1 | Migration to MuJoCo physics simulator

Because of limitations of the current Gazebo-Robot Operating System (ROS) integration, new simulation engines are explored. The problem which occurred is posted on the ROS Answers forum [?]. The quote below describes the problem:

> I am trying to use an Effort controller called 'JointTrajectoryController' in order to have the robot arm pass through certain joint angle way-points. Although it runs with this controller for 4 seconds (Simulation time), the Gazebo model then breaks down and the joint states report NaN values for all the joints.

Unfortunately, no solutions have been found. However, several other examples of this problem have been reported on the forum. This suggests the problem is not easily solvable and is inherently caused by Gazebo. The problem is also video recorded and can be found by here.
When the `JointTrajectoryController` is used, the engine crashes after a certain amount of time. Since the trajectory following controller is needed for simulating the Optimal Control Problem (OCP) trajectory of the Mobile-Manipulator (MM), Gazebo cannot be used for this as long as the problem cannot be solved.

The physics engine which is going to be used is called Multi-Joint dynamics with Contact (MuJoCo). A start has already been made to implement the physics engine for simulations of the MM. A custom viewer has been created to have camera views of the cameras mounted on the Mobile-Base (MB) and Robot Manipulator Arm (RMA). Furthermore, XML files are made, which describe the model of the MM. This is the current state of the MuJoCo physics engine used in the Universtiy of Waterloo Advanced Robotics Lab (UWARL).

Firstly, a clock publisher is created to be able to synchronize all nodes. This is done by using the `time.time()` command to get current wall-time. Every iteration of the clock, the difference in wall-time compared to the start of the node is computed and published. This results in a clock time starting at 0 and being published at a frequency of approximately 1400 Hz. This high frequency is needed to get a sufficient resolution of time used by the ROS-nodes.

Running the MuJoCo simulator in a ROS-node is the second objective. To make sure MuJoCo is updating in real-time, it is synced to step forward in time exactly the same amount of time as the node is progressing in time. This way, the ROS time which is used in all nodes connected to the ROS-core matches the simulation time in MuJoCo.
Since the computation time of the MuJoCo engine for stepping forward in time is not constant, especially when high forces are applied or a lot of contact are present, it can take longer than the time step in the ROS-node. This means the node will not keep its desired frequency and will lag behind compared to the published clock which is used by all the controls. When the node lags behind, the MuJoCo engine will lag behind to, since it is updated every iteration of the node. This problem is shown in Figure 1.1. In this figure, the short thick vertical lines represent the end of a loop/iteration of the ROS-node that updates the engine. The thin, long vertical lines are plotted to compare against the real-time published by the clock publisher on the `/clock` topic.

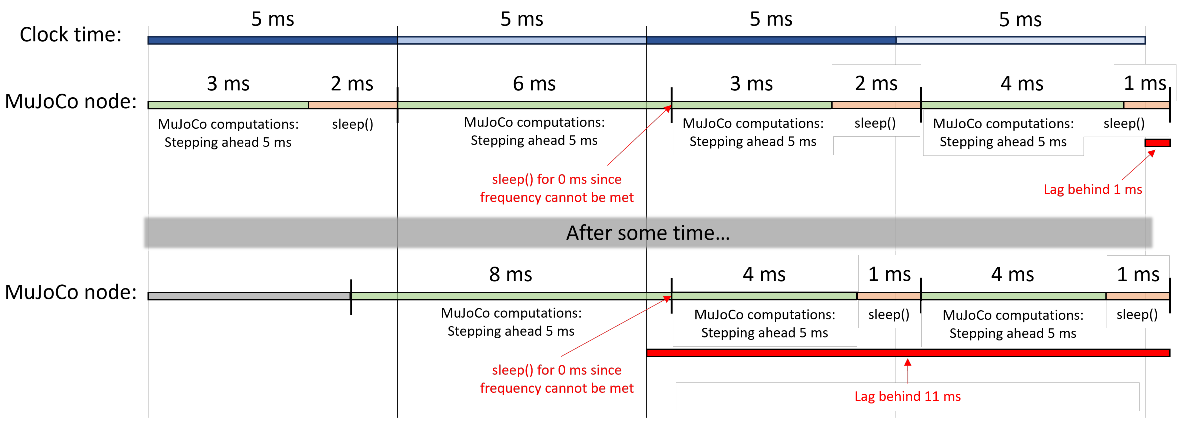

Figure 1.1: MuJoCo engine computation time syncing problem

An algorithm has been developed to speed up the node when it is lagging behind. This is done by having 1 step in 100 iterations of the node which is not regulated to a fixed frequency. This means that it can catch up gradually when needed.
The step time of the MuJoCo engine is 2.5 ms. The frequency of the ROS-node is 200 Hz. From this, it follows that the MuJoCo engine steps forward in time by 2 steps (5 ms) every iteration of the node, which is the same time step as the node. It is important that the nodes timestep is the same, or a multiple of the engines step time to enable this real time syncing.
Having 1 step not regulated every 100 iterations means that for a this iteration, the node updates the simulation without sleeping and thus progresses in time more than the real time does. When this is the case, and the `rospy.Rate.sleep()` command is not evaluated during one step, the next iteration the function will compare the current time to the last time the function was called to know how long it has to sleep. This last time was two iterations ago so it will not start sleeping when more than 5 ms have elapsed. It means that in practice, the function is skipped once, but the second time it encounters the sleep command, it will probably not sleep, thus not sleeping for two times in a row.

The simulation can theoretically catch up 10 ms every 100 iterations minus the time that has elapsed, which is minimally 5 ms. The time that is elapsed is minimally 5 ms, since the `rospy.Rate.sleep()` will sleep the second time when the computation time was less than 5 ms over the last two iterations. All in all, the simulation will therefore catch up by a maximum of 5 ms every 100 iterations, or a maximum of 10 ms every second.
In practice, this will probably be less. The algorithm makes sure the simulation time gradually merges with the published clock time. When the simulation time has merged to less than 10 ms difference with the clock time, the loop will be constantly regulated again by the `rospy.Rate.sleep()` function. The solution is shown in Figure 1.2.
This deregulation of frequency is only applied when simulation time is lagging behind by more than 10 ms to not overtake, and thus lead the clock-time. On the other hand, when when the computation time is to high, and the simulation is lagging behind constantly without catching up, the node's frequency should be scaled down. This decreases the accuracy of the simulation since control loops will run at a lower frequency, and the engine step time per iteration is increased.

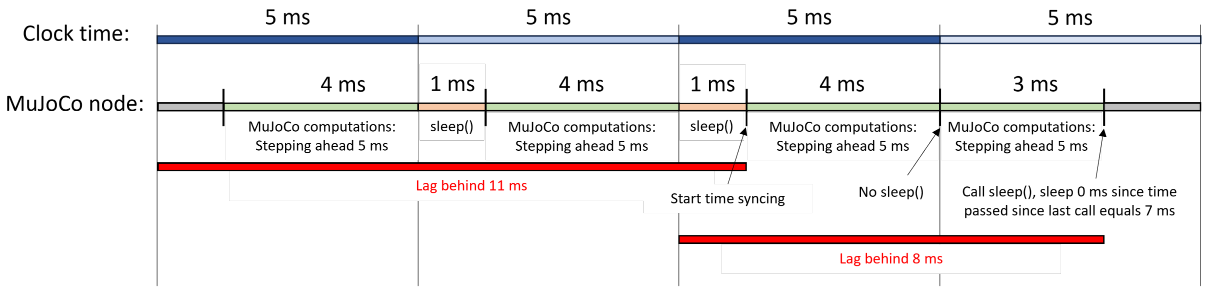

Figure 1.2: MuJoCo engine computation time syncing solution

Next, a ROS-node has been made to publish the `joint_states` and `link_states` which were normally published by the Gazebo engine. The pointer to the MuJoCo data is passed when initializing the publisher class. The data is read every iteration, transformed into the correct topic message formats and is published into their respective topics. For the parent links of the MM and the cart, the initial rotation is set to 0 to be independent on the orientation in which the model is spawned.

When the `joint_states` and `link_states` are being published, controllers can be implemented to control the joint angles for the RMA and the velocities for the MB. To do this, the `ros_controllers` package will be used again. Specifically, the `effort_controllers/JointTrajectoryController` and `effort_controllers/JointPositionController` will be used for the RMA. These ROS controllers communicate to the simulated robot via the Hardware Simulation Interface (HSI).
Figure 1.3 shows the framework which is used for the Gazebo-ROS integration. The controllers are shown in yellow. They communicate to a Hardware Resource Interface Layer, as shown in the figure. These interfaces, which are joint specific, communicate to a populated HSI. These are populated by the Gazebo `pluginlib`. Since Gazebo is not used, a new way of populating them has to be developed. Shadow Robot [2] has created a ROS-MuJoCo integration from which parts can be used to create these interfaces. This is done by reading the Unified Robot Description Format (URDF) files that describe the robot in a similar fashion as the Gazebo `pluginlib` does.

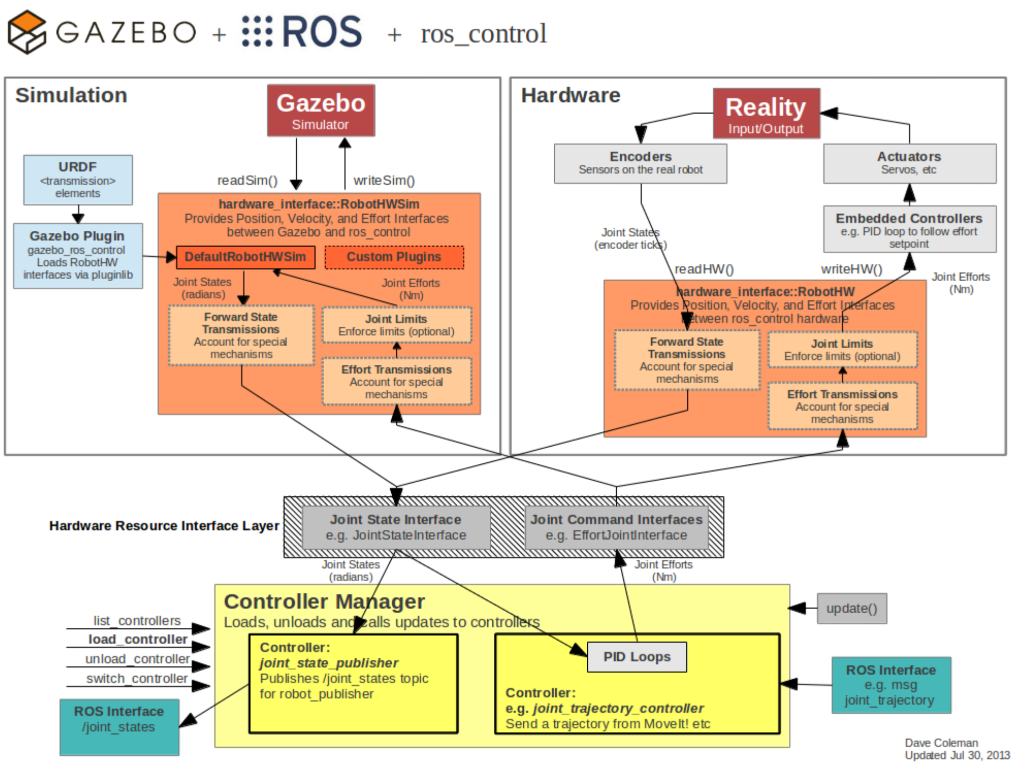
GAZEBO + ROS + ros_control
Simulation
Gazebo
Simulator
URDF
<transmission> elements
Gazebo Plugin
gazebo_ros_control
Loads RobotHW interfaces via pluginlib
readSim()
writeSim()
hardware_interface::RobotHWSim
Provides Position, Velocity, and Effort Interfaces between Gazebo and ros_control
DefaultRobotHWSim
Custom Plugins
Joint States (radians)
Joint Efforts (Nm)
Forward State Transmissions
Account for special mechanisms
Joint Limits
Enforce limits (optional)
Effort Transmissions
Account for special mechanisms
Hardware
Reality
Input/Output
Encoders
Sensors on the real robot
Actuators
Servos, etc
Embedded Controllers
e.g. PID loop to follow effort setpoint
Joint States (encoder ticks)
readHW()
writeHW()
Joint Efforts (Nm)
hardware_interface::RobotHW
Provides Position, Velocity, and Effort Interfaces between ros_control hardware
Forward State Transmissions
Account for special mechanisms
Joint Limits
Enforce limits (optional)
Effort Transmissions
Account for special mechanisms
Hardware Resource Interface Layer
Joint State Interface
e.g. JointStateInterface
Joint Command Interfaces
e.g. EffortJointInterface
Joint States (radians)
Joint Efforts (Nm)
Controller Manager
Loads, unloads and calls updates to controllers
update()
list_controllers
load_controller
unload_controller
switch_controller
ROS Interface
/joint_states
Controller:
joint_state_publisher
Publishes /joint_states topic for robot_publisher
PID Loops
Controller:
e.g. joint_trajectory_controller
Send a trajectory from MoveIt! etc
ROS Interface
e.g. msg
joint_trajectory
Dave Coleman
Updated Jul 30, 2013

Figure 1.3: Gazebo-ROS framework

Shadow Robot has developed a way to create the MuJoCo model, data, engine and viewer, and populate the HSI all in the same node, which enables them to access the MuJoCo data in the ROS-node, in which the HSI data are also available.
This is not the case in the current configuration of MuJoCo which is used by the UWARL. MuJoCo is launched via a Python wrapper of the MuJoCo C++ source code, by using `ctypes`. This configuration is chosen, since it is much simpler and is easier to connect with other Python classes from other Python ROS nodes. It is preferred to have as much parts of the simulation in Python. The downside of this is that it is hard to pass MuJoCo data to C++ Shadow Robot scripts because of the difference in language. To be able to communicate between the two languages, ROS topics are used. This is considered to be the

easiest way and enables access in both Python and C++ nodes. Research has been done to create custom wrappers to pass the data via functions wrapping the C++ data. However, since this is out of the scope of the project it is considered to be too complicated.

Figure 1.4 shows the configuration which has been created for the ROS-MuJoCo integration. Parsing the URDF files and creating the HSI's is now done in the `mujoco_ros_control` package based on the package created by Shadow Robot [1]. The script registers the HSIs and enforces limits to the joints. This means the computed control commands will make sure these limits will not be exceeded. Furthermore, it registers memory locations to store the joint data. This is all done in a plugin instance. The details of this configuration will be explained below.

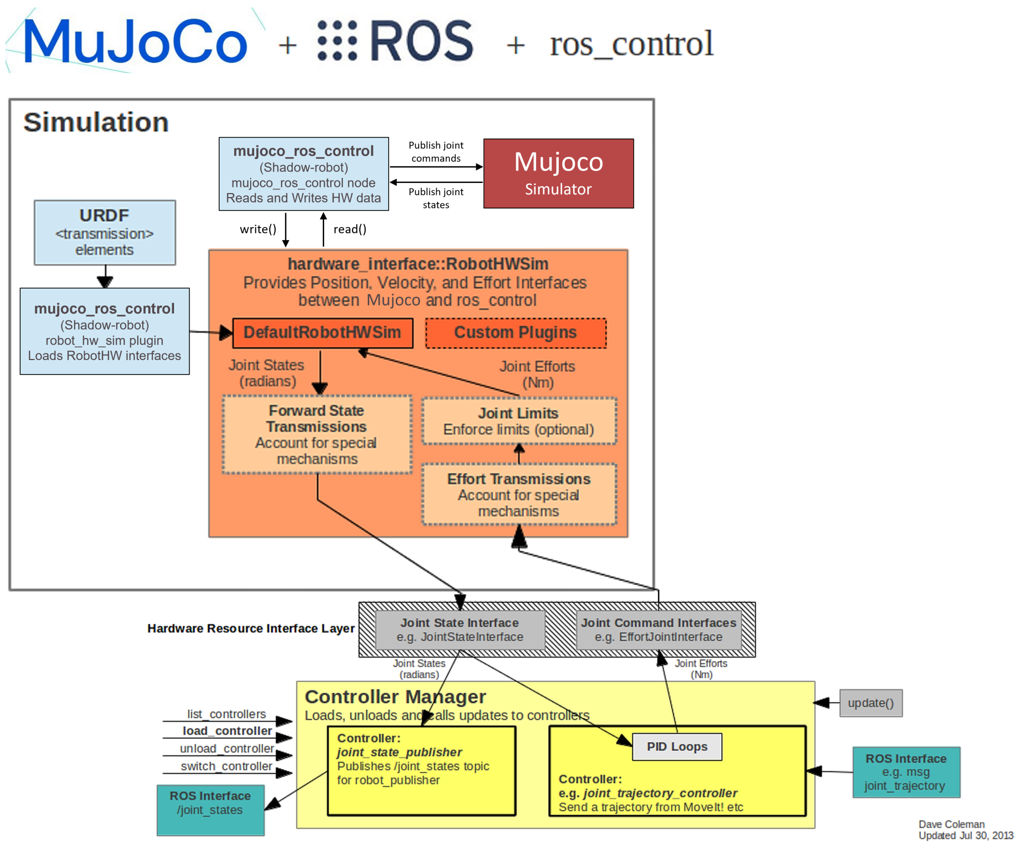
MuJoCo + ROS + ros_control
Simulation
mujoco_ros_control (Shadow-robot) mujoco_ros_control node Reads and Writes HW data
Publish joint commands
Publish joint states
Mujoco Simulator
URDF <transmission> elements
write()
read()
mujoco_ros_control (Shadow-robot) robot_hw_sim plugin Loads RobotHW interfaces
hardware_interface::RobotHWSim Provides Position, Velocity, and Effort Interfaces between Mujoco and ros_control
DefaultRobotHWSim
Custom Plugins
Joint States (radians)
Joint Efforts (Nm)
Forward State Transmissions Account for special mechanisms
Joint Limits Enforce limits (optional)
Effort Transmissions Account for special mechanisms
Hardware Resource Interface Layer
Joint State Interface e.g. JointStateInterface
Joint Command Interfaces e.g. EffortJointInterface
Joint States (radians)
Joint Efforts (Nm)
Controller Manager
Loads, unloads and calls updates to controllers
update()
list_controllers
load_controller
unload_controller
switch_controller
Controller: joint_state_publisher Publishes /joint_states topic for robot_publisher
PID Loops
Controller: e.g. joint_trajectory_controller Send a trajectory from MoveIt! etc
ROS Interface /joint_states
ROS Interface e.g. msg joint_trajectory
Dave Coleman Updated Jul 30, 2013

Figure 1.4: MuJoCo-ROS framework

The joint data can be read and written by the `mujoco_ros_control` node via the `RobotHWSimPlugin` instance called `robot_hw_sim`. However, this is not as easy as it sounds. The node has no access to the MuJoCo data, so cannot write the joint angles directly into the interface's memory, which in turn, can be accessed by the plugin instance only.
Similarly, the effort commands written in the memory by the `ros_controllers` controllers is only accessible via the `robot_hw_sim` plugin, which means the node has no direct access to this data. The data has to be gathered from the plugin instance.
To do so, new functions have to be made and used inside the `mujoco_ros_control` node. To visualize the data-stream, Figure 1.5 is created. The figure shows the functions which are used to pass the data, and in what part of the code, the joint data is being published.
Since `mujoco_ros_control` is the only ROS-node and the other instances are plugins, and thus do not run constantly, it is the only instance that can publish topics and subscribe to them. Therefore, the node subscribes to the `joint_states` topic and passes the states to the plugin instance by a function called `pass_mj_data`. It is then processed and written into the correct memory (HSI) which can be accessed by the `ros_controllers` controllers linked to the interfaces.

When the controllers write effort commands to the joint data memory, a copy is stored and passed by the **get_mj_data** function from the plugin instance back to the node. When the **mujoco_ros_control** node receives this data, it publishes the effort data on a ROS topic. This topic will be processed in the next step.

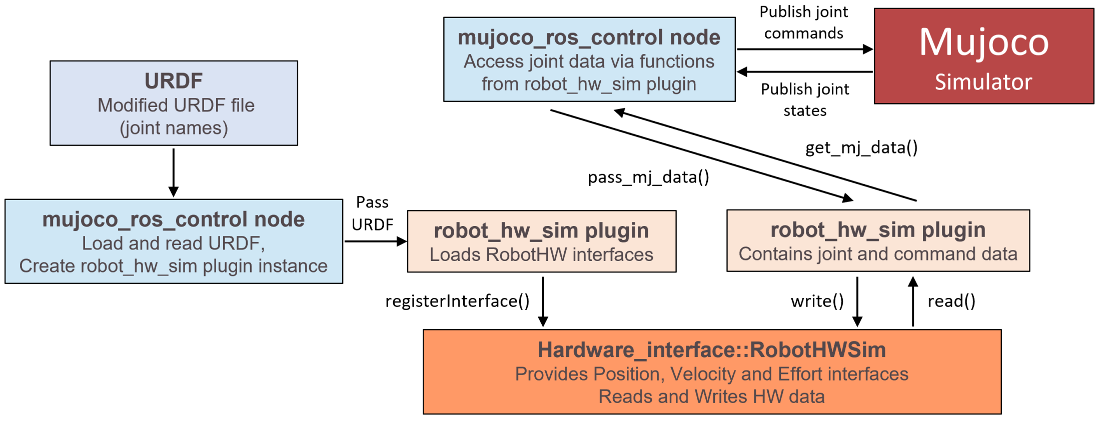

Figure 1.5: MuJoCo-ROS framework detailed

The control commands are now published in a ROS topic. To access them and write them into the MuJoCo data, a **ControlCommand** class has been created. This object is called when initializing the MuJoCo simulation and contains subscribers with callbacks to write the MuJoCo data at the correct location. For the controls of the RMA the processing is straight forward. Comparing the names of the joints and writing the effort commands into the correct actuator in the MuJoCo data.
For the control of the base, PID controllers have to be created in order to track the reference velocity to mimic the internal control of the real MB hardware. These controllers have to be tuned to have accurate velocity tracking of the base. The received velocity commands are stored in a variable and can be accessed by the PID controllers anytime. This is done since the PID controllers run at a much higher frequency.

2 | References

[1] Shadow Robot. mujoco_ros_control, 2020.

[2] Shadow Robot. Dexterous Robotic Hands & Teleoperated Robots, 2023.