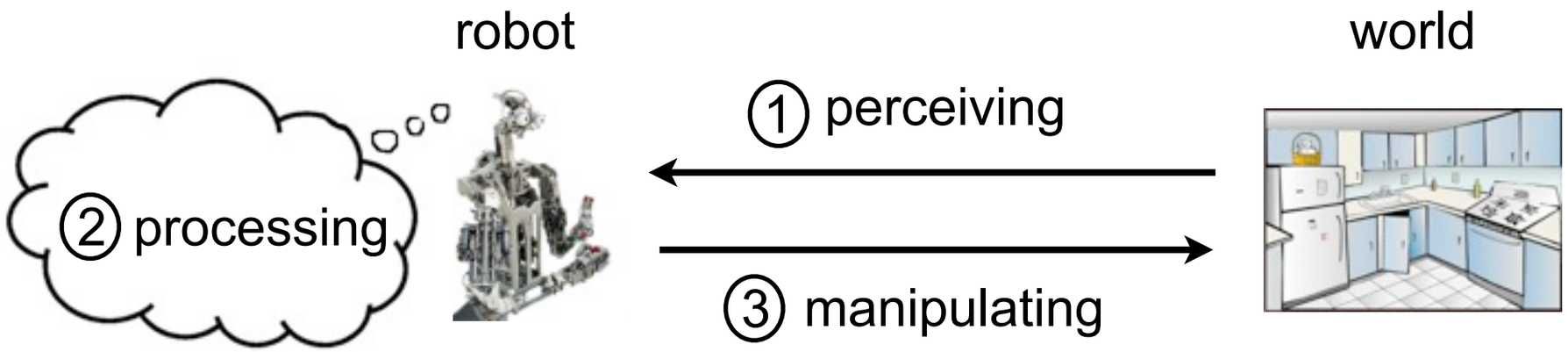
OS

Lack of standards for robotics



CS 545
Introduction to ROS

Typical scenario



① Many sensors require **device drivers** and **calibration procedures**

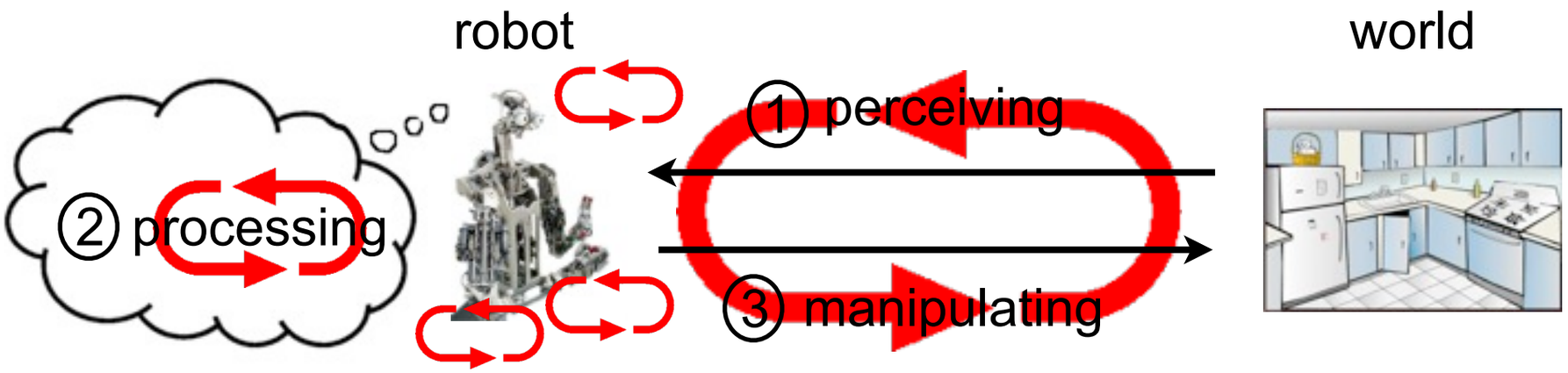
For example cameras: stereo processing, point cloud generation...

Common to many sensors: filtering, estimation, coordinate transformation, representations, voxel grid/point cloud processing, sensor fusion,...

② Algorithms for **object detection/recognition**, **localization**, **navigation**, **path/motion planning**, **decision making**, ...

③ Motor control: **inverse kinematics/dynamics**, **PID control**, **force control**, ...

Control loops



Many control loop on **different time scales**

Outer most control loop may run once every second (1Hz) or slower

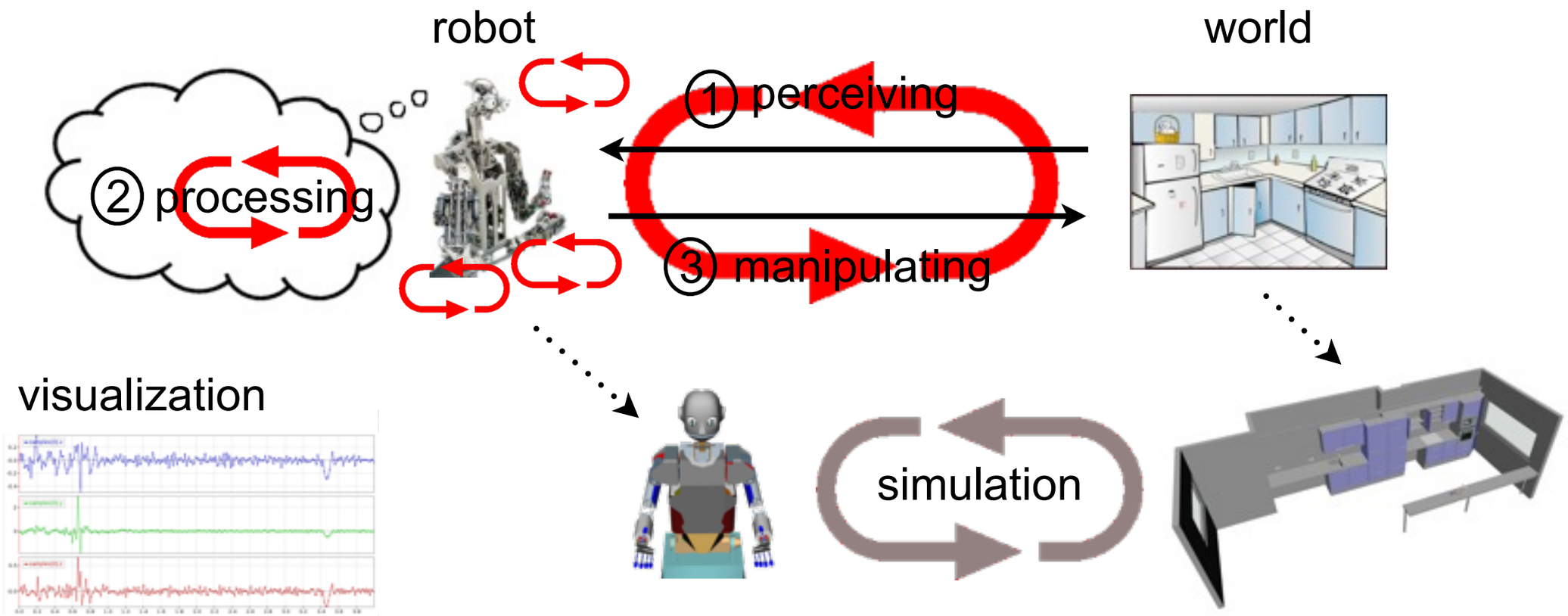
Inner most may run at 1000Hz or even higher rates

Software requirements:

Distributed processing with **loose coupling**. Sensor data comes in at various time scales.

Real-time capabilities for **tight** motor control **loops**.

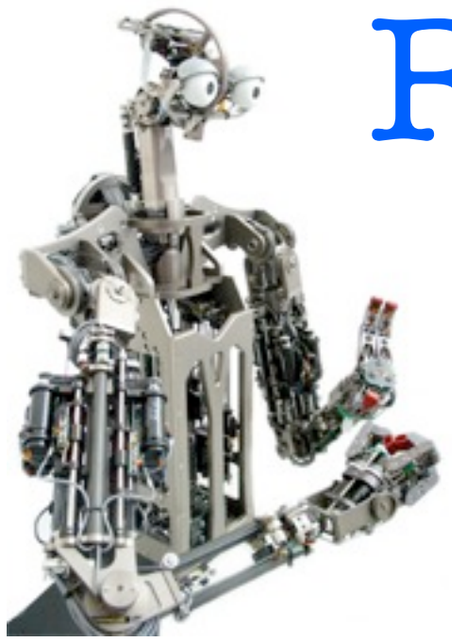
Debugging tools



Simulation: No risk of breaking real robots, reduce debugging cycles, test in super real-time, controlled physics, perfect model is available...

Visualization: Facilitates debugging, ...looking at the world from the robot's perspective. Data trace inspections allow debugging on small time scales.

Overview



ROS

navigation

task executive

visualization

simulation

perception

control

planning

data logging

message passing

device drivers

real-time capabilities

OS

web browser

networking

window manager

memory management

process management

scheduler

device drivers

file system

Overview

[1] Orocos: <http://www.orocos.org>
[2] OpenRTM: <http://www.is.aist.go.jp>
[3] ROS: <http://www.ros.org>
[4] OPRoS: <http://opros.or.kr>
[5] JOSER: <http://www.joser.org>
[6] InterModalics: <http://intermodalics.eu>
[7] Denx: <http://denx.de>
[8] GearBox: <http://gearbox.sourceforge.net/gbx_doc_overview.html>

Why should we agree on one standard ?

Code reuse, code sharing:
    Stop inventing the wheel again and again... instead build on top of each other's code.

Ability to run the same code across multiple robots:
    Portability facilitates collaborations and allows for comparison.

What is ROS ?

ROS is an **open-source**, **meta-operating** system and stands for Robot Operating System.

It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management.

http://www.ros.org (code + documentation)
http://www.osrf.org (support + development)
https://lists.sourceforge.net/lists/listinfo/ros-users (mailing list)

http://www.ros.org/wiki/ROS/Installation (it's open, it's free !!)

Mainly supported for Ubuntu linux, experimental for Mac OS X and other unix systems.

http://www.ros.org/wiki/ROS/StartGuide (tutorials)

Robots using ROS

http://www.ros.org/wiki/Robots

many more....
...and many more to come...

ROS package system

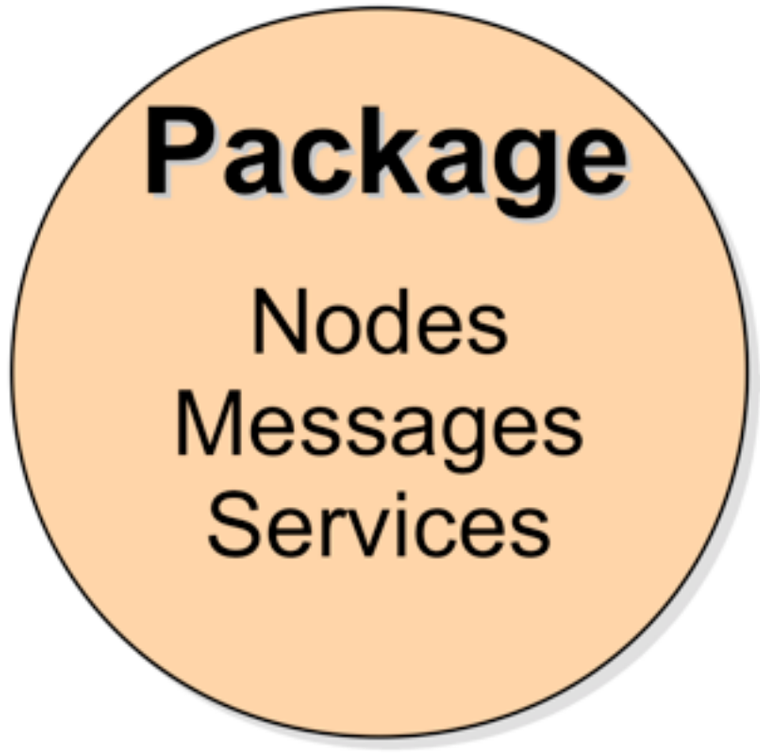
How to facilitate code sharing and code reuse ?

A package is a **building block** and implements a reusable capability

Complex enough to be useful

Simple enough to be reused by other packages

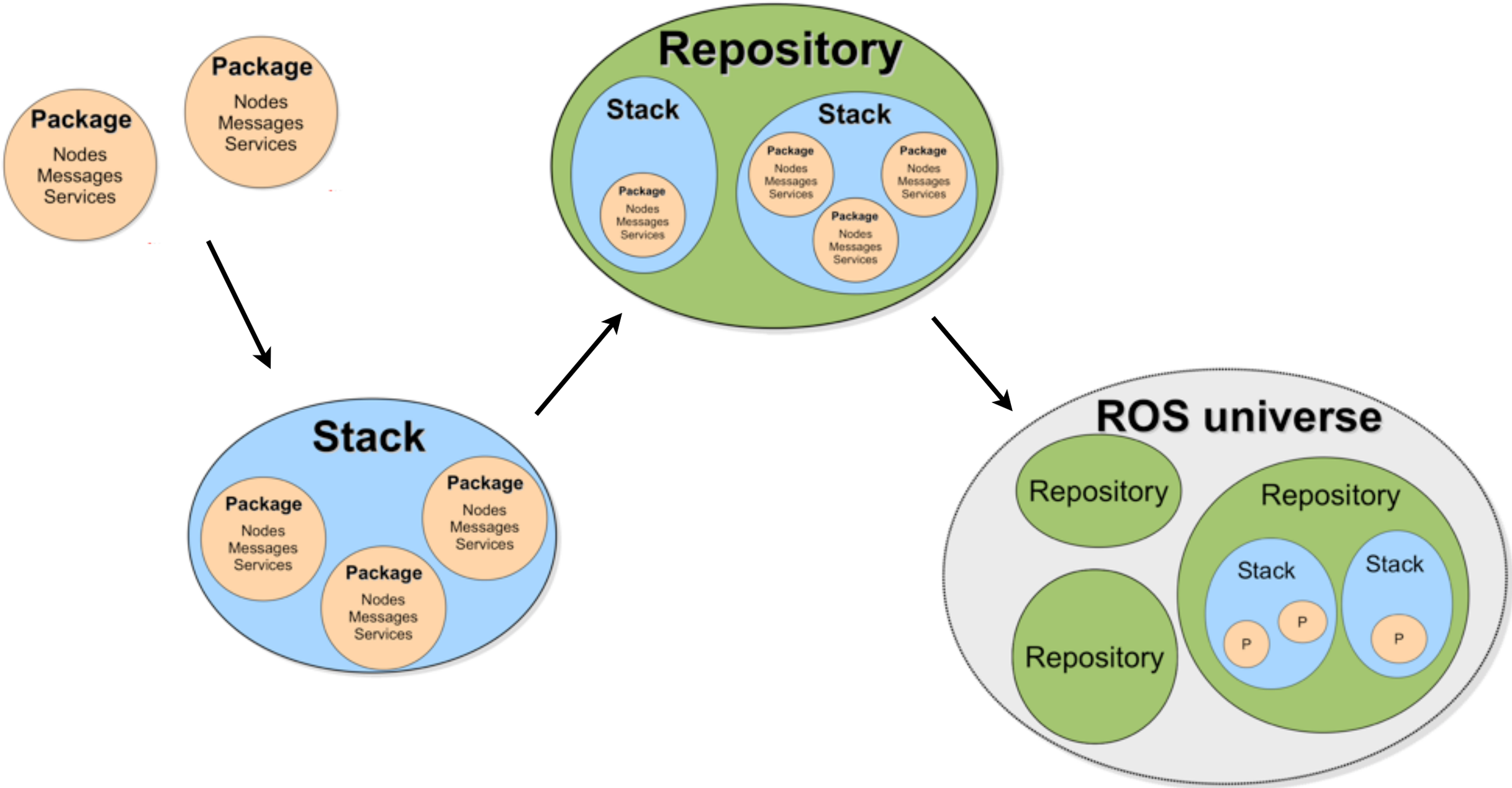
A **package** contains one or more executable processes (nodes) and provides a ROS interface:



Messages describe the data format of the in/output of the nodes. For example, a door handle detection node gets camera images as input and spits out coordinates of detected door handles.

Service and **topics** provide the standardized **ROS interface** to the rest of the system.

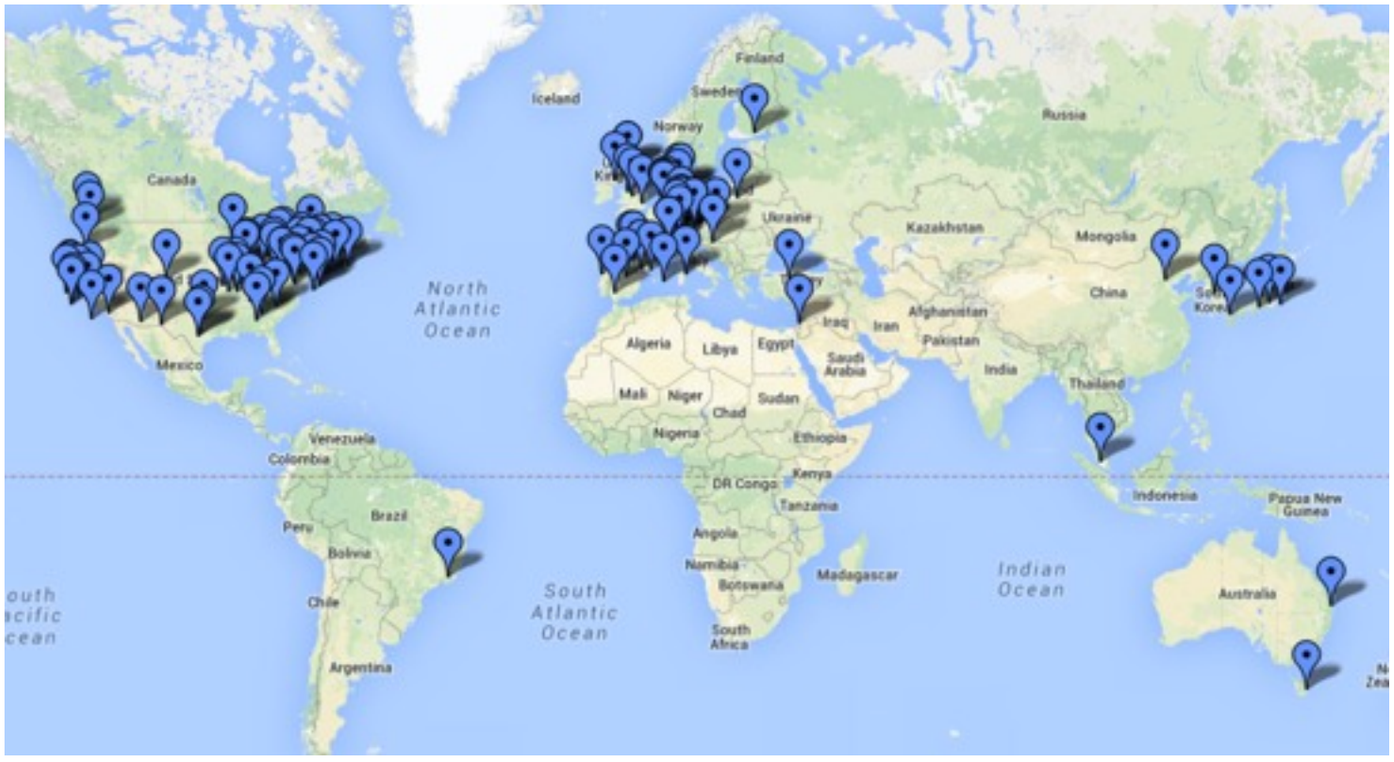
ROS package system



ROS package system

Collection of packages and stacks, hosted online
Many repositories (>90): Stanford, CMU, TUM, Leuven, USC, Bosch, ...

http://www.ros.org/wiki/Repositories (check it out...)



Version Control Systems (SVN, Git, Mercurial, Bazaar,...) are used to collaborate on the same code base. If you haven't worked with any, I **strongly** recommend to do so.

ROS package system

A **package** is a directory that might contain ROS nodes, a ROS-independent library, a dataset, configuration files, third-party software, or anything else that logically constitutes a useful module.

http://wiki.ros.org/Packages

ROS packages tend to follow a common structure. Here are some of the directories and files you may notice.

- `bin/`: compiled binaries (**C++ nodes**)
- `include/package_name`: C++ include headers
- `msg/`: **Message** (msg) types
- `src/`: C++ source files
- `src/package_name/`: Python source files
- `srv/`: **Service** (srv) types
- `scripts/`: executable scripts (Python nodes)
- `launch/`: launch files
- `CMakeLists.txt`: CMake build file (**rosbuild/catkin**)
- `manifest.xml`: Package **Manifest**
- `mainpage.dox`: Doxygen mainpage documentation

ROS package system

manifest.xml →

```
<package>
  <description brief="one line of text">
    long description goes here,
    <em>XHTML is allowed</em>
  </description>
  <author>Alice/alice@somewhere.bar</author>
  <license>BSD</license>

  <depend package="roscpp"/>
  <depend package="my_package"/>
  <rosdep name="libreadline5-dev"/>
  <export>
    <cpp cflags="-I${prefix}/include"
         lflags="-L${prefix}/lib -lmy_lib"/>
  </export>

</package>
```

The **manifest** is a minimal specification about a package and supports a wide variety of ROS tools.

http://wiki.ros.org/Manifest

ROS core

master

The **roscore** is a collection of nodes and programs that are pre-requisites for a ROS-based system.

It provides naming and registration services to the rest of the **nodes** in the ROS system. It tracks publishers and subscribers to **topics** as well as **services**.

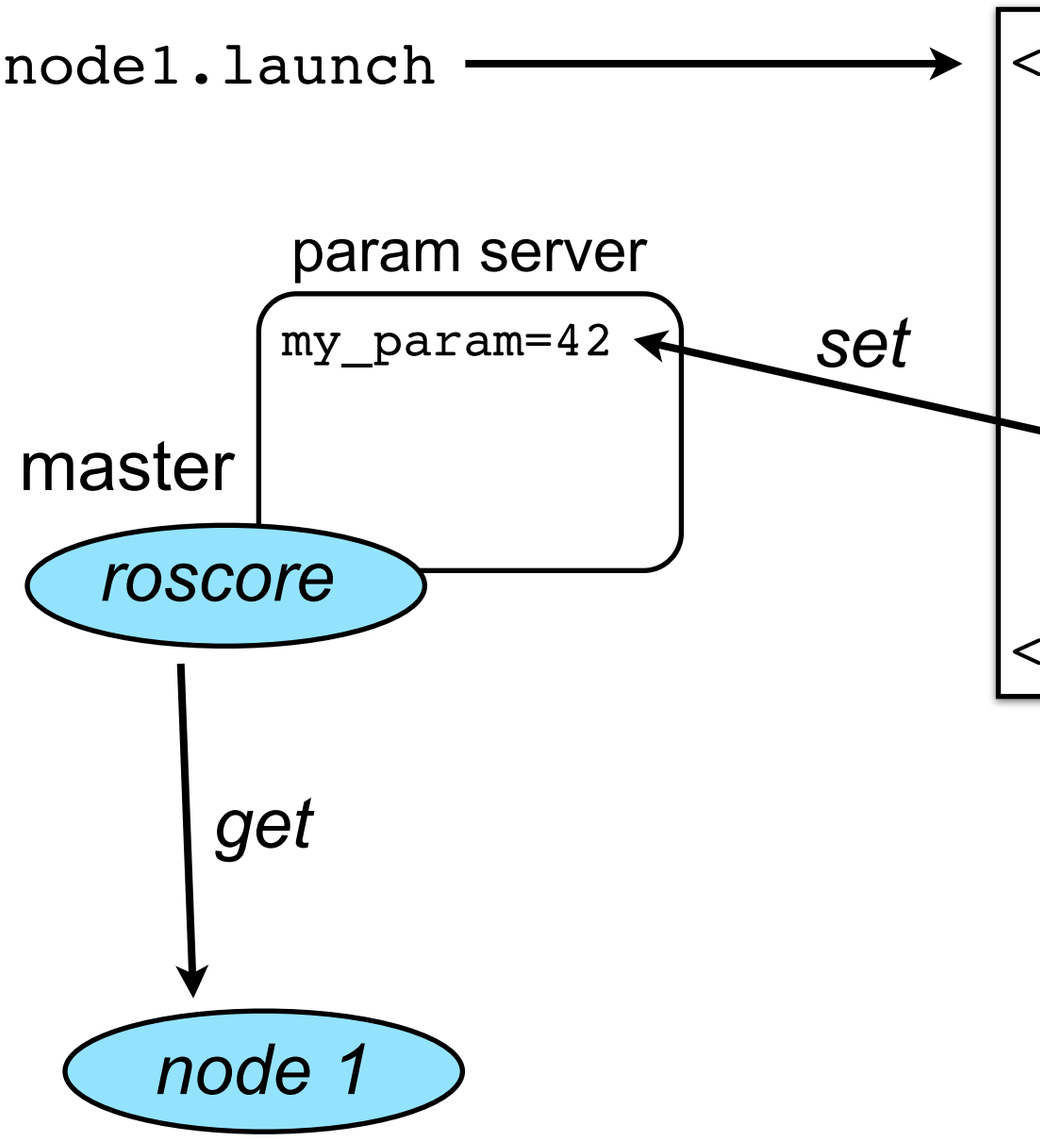
The role of the master is to enable individual ROS **nodes** to locate one another. Once these nodes have located each other they communicate with each other peer-to-peer.

ROS uses socket communication to facilitate networking. The **roscore** is reachable at the socket address stored in the environment variable `ROS_MASTER_URI`, for example `http://my_computer:11311`

ROS package system

node1.launch

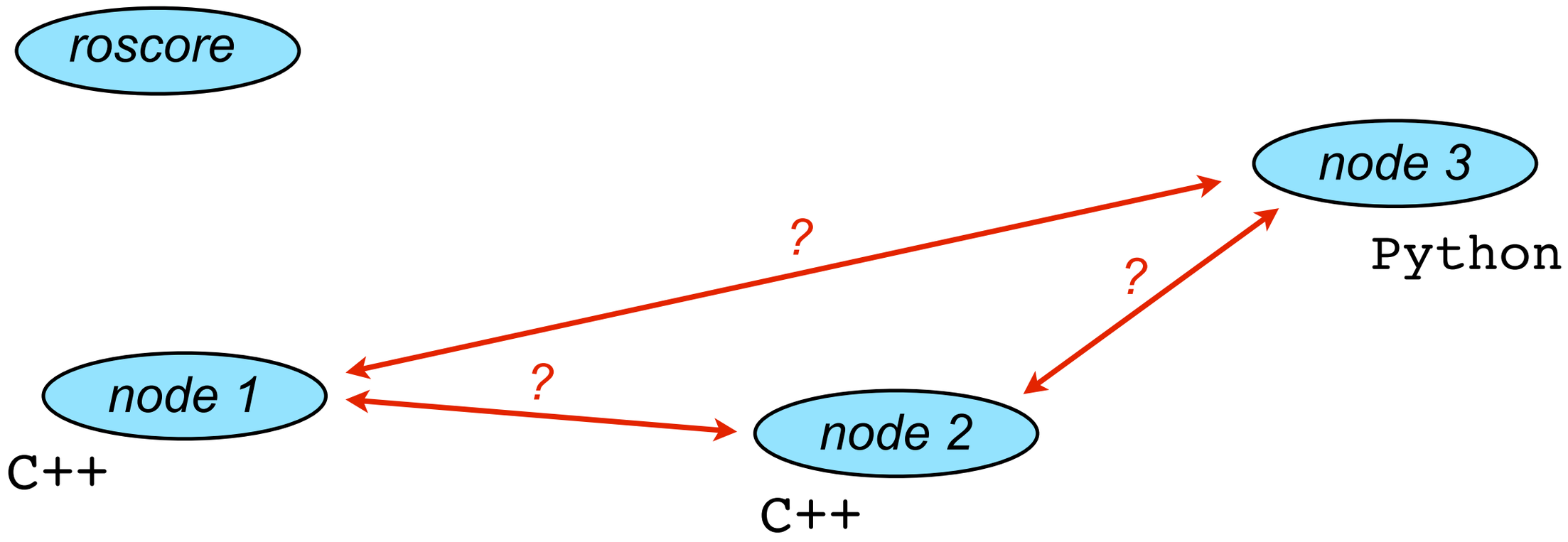
```
<launch>

  <node pkg="my_package"
        type="node1"
        name="node1" args="--test">

    <param name="my_param" value="42"/>
  </node>

</launch>
```



ROS: message passing

Problem:

Synchronization and message passing across multiple processes, maybe even across multiple computer and/or robots.



ROS: message passing

Inter process communication using sockets : `TCPROS` or `UDPROS`

Serialization/Deserialization is taken care of by the ROS message system.

msg/MyData.msg

```
int32 data
string name
```

code generation (at compile time)

include/my_package/MyMsg.h

```
#include ...
class MyData ...
{...
```

master

roscore

node 1

C++

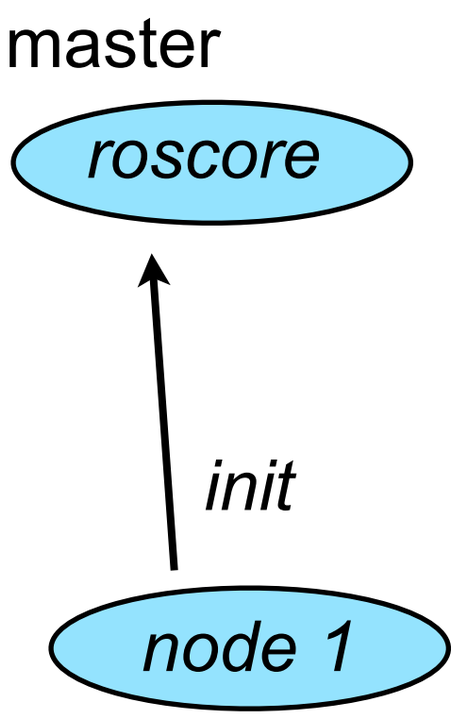
node 2

C++

node 3

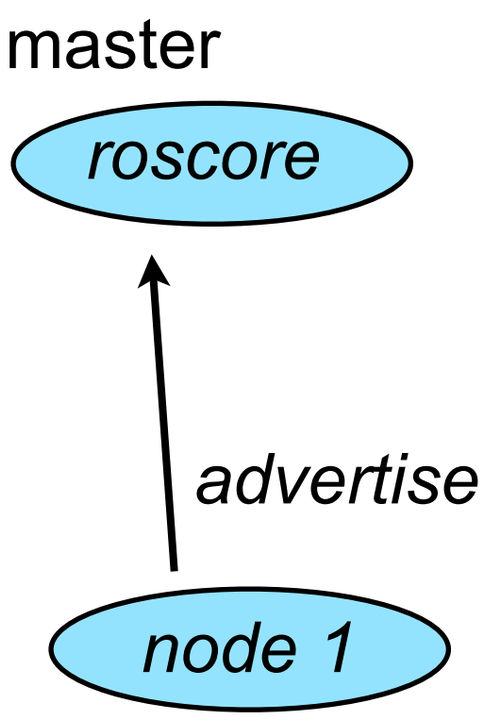
Python

ROS: message passing



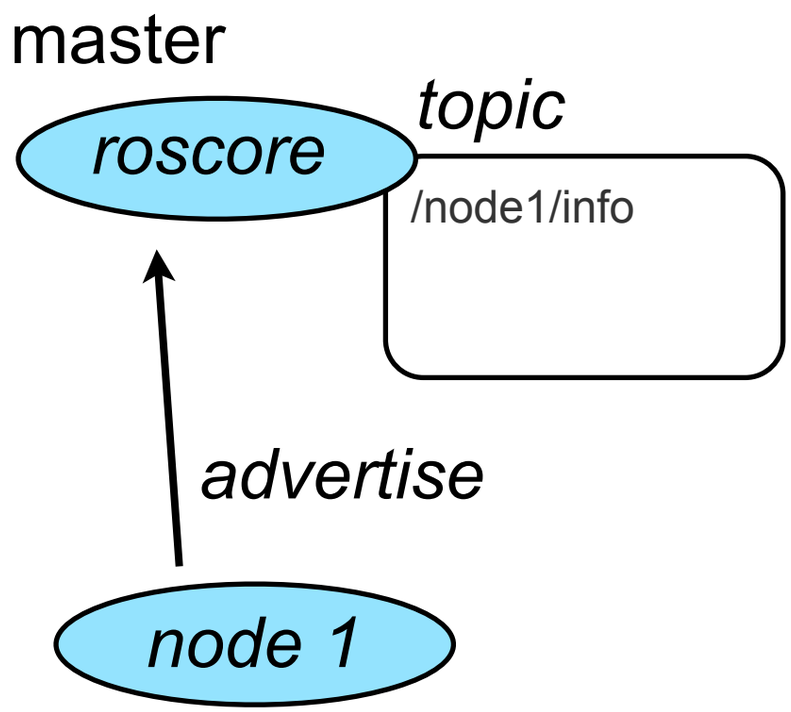

```
#include <ros/ros.h>

// auto generated message
#include <my_package/MyData.h>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "node1");
  ros::NodeHandle n;
  ros::Publisher chatter_pub =
    n.advertise<my_package::MyData>("info", 1000);
  ros::Rate loop_rate(10);
  int count = 0;
  while (ros::ok())
  {
    my_package::MyData msg;
    msg.data = count;
    msg.name = “hello world”;
    chatter_pub.publish(msg);
    ros::spinOnce();
    loop_rate.sleep();
    ++count;
  }
  return 0;
}
```

ROS: message passing



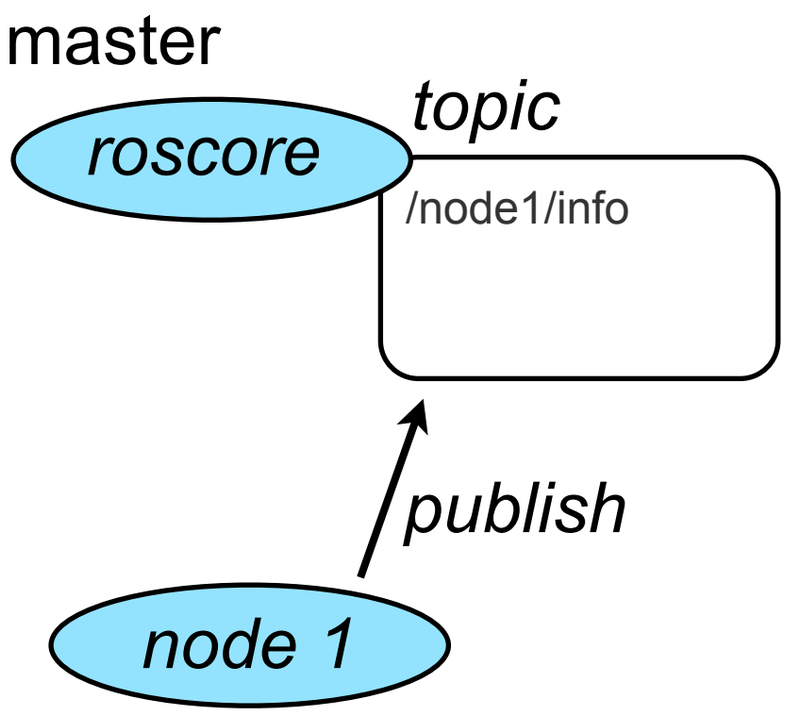

```
#include <ros/ros.h>

// auto generated message
#include <my_package/MyData.h>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "node1");
  ros::NodeHandle n;
  ros::Publisher chatter_pub =
    n.advertise<my_package::MyData>("info", 1000);
  ros::Rate loop_rate(10);
  int count = 0;
  while (ros::ok())
  {
    my_package::MyData msg;
    msg.data = count;
    msg.name = “hello world”;
    chatter_pub.publish(msg);
    ros::spinOnce();
    loop_rate.sleep();
    ++count;
  }
  return 0;
}
```

ROS: message passing




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

// auto generated message
#include <my_package/MyData.h>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "node1");
  ros::NodeHandle n;
  ros::Publisher chatter_pub =
    n.advertise<my_package::MyData>("info", 1000);
  ros::Rate loop_rate(10);
  int count = 0;
  while (ros::ok())
  {
    my_package::MyData msg;
    msg.data = count;
    msg.name = “hello world”;
    chatter_pub.publish(msg);
    ros::spinOnce();
    loop_rate.sleep();
    ++count;
  }
  return 0;
}
```

ROS: message passing

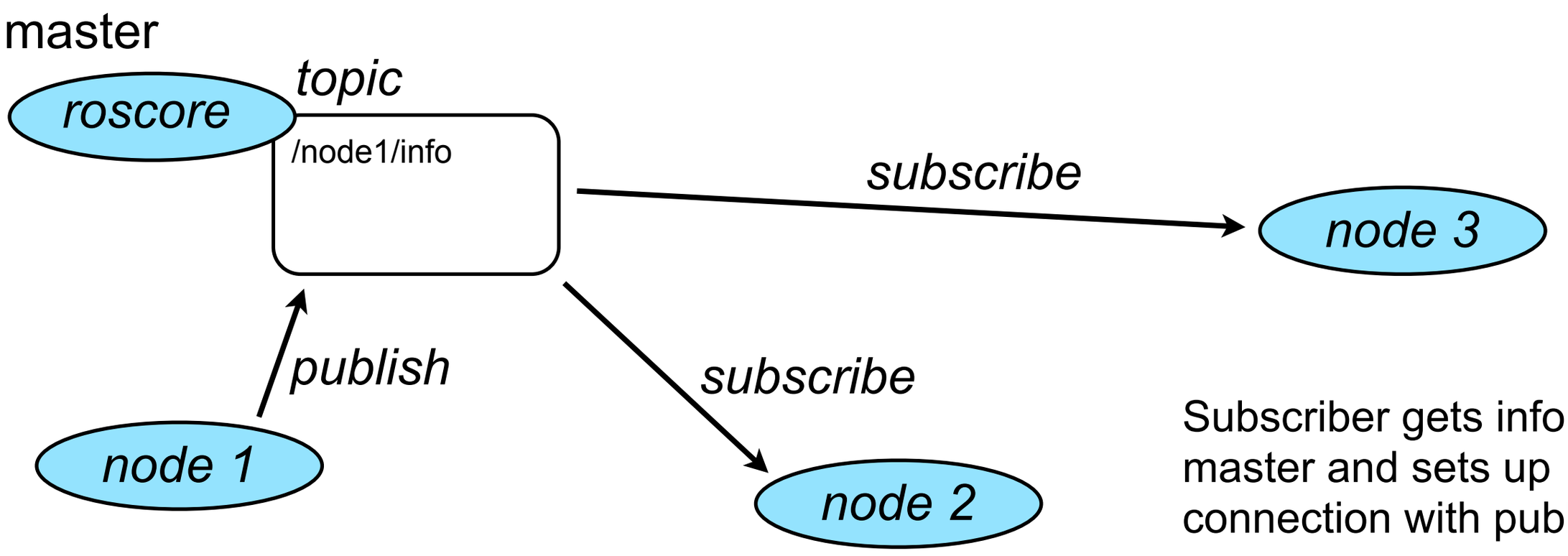



```
#include <ros/ros.h>

// auto generated message
#include <my_package/MyData.h>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "node1");
  ros::NodeHandle n;
  ros::Publisher chatter_pub =
    n.advertise<my_package::MyData>("info", 1000);
  ros::Rate loop_rate(10);
  int count = 0;
  while (ros::ok())
  {
    my_package::MyData msg;
    msg.data = count;
    msg.name = “hello world”;
    chatter_pub.publish(msg);
    ros::spinOnce();
    loop_rate.sleep();
    ++count;
  }
  return 0;
}
```

ROS: message passing

Note:

Message data does **not** flow through the master. It only provides name service, connecting subscribers with publishers.

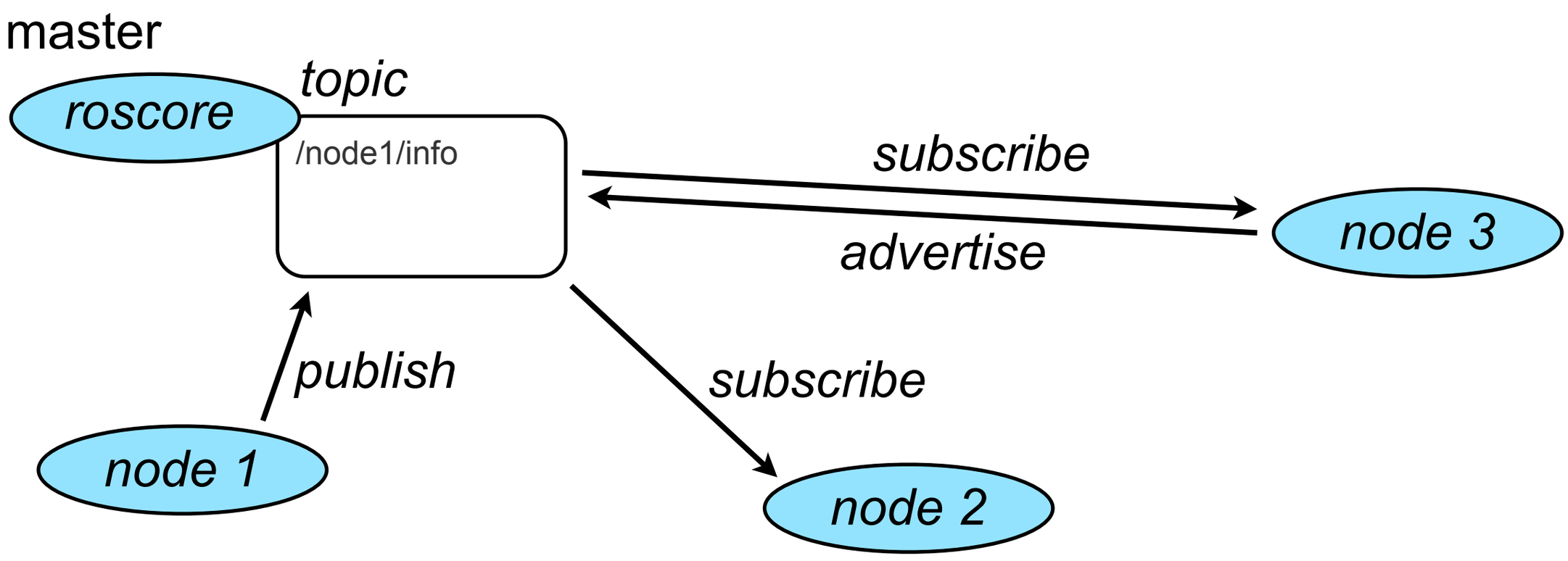


Subscriber gets info from master and sets up connection with publisher.

ROS: message passing

Note:

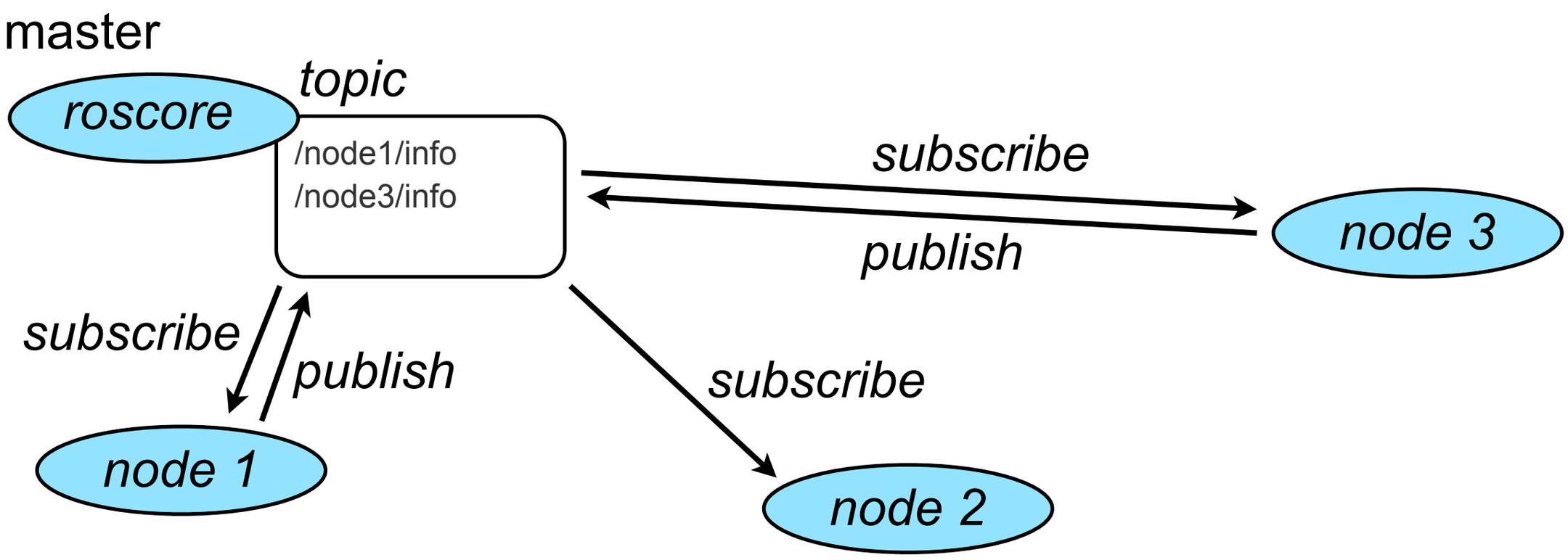
Message data does **not** flow through the master. It only provides name service, connecting subscribers with publishers.



ROS: message passing

Note:

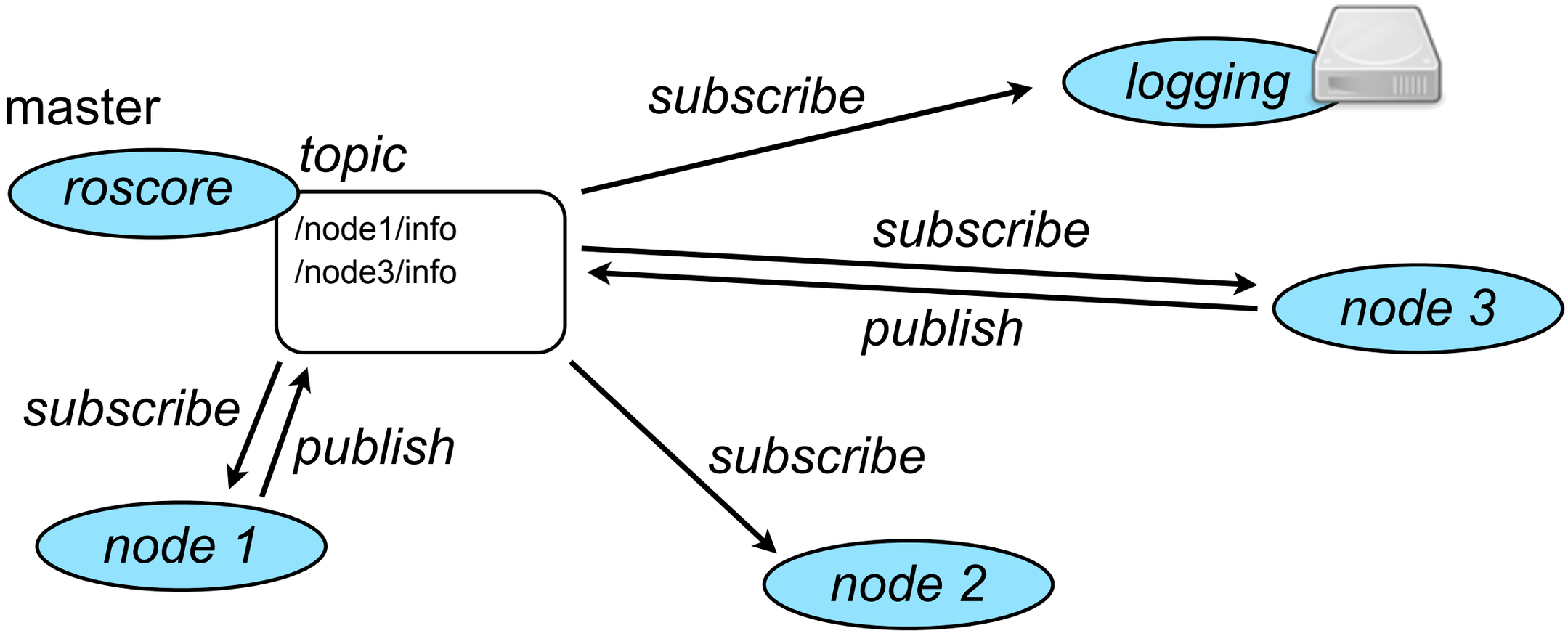
Message data does **not** flow through the master. It only provides name service, connecting subscribers with publishers.



ROS: logging

Problem:

How to log sensor data or debug information from potentially multiple processes simultaneously ?



ROS: logging

rosbag: This is a set of tools for recording from and playing back to ROS topics. It can be used to mimic real sensor streams for offline debugging.

http://www.ros.org/wiki/rosbag

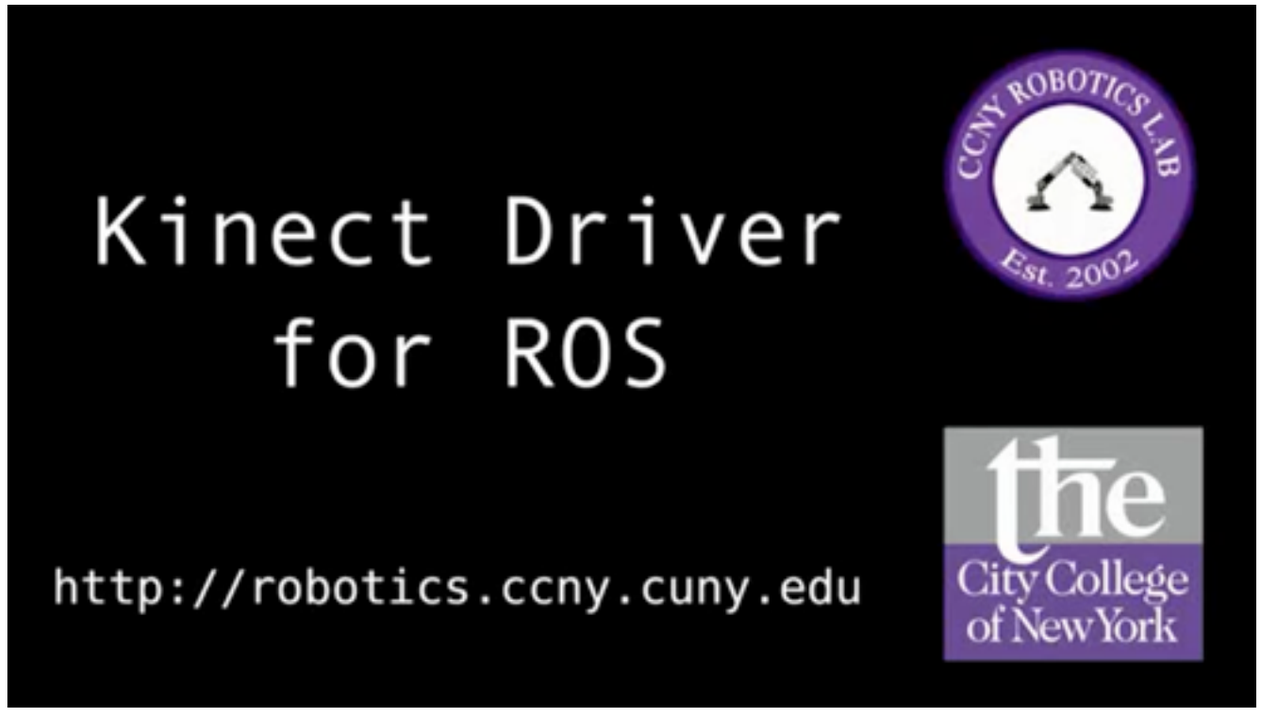
ROS: device drivers

Problem:

Many sensors do not come with **standardized interfaces**. Often the manufacturer only provides support for a single operating system (e.g. Microsoft Windows).

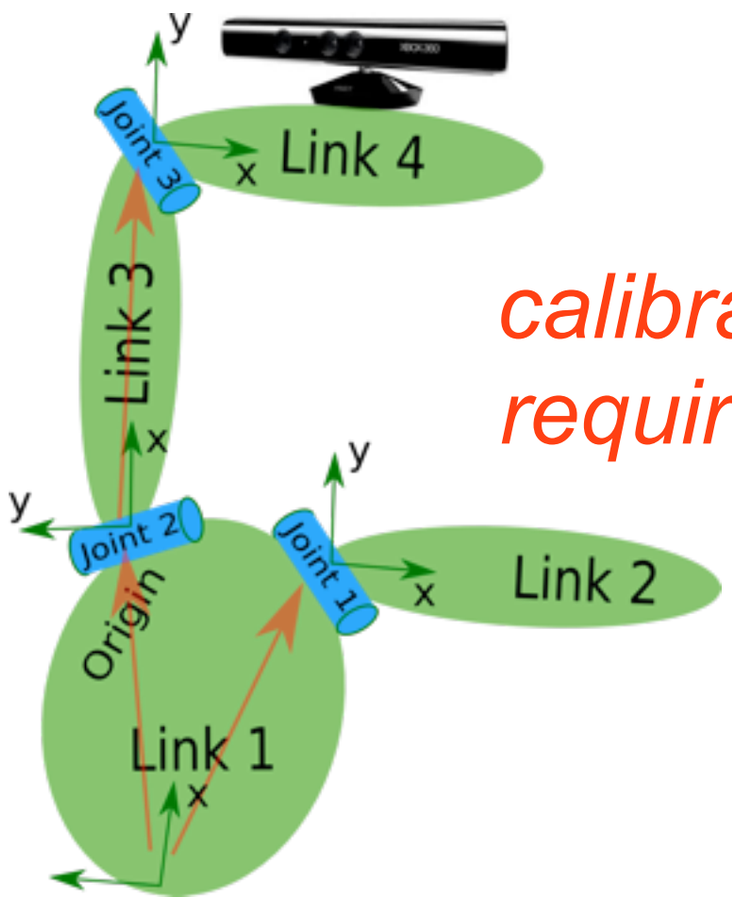
Thus, everybody that wants to use a particular sensor is required to write their own **device driver**, which is time consuming and tedious.

The **ROS interface** facilitates code reuse: many can (re-)use code that very few people wrote and build on top of it.



ROS: robot descriptions

urdf: This package contains a C++ parser for the **U**nified **R**obot **D**escription **F**ormat (URDF), which is an XML format for representing a robot model.



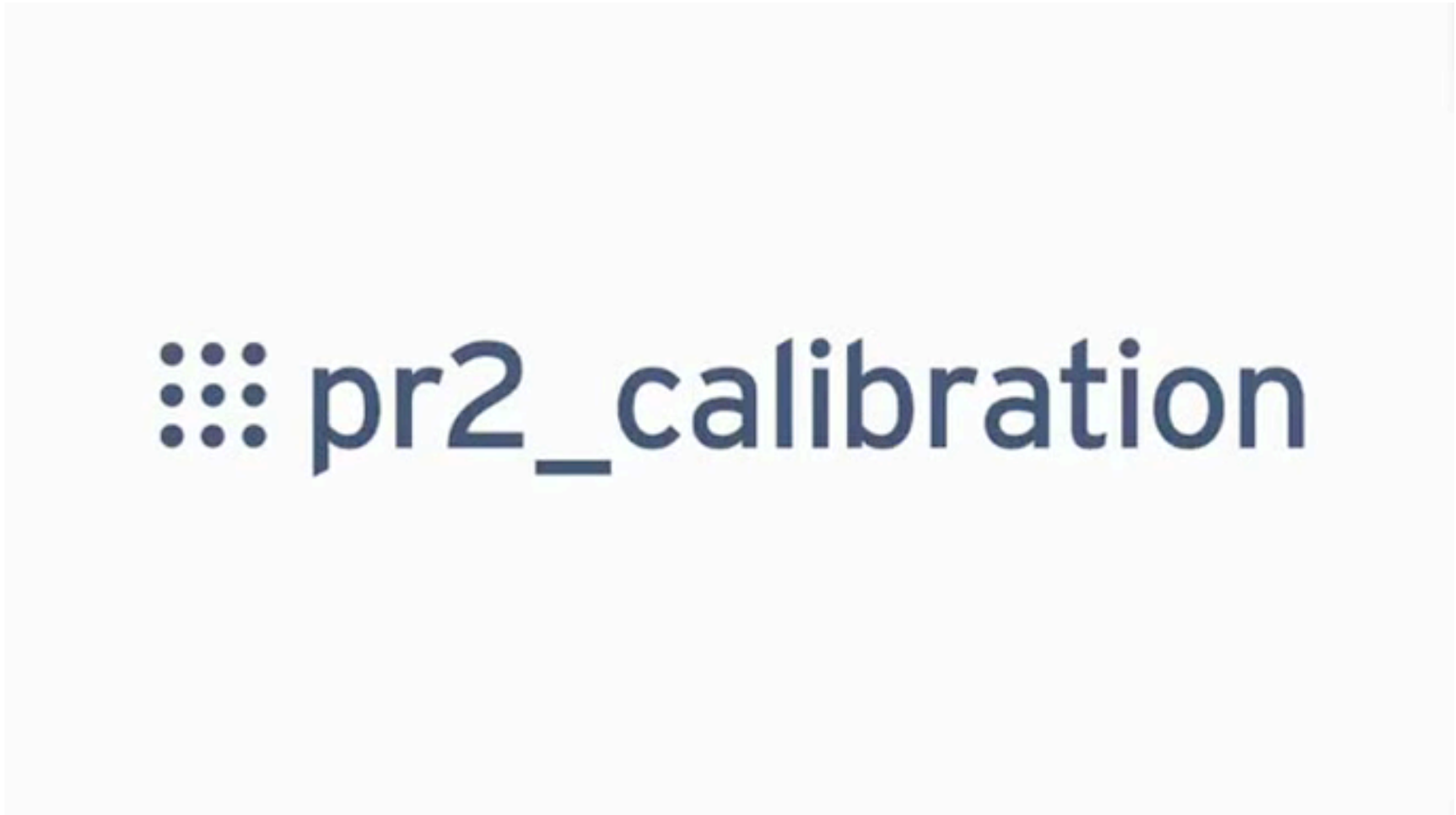
calibration required !!

```
<robot name="test_robot">
  <link name="link1" />
  <link name="link2" />
  <link name="link3" />
  <link name="link4" />

  <joint name="joint1" type="continuous">
    <parent link="link1"/>
    <child link="link2"/>
  </joint>

  <joint name="joint2" type="continuous">
    <parent link="link1"/>
    <child link="link3"/>
  </joint>

  <joint name="joint3" type="continuous">
    <parent link="link3"/>
    <child link="kinect_link"/>
  </joint>
</robot>
```

http://www.ros.org/wiki/urdf

ROS: calibration

Provides a toolchain running through the robot calibration process. This involves capturing pr2 calibration data, estimating pr2 parameters, and then updating the PR2 URDF.

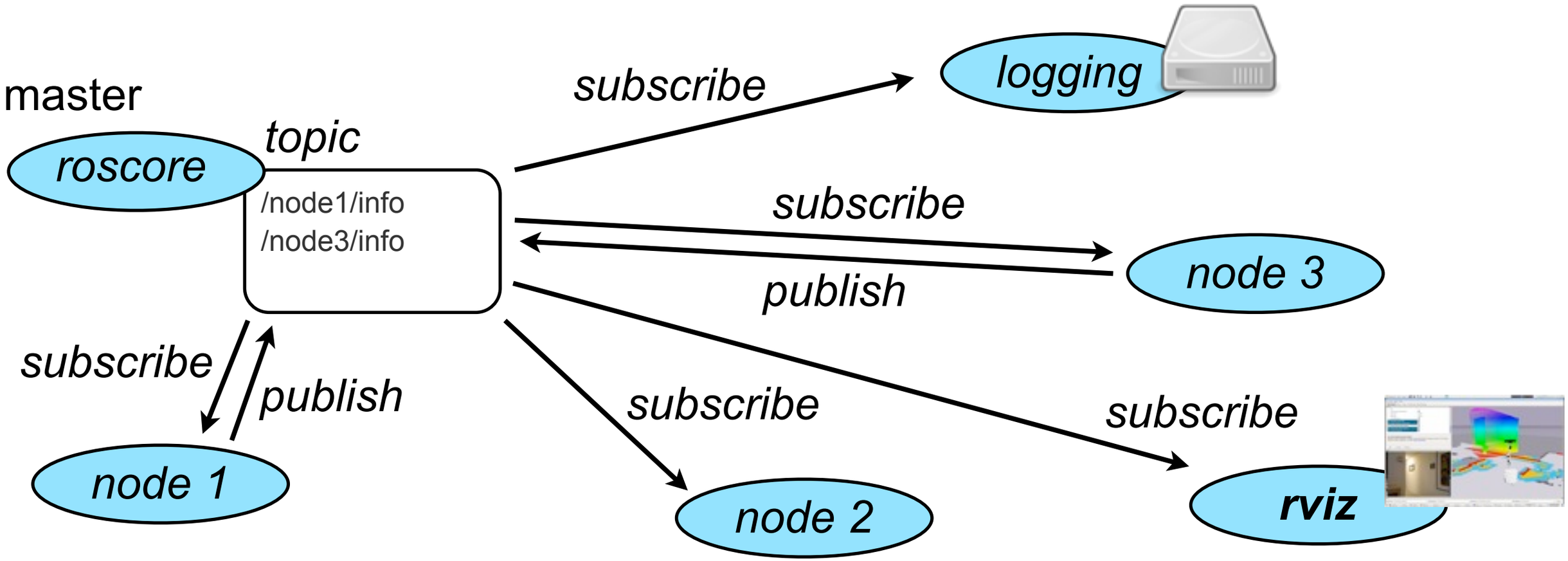


http://www.ros.org/wiki/pr2_calibration

ROS: debugging

Problem:

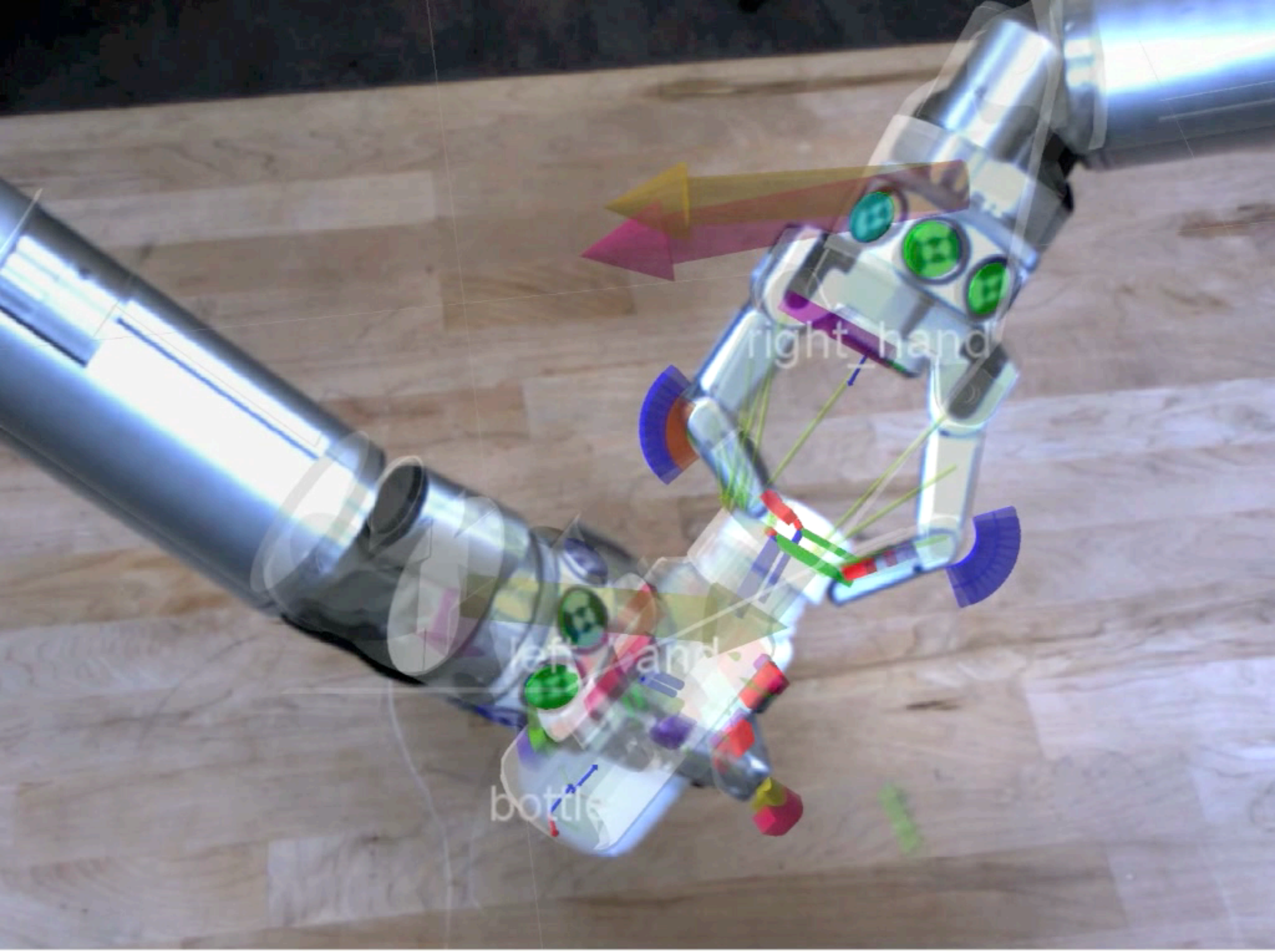
Debugging tools are **very** important. Visualizing the state of the robot and the perceived sensor data allows you to better understand what's going on.



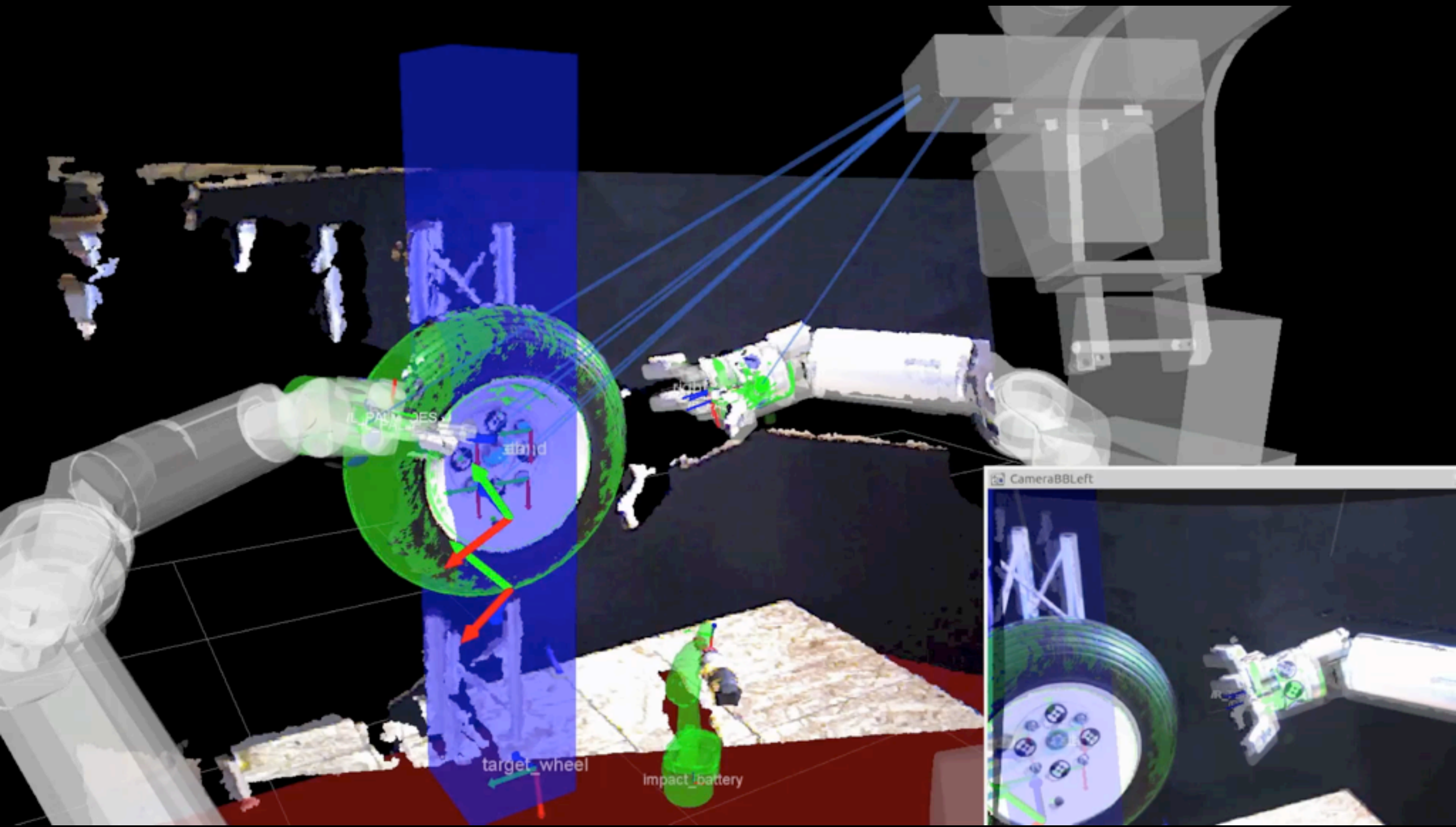
ROS: visualization

rviz: This is a 3D visualization environment for robots. It allows you to see the world through the eyes of the robot.

http://www.ros.org/wiki/rviz


right_hand
left_hand
bottle

ROS: visualization

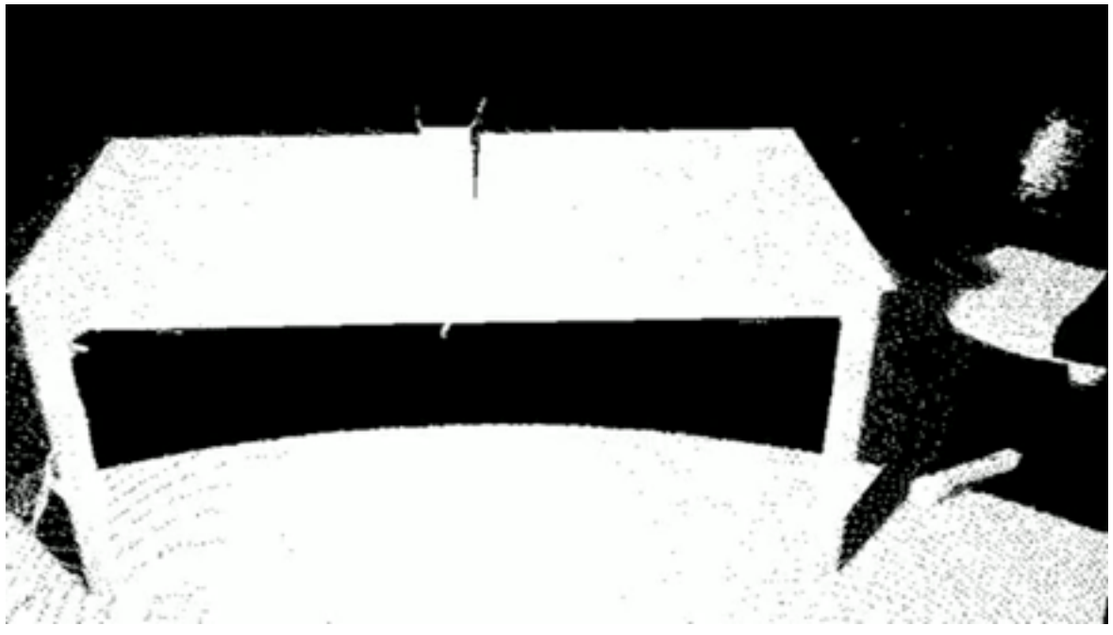


ROS: 2D/3D perception

OpenCV: (**Open** Source **C**omputer **V**ision) is a library of programming functions for real time computer vision. http://opencv.willowgarage.com/wiki/

Check out CS 574 (Prof. Ram Nevatia) !!

PCL - Point **C**loud **L**ibrary: a comprehensive open source library for **n-D Point Clouds** and **3D geometry processing**. The library contains numerous state-of-the art algorithms for: filtering, feature estimation, surface reconstruction, registration, model fitting and segmentation, etc.



http://www.ros.org/wiki/pcl

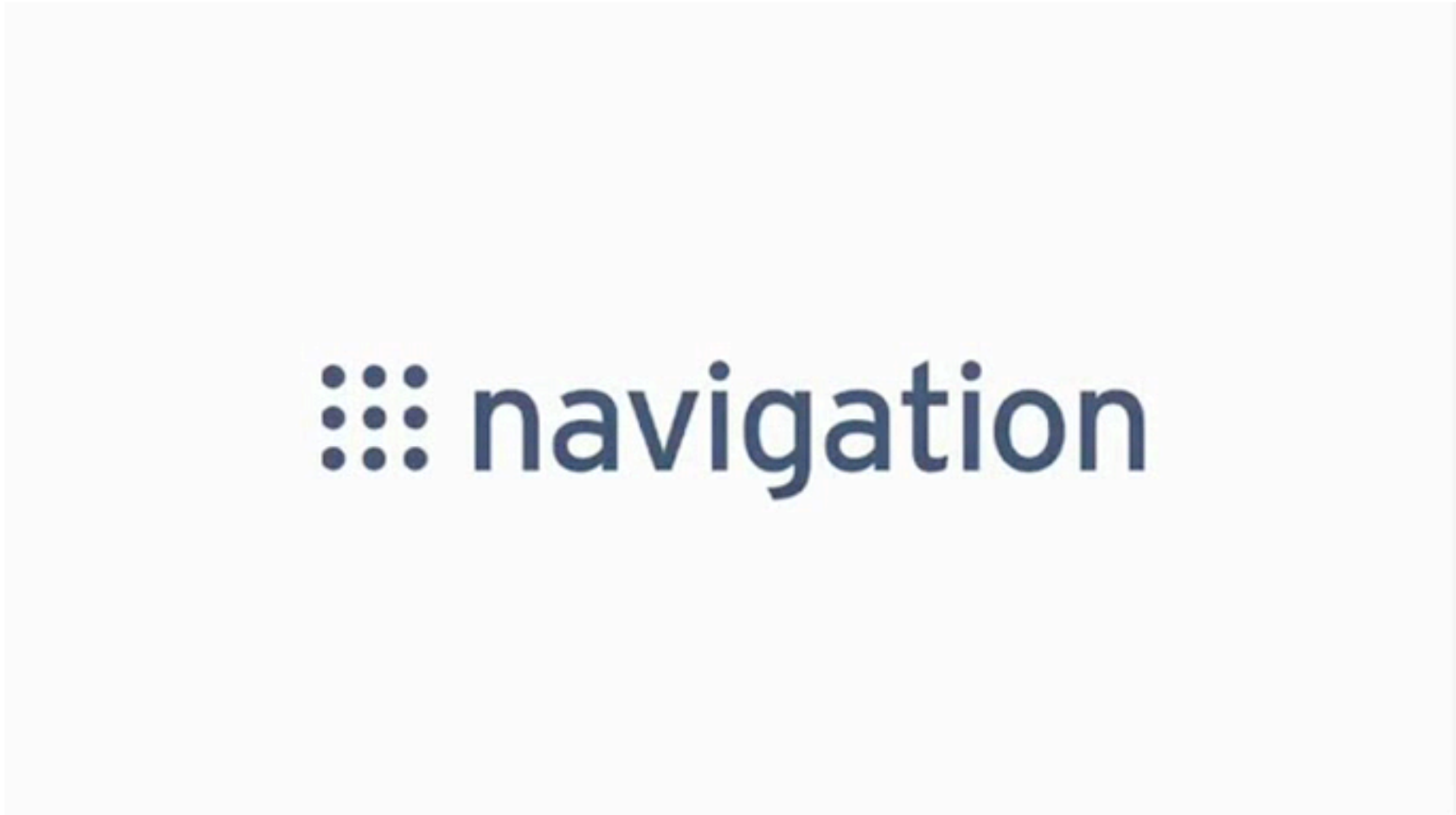
ROS: planning

MoveIt! is state of the art software for mobile manipulation, incorporating the latest advances in motion planning, manipulation, 3D perception, kinematics, control and navigation.

http://moveit.ros.org/documentation/concepts/

ROS: navigation

navigation: A 2D navigation stack that takes in information from odometry, sensor streams, and a goal pose and outputs safe velocity commands that are sent to a mobile base.

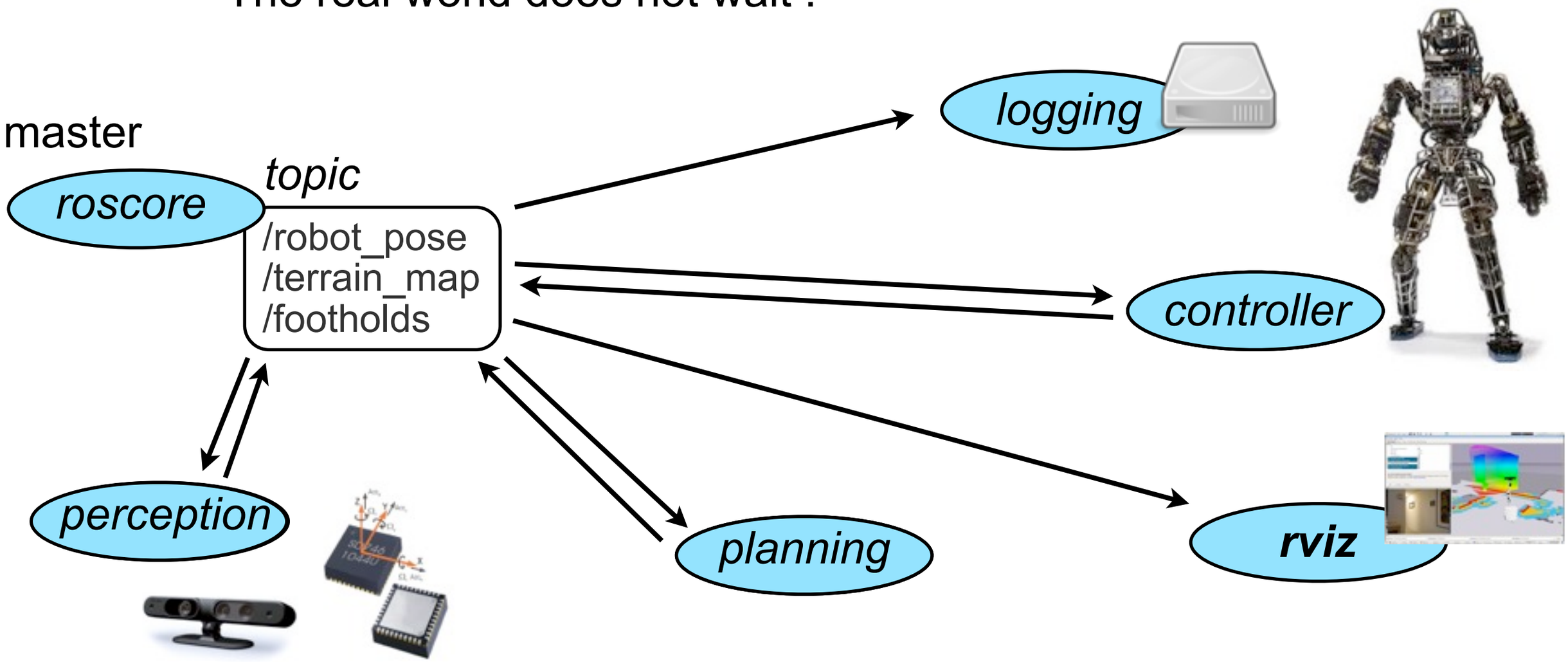


http://www.ros.org/wiki/navigation

System integration

Problem: Real-Time requirements

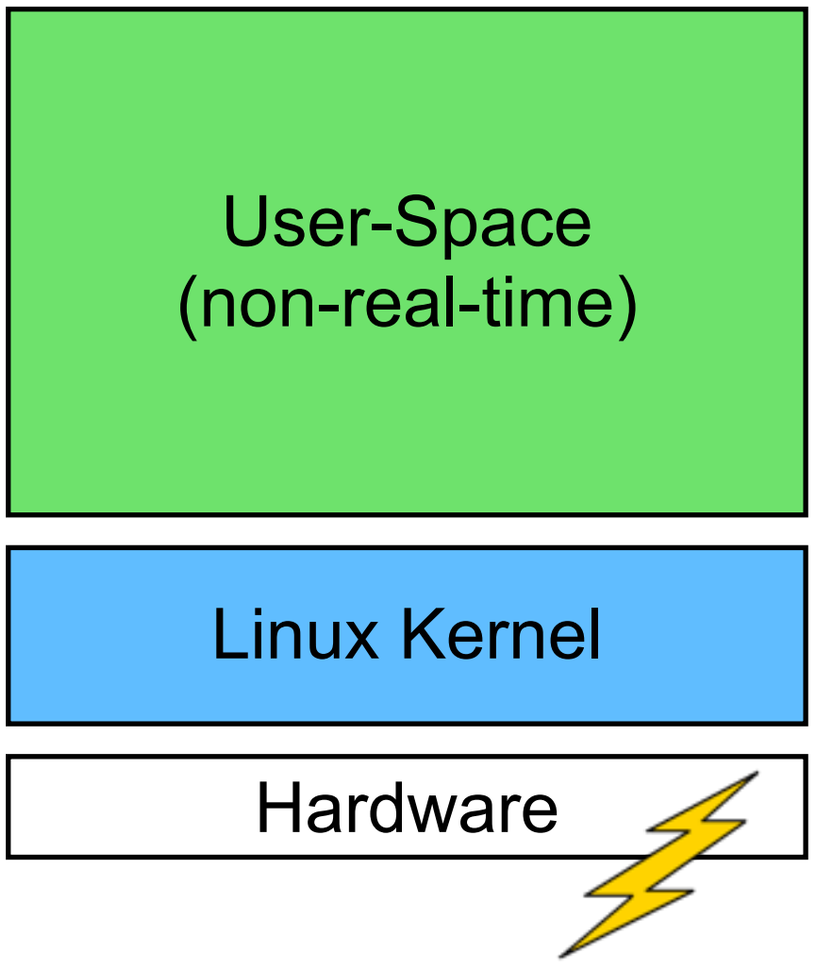
The real world does not wait !



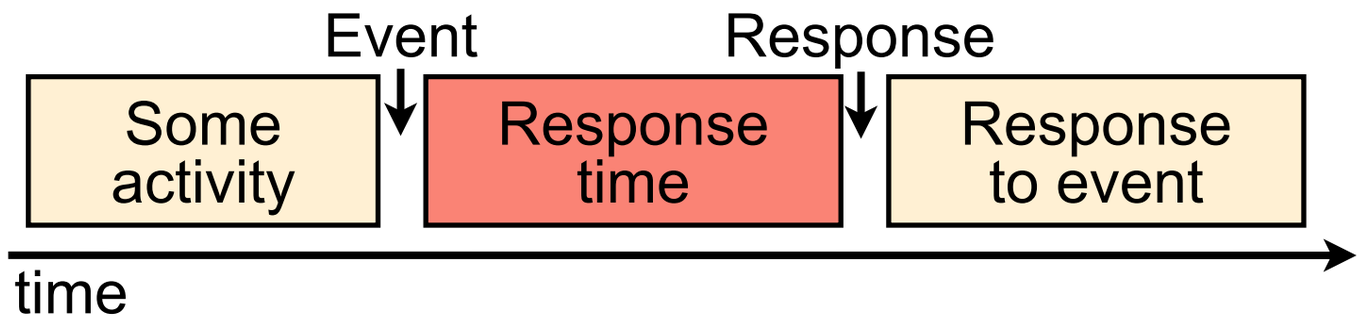
ROS: realtime

Real-time processes can guarantee response within strict time constraints.

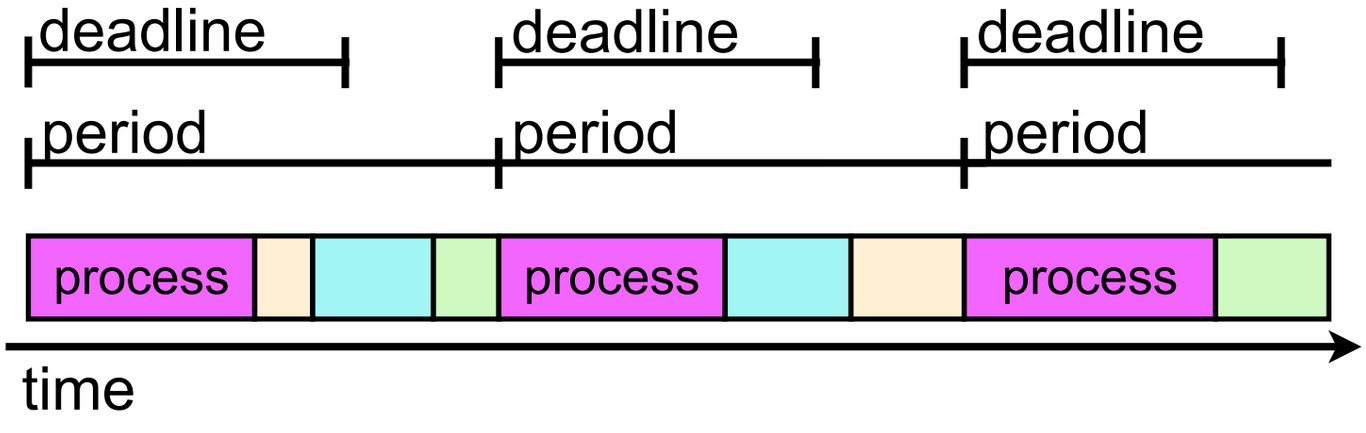
Linux



Interrupt latency (sensor processing)



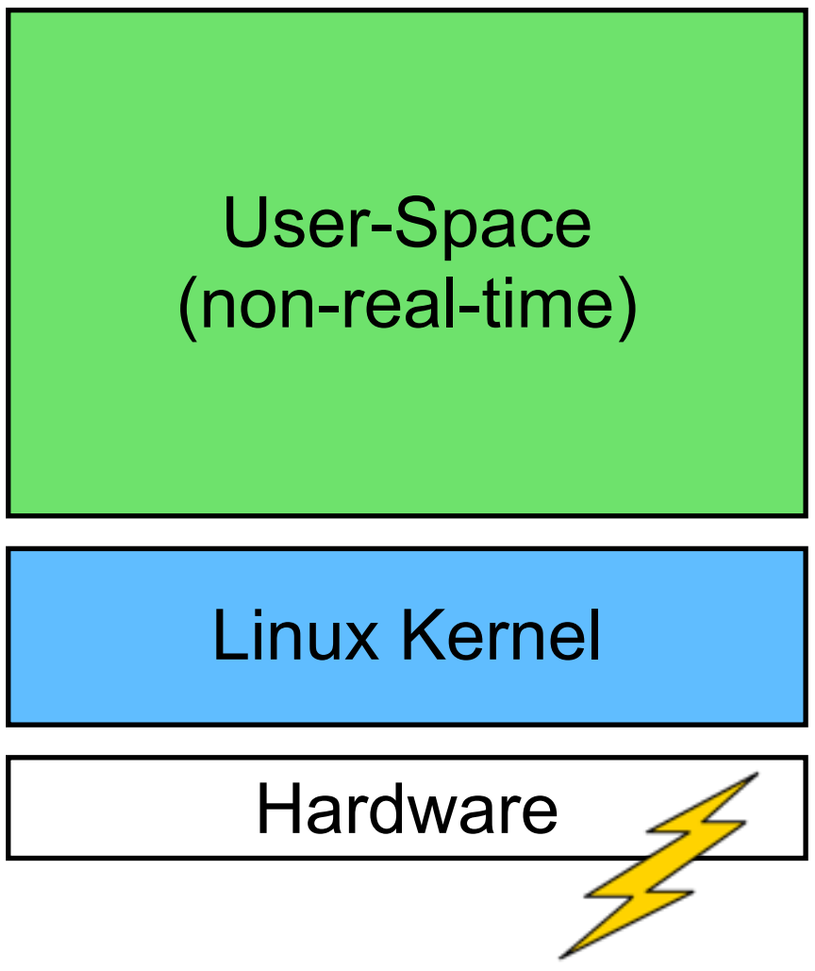
Task scheduling (control loop)



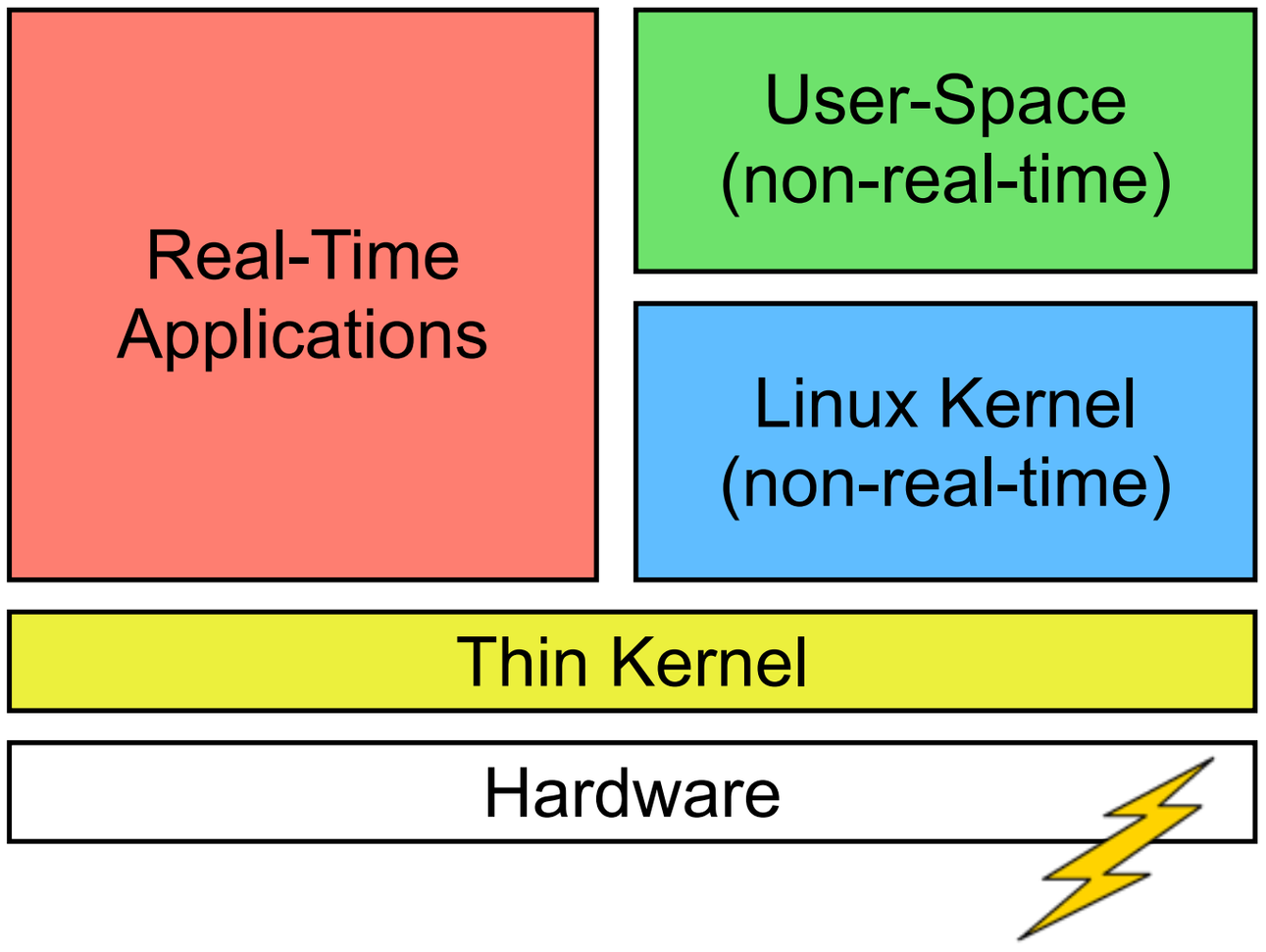
ROS: realtime

Real-time processes can guarantee response within strict time constraints.

Linux



Linux + Xenomai (http://xenomai.org)

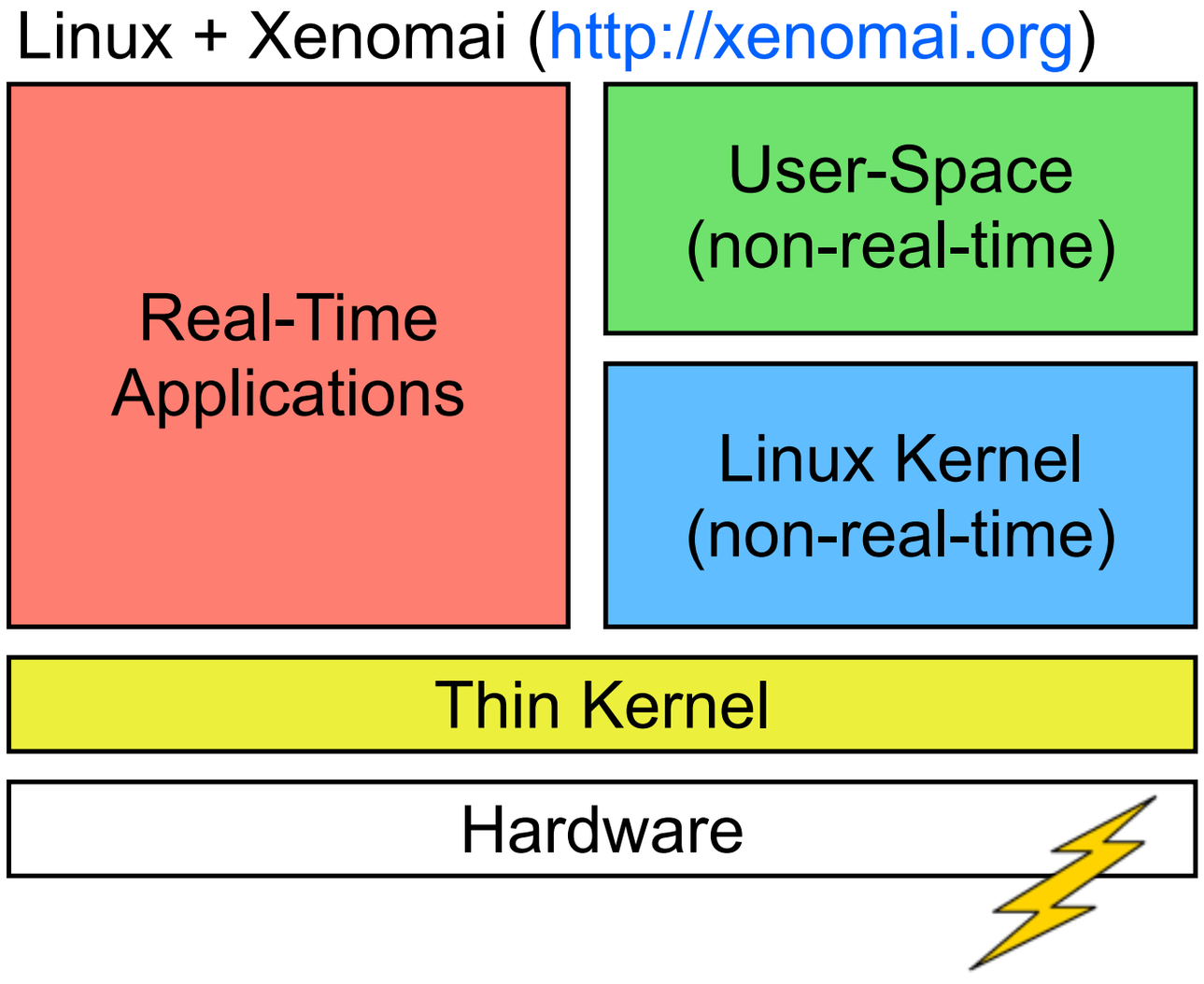


ROS: realtime

Real-time processes can guarantee response within strict time constraints.

Real-time programming requires special care:
No linux system calls (memory allocation, printf, ...)

Interface between real-time and non-real-time processes using `rosrt`.
http://wiki.ros.org/rosrt



Gazebo: simulation

Gazebo is a open source robot simulator. Simulation avoids risk of breaking real robots, reduces debugging cycles, allows to control the environment.

Simulating contacts however is problematic.

http://www.gazibosim.org

ROS: task executive

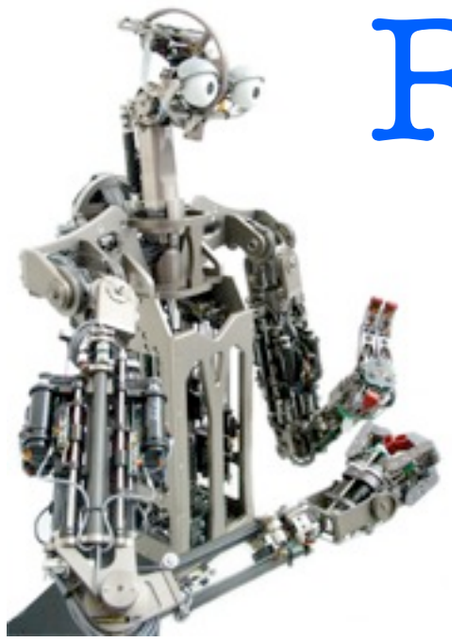
SMACH, which stands for 'state machine', is a task-level architecture for rapidly creating complex robot behavior.

http://www.ros.org/wiki/smach

Example application

poolshark

Overview



ROS

navigation

task executive

visualization

simulation

control

planning

...and more !

message passing

device

real-time capabilities

OS

web browser

networking

window manager

memory management

process management

scheduler

device drivers

file system

Why should you use ROS ?

Build on top of existing software, make use of existing tools, and focus on your own research.

Provide the community your own work such that people can reproduce your experiments and build on top of it.

More information about ROS

List of courses on ROS

http://wiki.ros.org/Courses

Getting started:

http://wiki.ros.org/ROS/StartGuide