TALOS Model/Simulation/ros_control

Training

Outline

- **TALOS robot model**
  Description of the robot, conventions and tools for describing it.
- **Gazebo simulator**
  Description of the simulator, characteristics and how do we use to simulate TALOS
- **ros_control**
  The framework to write real-time controllers for the robot that is agnostic to the real robot or the simulation

ROS URDF

Universal Robotic Description Format

- URDF is a description language that consists in a set of XML specifications for robot models, sensors, scenes, etc.

http://wiki.ros.org/urdf

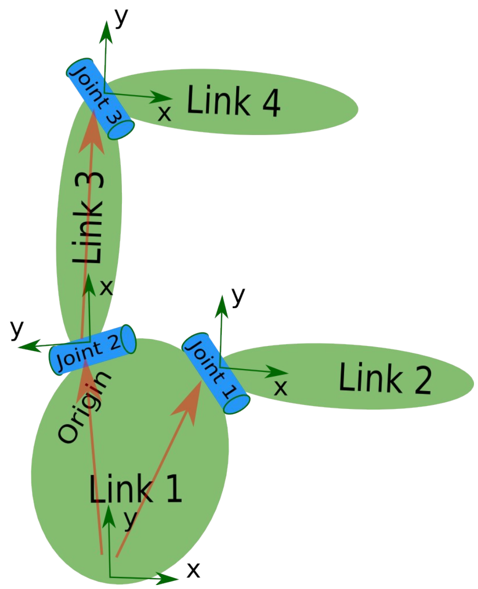
Xacro

- Xacro is an XML macro language. With xacro, you can construct shorter and more readable XML files by using macros that expand to larger XML expressions.

http://wiki.ros.org/xacro

Simple example of a kinematic tree:

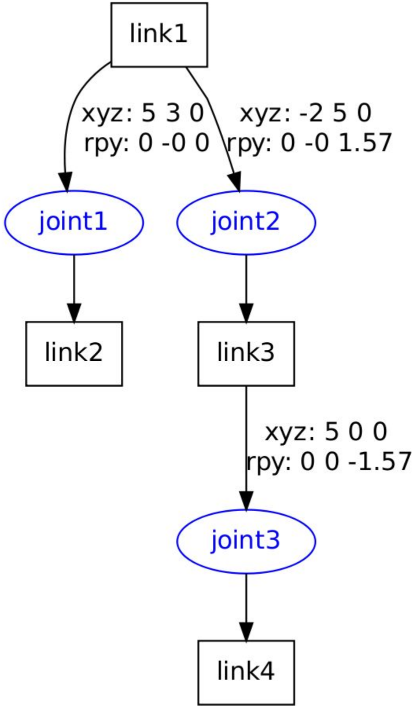
Robot description (URDF)




```xml
<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="link4"/>
  </joint>
</robot>
```

PAL ROBOTICS

TALOS T.S.




```xml
<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"/>
    <origin xyz="5 3 0" rpy="0 0 0" />
    <axis xyz="-0.9 0.15 0" />
  </joint>

  <joint name="joint2" type="continuous">
    <parent link="link1"/>
    <child link="link3"/>
    <origin xyz="-2 5 0" rpy="0 0 1.57" />
    <axis xyz="-0.707 0.707 0" />
  </joint>

  <joint name="joint3" type="continuous">
    <parent link="link3"/>
    <child link="link4"/>
    <origin xyz="5 0 0" rpy="0 0 -1.57" />
    <axis xyz="0.707 -0.707 0" />
  </joint>
</robot>
```

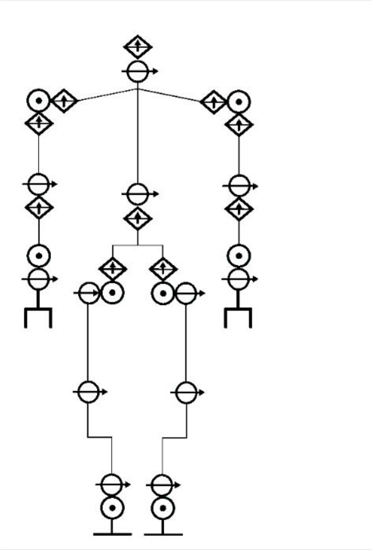
TALOS Description Files

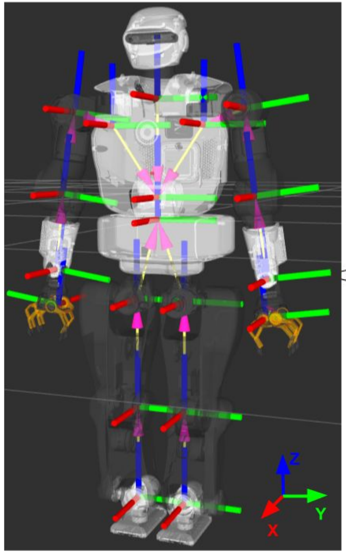
package://talos_description

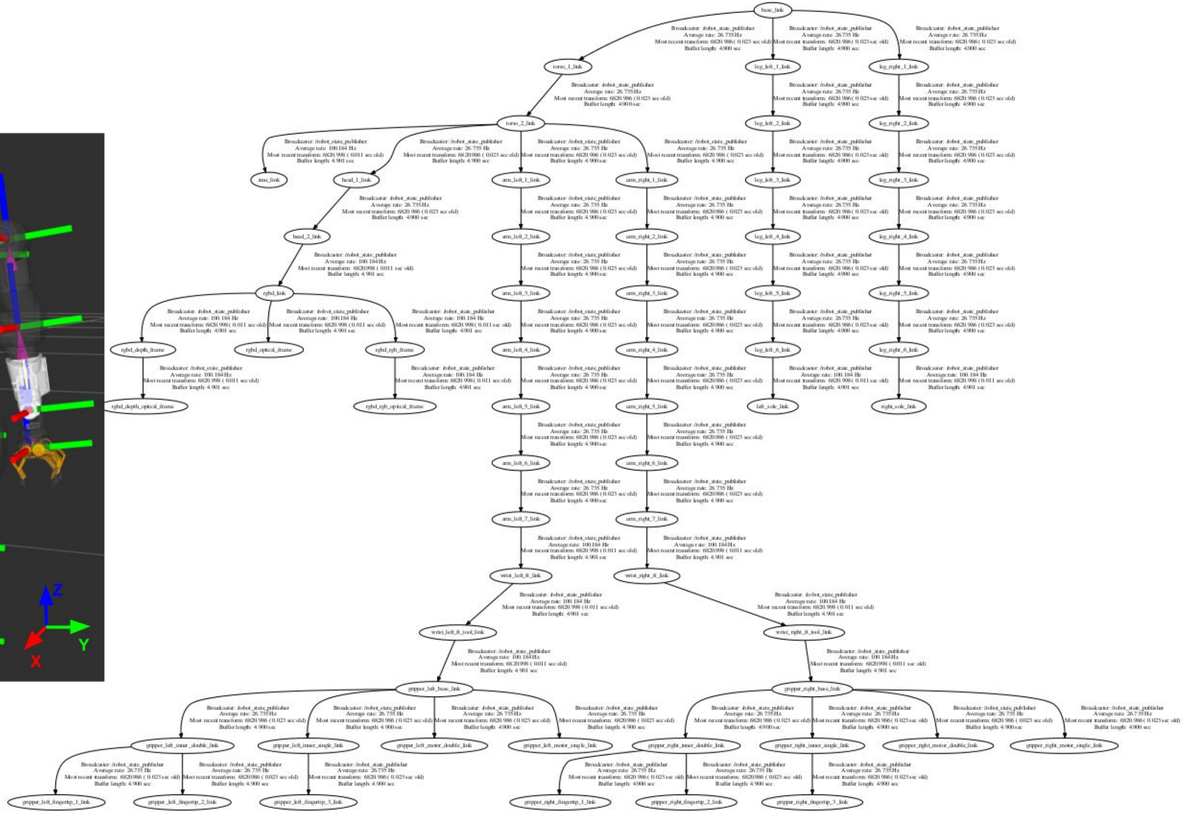
File structure for TALOS Description

```
urdf/
  torso/torso.urdf.xacro
  gripper/gripper.urdf.xacro
  leg/leg.urdf.xacro
  arm/arm.urdf.xacro
  sensors/
    Description of the sensors links, and characteristics for simulation
meshes/
  Contains all the visualization and collision meshes of the robot
robot/
  talos_full_no_grippers.urdf.xacro
  talos_full_v1.urdf.xacro
  talos_full_v2.urdf.xacro
  talos_lower_body.urdf.xacro
gazebo/
  gazebo.urdf.xacro
```

TALOS Kinematic Tree







Why do we need a unified description language?

- Have a unified language to describe robots
- Realistic simulation of the robot physics
- Have all the software components that rely on the robot description use single entry point that guarantees consistency between all of them http://wiki.ros.org/urdf_parser
- Allow the developer to easily access the robot description
- Already existing kinematic and dynamic libraries like KDL (Kinematic and Dynamics Library) already have built in support to parse URDF.

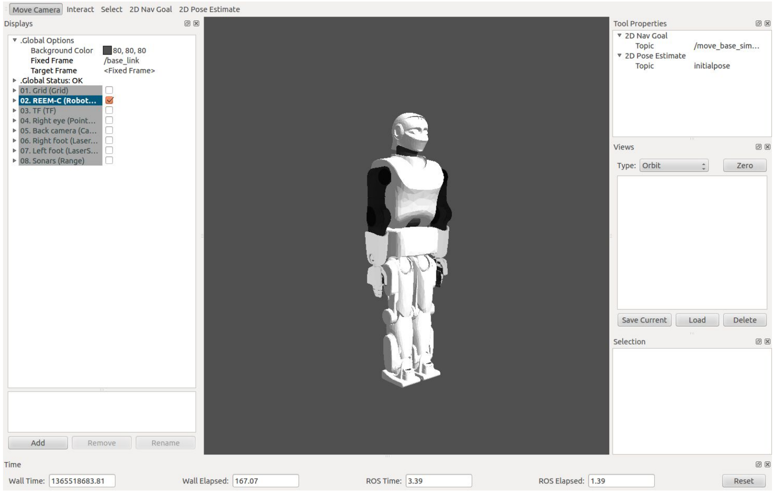
RVIZ (Robot Visualization)

Rviz lets you visualize the state of the robot, sensors, and extra relevant information to the user

The status of the robot is defined by the its joint configuration.
To visualize the robot we need to perform *Forward Kinematics* using the robot description and its actual joint configuration.

The robot state publisher takes care of this

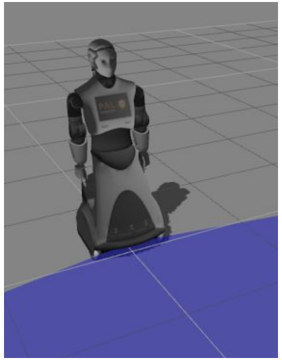
http://wiki.ros.org/robot_state_publisher

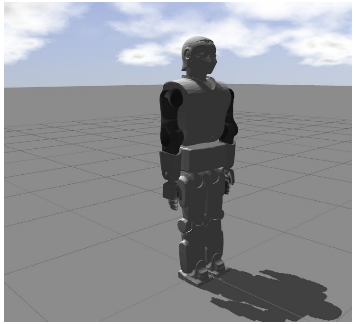


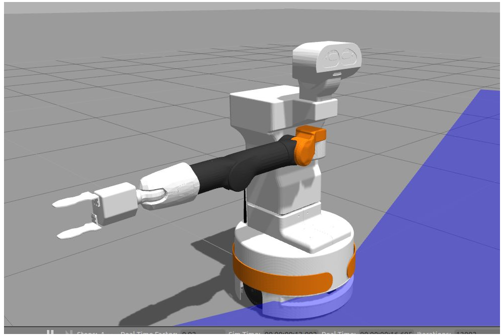
http://wiki.ros.org/rviz

GAZEBO

- Simulation for Robots:
  Towards accurate physical simulation
- Easy transition to and from simulation
  Remove hardware issues and resource constraints
- Support common robot control software
  Custom client code

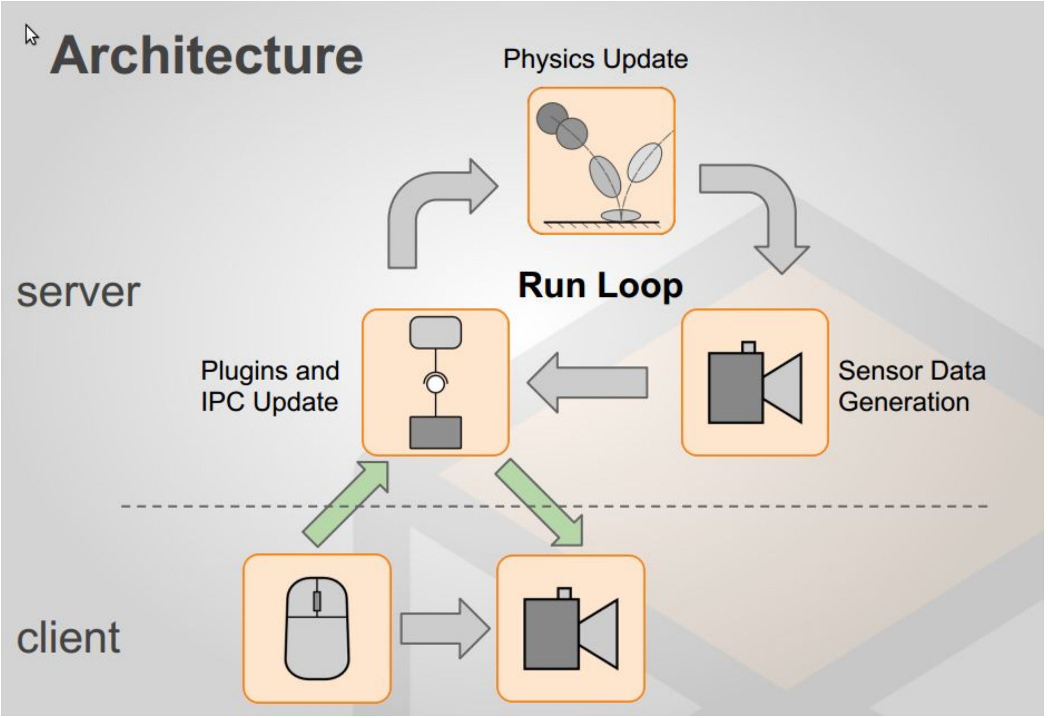




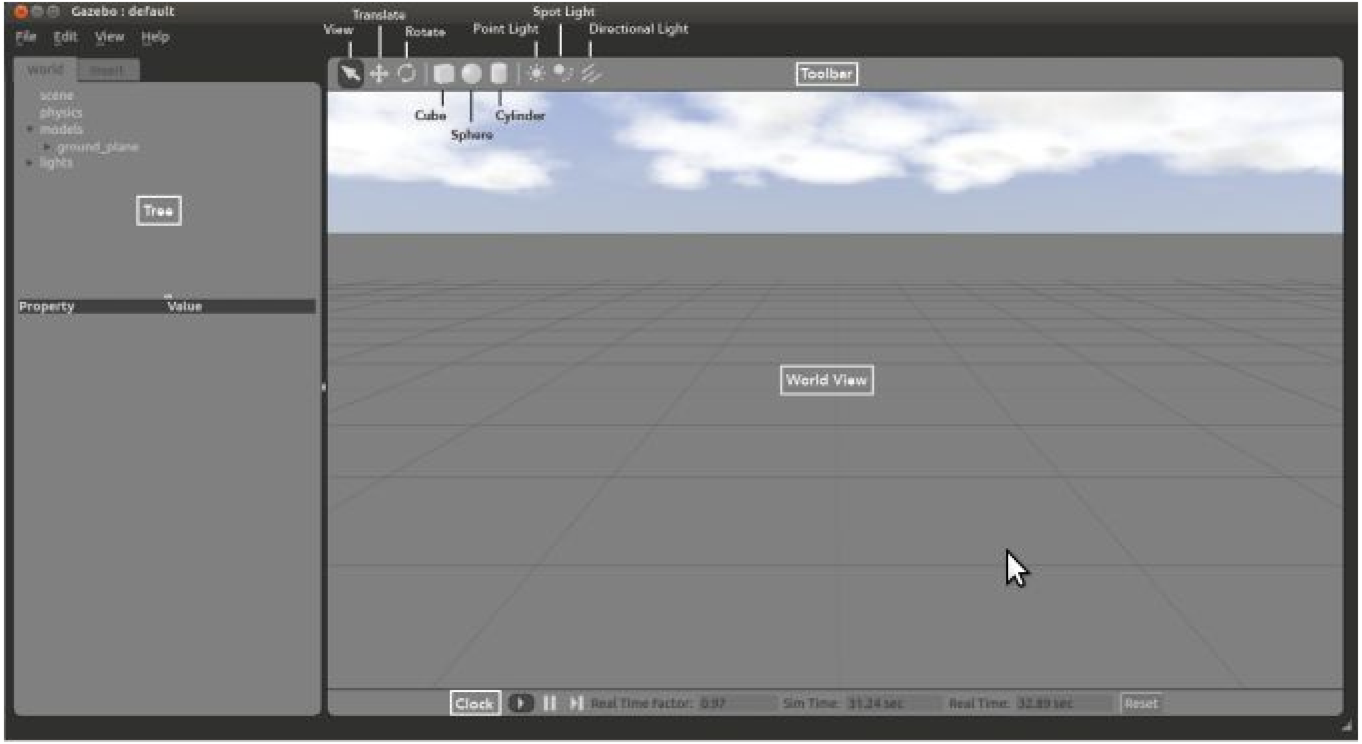


PAL ROBOTICS

Gazebo structure



Gazebo window



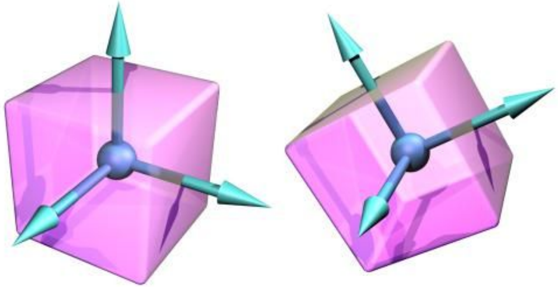
OPEN DYNAMICS ENGINE™

Open Dynamics Engine (**ODE**) will be the physics engine used for simulation

Its an open source, high performance library for simulating rigid body dynamics. It is fully featured, stable, mature and platform independent with an easy to use C/C++ API. It has advanced joint types and integrated collision detection with friction. ODE is useful for simulating vehicles, objects in virtual reality environments and virtual creatures.
It is currently used in many computer games, 3D authoring tools and simulation tools.

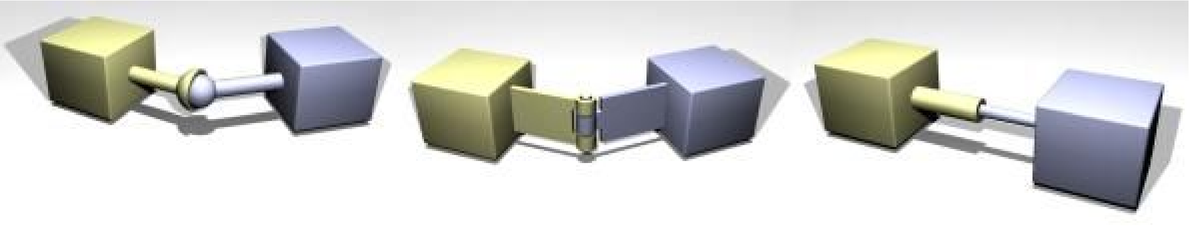
How does ODE do physics?

- Kinematics constraints
- Collision and contact constraints
- Rigid body dynamics



ODE's constraint solver uses a full coordinate system approach and enforces joint and contact constraints as posed by the linear complementarity problem

Joints and constraints in ODE



3 of the more common types of joints supported in ODE

- *ball socket*: constraints one body to be in the same location as the "socket" of another body.
- *hinge joint*: constraints the two parts of the hinge to be in the same location and to line up along the hinge axle.
- *prismatic*: constraints the "piston" and "socket" to line up, and additionally constraints the two bodies to have the same orientation.

All the joints in TALOS are revolute joints, thus we will only use the hinge joints, and simulate a pid controller for every joint inside the simulator to achieve position control.

PAL ROBOTICS

Sensors simulation in gazebo

- Cameras
- IMU (Inertial measuring unit)
- Force torque sensors

TALOS T.S.

PAL ROBOTICS

Launching TALOS in gazebo

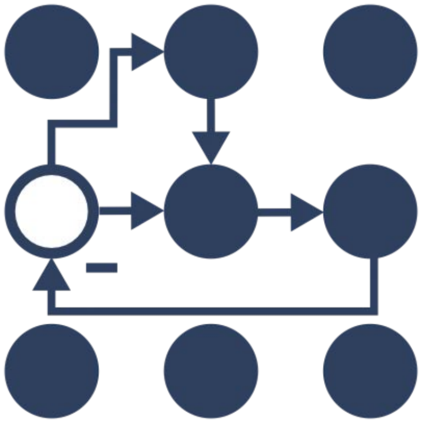
Gazebo can be started with the TALOS robot spawned using this comands:

- TALOS in a house map world
  ros_launch talos_gazebo talos_gazebo.launch

- Only TALOS in an empty world
  roslaunch talos_gazebo talos_gazebo.launch world:=empty

By default no controllers will be active except the ones that publish the state of the robot, joint state publisher, force publisher and IMU publisher
(The sames ones as in the real robot)

ROS_control



https://github.com/ros-controls/ros_control/wiki

http://wiki.ros.org/ros_control

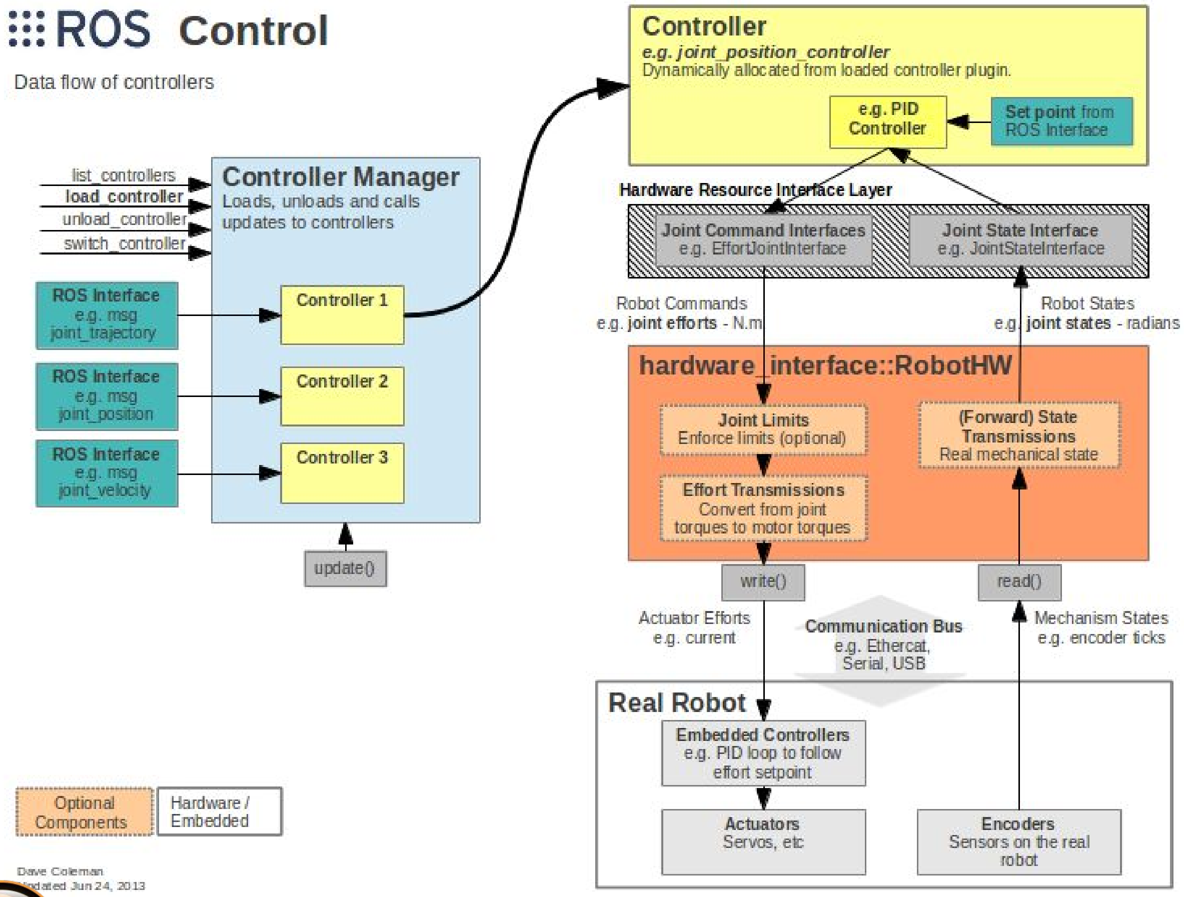
ROS_control

Motivation

- Is a consistent interface to access the actuators and sensors of a robot.
- RAW data that comes from hardware components to the various controllers in an organised and intuitive way
- It also has the function of being a resource handler for all sensors and actuators allowing different controllers to access the same device simultaneously using custom policies
- Is an abstraction layer for hardware it allows to write controllers that are robot agnostic and can be reused in different types of robots with different physical configurations

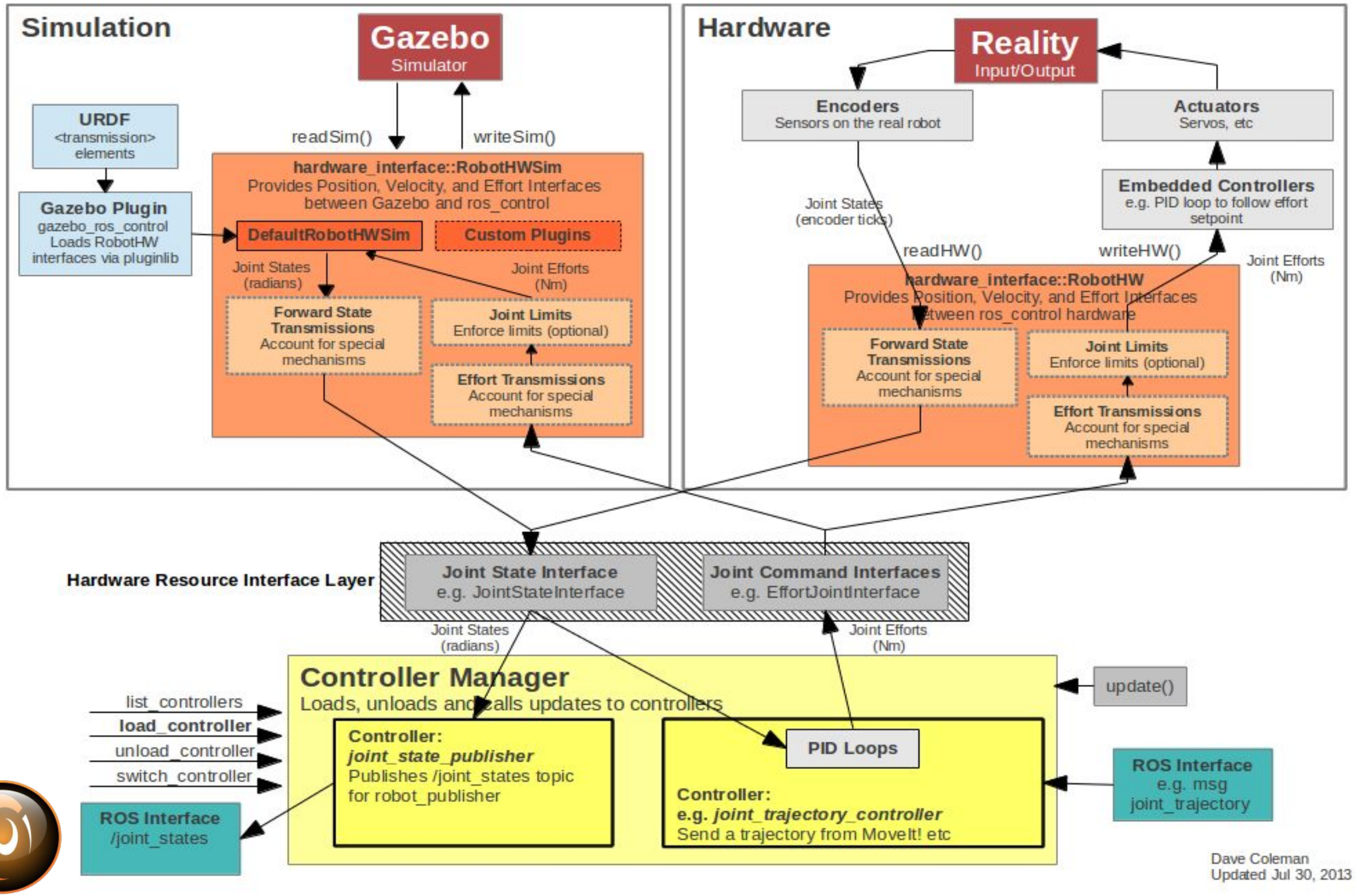
PAL ROBOTICS

TALOS T.S.


ROS Control
Data flow of controllers
list_controllers
load_controller
unload_controller
switch_controller
Controller Manager
Loads, unloads and calls updates to controllers
ROS Interface
e.g. msg
joint_trajectory
ROS Interface
e.g. msg
joint_position
ROS Interface
e.g. msg
joint_velocity
Controller 1
Controller 2
Controller 3
update()
Controller
e.g. joint_position_controller
Dynamically allocated from loaded controller plugin.
e.g. PID Controller
Set point from ROS Interface
Hardware Resource Interface Layer
Joint Command Interfaces
e.g. EffortJointInterface
Joint State Interface
e.g. JointStateInterface
Robot Commands
e.g. joint efforts - N.m
Robot States
e.g. joint states - radians
hardware_interface::RobotHW
Joint Limits
Enforce limits (optional)
Effort Transmissions
Convert from joint torques to motor torques
(Forward) State Transmissions
Real mechanical state
write()
read()
Actuator Efforts
e.g. current
Communication Bus
e.g. Ethercat, Serial, USB
Mechanism States
e.g. encoder ticks
Real Robot
Embedded Controllers
e.g. PID loop to follow effort setpoint
Actuators
Servos, etc
Encoders
Sensors on the real robot
Optional Components
Hardware / Embedded
Dave Coleman
Updated Jun 24, 2013

TALOS T.S.

GAZEBO + ROS + ros_control



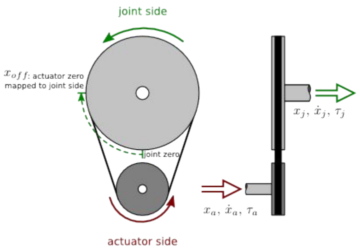
PAL ROBOTICS

TALOS T.S.

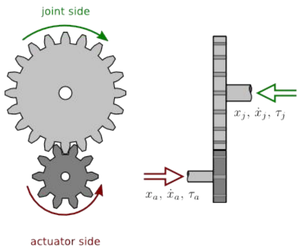
Transmission interfaces

Ros control abstract all the transmission from the real robot hardware in order to present only joints to the user

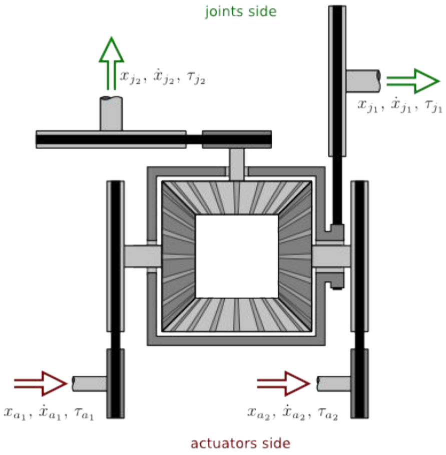
- Gear transmissions
- Pulleys
- Differentials
- Four bar linkage



Timing belt



Gears



TALOS T.S.

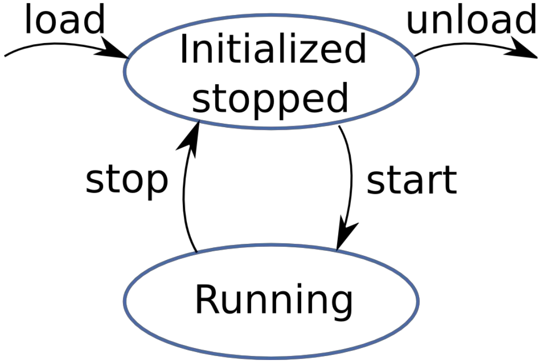
PAL ROBOTICS

Controller State Machine

Every controller has a state machine, the states can be described by:

Presence

- *Loaded*: All controller specific configurations are loaded in the parameter server and \Verb|ros_control| has created an instance of the controller that is in initialised state.
- *Not loaded*: \Verb|ros_control| has knowledge of the controller type, but no instance of the controller was created.



Activity

- *Running*: The update function of the controller is called at every control cycle.
- *Stopped*: The controller is initialized but is not performing any action.

Controller_manager

The main hub of ROS_control is the \Verb|controller_manager|.
The controller manager exposes its interface through ROS services.
It's responsible for managing all the controllers, resources of the robot and triggering events.

How to list available controllers
rosservice call /controller_manager/load_controller

How to list controllers status
rosservice call /controller_manager/list_controllers

How to load and unload controllers
rosservice call /controller_manager/load_controller "name: 'hello_controller'"
rosservice call /controller_manager/unload_controller "name: 'hello_controller'"

How to start and stop controllers
rosservice call /controller_manager/switch_controller "start_controllers: - 'hello_controller' stop_controllers: - 'imu_controller' strictness: 0"

PAL ROBOTICS

Requirements to create ros_control plugin

- Create a class that inherits from controller_interface::Controller
- Add an xml file in the package that belongs to the controller that will allow ros_control to now its existence
- Have a .yaml file that specifies the name and type of the controller (In order to allow ros_control to know what parameters to load for the controller)

Why do we need Real time?

- We need to guarantee determinism of our algorithms
  - To send periodic continuous periodic commands to the motors
  - To have deterministic integration of time inside the algorithms
  - To react to unforeseen events in deterministic allowed times

Real time considerations

- We use a Linux kernel with PREEMPT_RT patch to have real time guarantees for our controllers. No system calls or allocations can be done inside this code

Controllers by example

hello_controller.cpp

```
#include <controller_interface/controller.h>
#include <hardware_interface/joint_state_interface.h>
#include <pluginlib/class_list_macros.h>

namespace ros_controllers_tutorials{

class HelloController
  : public controller_interface::Controller<hardware_interface::JointStateInterface>
{
 public:
  bool init(hardware_interface::JointStateInterface* hw, ros::NodeHandle &n)
  {
        return true;
  }
  void update(const ros::Time& time, const ros::Duration& period)
  {
        // WARNING! This will not be realtime-safe
        ROS_INFO_NAMED("hello_controller", "Hello ros_control!");
  }
  void starting(const ros::Time& time) { }
  void stopping(const ros::Time& time) { }
private:
};
PLUGINLIB_DECLARE_CLASS(ros_controllers_tutorials, HelloController,
                        ros_controllers_tutorials::HelloController, controller_interface::ControllerBase);
}
```

PAL ROBOTICS

TALOS T.S.

ros_controllers_tutorials.xml

```xml
<class name="ros_controllers_tutorials/HelloController"
 type="ros_controllers_tutorials::HelloController"
        base_class_type="controller_interface::ControllerBase">
        <description>
        The HelloController does nothing useful.
        It's only a demonstrational controller that prints a hello message.
        </description>
  </class>
```

hello_controller.yaml

```yaml
hello_controller:
  type: "ros_controllers_tutorials/HelloController"
```

TALOS T.S.

PAL ROBOTICS

Joint controller

```
[...]
namespace ros_controllers_tutorials{
  class JointController : public controller_interface::Controller<hardware_interface::PositionJointInterface>
  {
  public:
      bool init(hardware_interface::PositionJointInterface* hw, ros::NodeHandle &n){
        cont = 0;
        controlled_joint_name_ = "head_2_joint"; // loading joint controller
        try{
          joint_ = hw->getHandle(controlled_joint_name_); // Get a joint handle,  throws on failure
        }
        catch (...){  [...] }

      void update(const ros::Time& time, const ros::Duration& period){
        double joint_comand = 0.2*sin(cont/1000.0); //Move the head using a sine wave
        joint_.setCommand(joint_comand);
        cont = cont + 1;
      }
[...]
  private:
      hardware_interface::JointHandle joint_;
      std::string controlled_joint_name_;
      double cont;
[...]
```

PAL ROBOTICS

TALOS T.S.

Read Inertial Measurement Unit

```
[...]
  class ImuController :  public controller_interface::Controller<hardware_interface::ImuSensorInterface>   {
  public:
          bool init(hardware_interface::ImuSensorInterface* hw, ros::NodeHandle &n) {
                  const std::vector<std::string>& sensor_names = hw->getNames(); // get all imu sensor names

            [...]
            sensor_ = hw->getHandle(sensor_names[i]);
            [...]
          }
          void update(const ros::Time& time, const ros::Duration& period)
          {
            using namespace hardware_interface;
            sensor_msgs::Imu value;
            if (sensor_.getOrientation())
            {
              value.orientation.x = sensor_.getOrientation()[0];
              value.orientation.y = sensor_.getOrientation()[1];
              value.orientation.z = sensor_.getOrientation()[2];
              value.orientation.w = sensor_.getOrientation()[3];
            }
          }
          [...]
  private:
          hardware_interface::ImuSensorHandle sensor_;
```

PAL ROBOTICS

Combined resources

```
class CombinedResourceController : public controller_interface::ControllerBase{
 public:
          bool initRequest(hardware_interface::RobotHW* robot_hw,  ros::NodeHandle& root_nh,
                                   ros::NodeHandle& controller_nh,  std::set<std::string>&  claimed_resources){
            [...]
            PositionJointInterface* pos_iface = robot_hw->get<PositionJointInterface>(); // Get a pointer to the joint position control interface
            [...]
            ForceTorqueSensorInterface* ft_iface = robot_hw->get<ForceTorqueSensorInterface>(); // pointer to the force-torque sensor interface
            [...]
            ImuSensorInterface* imu_iface = robot_hw->get<ImuSensorInterface>(); // Get a pointer to the IMU sensor interface
            [...]
          }
          bool init(PositionJointInterface* pos_iface, ForceTorqueSensorInterface* ft_iface, ImuSensorInterface* imu_iface,
                     ros::NodeHandle&  root_nh, ros::NodeHandle& controller_nh)
          { ...  }
          bool initJoints(PositionJointInterface* pos_iface, ros::NodeHandle&  controller_nh)
          { ...  }
          bool initForceTorqueSensors(ForceTorqueSensorInterface* ft_iface, ros::NodeHandle& controller_nh)
          { ...  }
          bool initImuSensors(ImuSensorInterface* imu_iface, ros::NodeHandle& controller_nh)
          { ...  }
```

PAL ROBOTICS

TALOS T.S.

Combined resources

```
void update(const ros::Time& time, const ros::Duration& period){
        geometry_msgs::Wrench ft[2]; //F-T sensor
        for(unsigned int s = 0; s<2; ++s){
          ft[s].force.x  = ft_sensors_[s].getForce()[0];
          ft[s].force.y  = ft_sensors_[s].getForce()[1];
          ft[s].force.z   = ft_sensors_[s].getForce()[2];
          ft[s].torque.x = ft_sensors_[s].getTorque()[0];
          ft[s].torque.y = ft_sensors_[s].getTorque()[1];
          ft[s].torque.z = ft_sensors_[s].getTorque()[2];
        }
        sensor_msgs::Imu value;
        if (imu_sensor.getOrientation())
        {
        value.orientation.x = imu_sensor.getOrientation()[0];
        value.orientation.y = imu_sensor.getOrientation()[1];
        value.orientation.z = imu_sensor.getOrientation()[2];
        value.orientation.w = imu_sensor.getOrientation()[3];
        }
        std::string getHardwareInterfaceType() const
        {return hardware_interface::internal::demangledTypeName<hardware_interface::PositionJointInterface>();
}
  private:
        std::vector<hardware_interface::JointHandle> joints_;
        hardware_interface::ImuSensorHandle imu_sensor;
        std::vector<hardware_interface::ForceTorqueSensorHandle> ft_sensors_;
```

PAL ROBOTICS