Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Telecommunications and Media Informatics

Introducing The Effects of The Variations of Network Characteristics Into Cyber Physical Systems

Students' Scientific Conference paper

Author	*Department Supervisor*	*Industrial Supervisor*
József Pető	Dr. Sándor Molnár	Dr. Géza Szabó
	Associate Professor	Ericsson Ltd.

October 27, 2017

Contents

Összefoglaló

A robotszimuláció egy nélkülözhetetlen eszköz minden robotikával foglalkozó eszköztárában. Egy jól megtervezett szimulátor segítségével gyorsan tudunk algoritmusokat tesztelni, robotokat tervezni, regressziós teszteket végezni, és MI rendszereket tanítani valósághoz hasonló környezetekben.

Egy népszerű robotszimulátorral, a Gazeboval képesek vagyunk pontosan és hatékonyan szimulálni robotokat komplex beltéri és kültéri környezetben. Felhasználóinak erős fizikai motort, kiváló minőségű grafikát és kényelmesen használható programozói és felhasználói interfészeket nyújt. Ezenkívül nyílt forráskódú és ingyenes, a körülötte létrejött közösség aktívan fejleszti.

A Gazebo-ból azonban hiányzik a vezérlési késleletetés modellezése, ami egy teljes értékű kiberfizikai szimulátorrá tenné.

A dolgozatomban bemutatok egy általam írt Gazebo plugint (`https://github.com/Ericsson/robot_hw_sim_latency`), amivel képessé teszem a Gazebo-t arra, hogy késleltesse a belső vezérlő- és állapotüzeneteit, így lehetővé teszi a következő mérést.

A mérendő kiberfizikai rendszer egy 6 szabadságfokú robotkar (UR5), amit távolról vezérelnek a Robot Operációs Rendszer (ROS) használatával.

A cél a robotkarvezérlés minőségének mérése egy szimulált környezetben különböző hálózati feltételek mellett.

Mérési eredményeim azt mutatják, hogy a szimulált robotkar viselkedésére a hálózati késletetésnek olyan hatása van, amit előzetesen feltételezni lehetett, de további vizsgálatok szükségesek.

Abstract

Robot simulation is an essential tool in every roboticist's toolbox. A well-designed simulator makes it possible to rapidly test algorithms, design robots, perform regression testing, and train AI system using realistic scenarios.

A popular robotic simulator, Gazebo, offers the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments. It provides a robust physics engine, high-quality graphics, and convenient programmatic and graphical interfaces. Gazebo is open source and free with support from the community.

However, Gazebo lacks the feature of simulating the effects of control latency that would make it a fully-fledged cyber-physical system (CPS) simulator.

The CPS that I address to measure is a 6-DOF robotic arm (UR5) controlled remotely with velocity commands using Robot Operating System (ROS).

The main goal is to measure Quality of Control (QoC) related KPIs during various network conditions in a simulated environment.

I propose a Gazebo plugin (`https://github.com/Ericsson/robot_hw_sim_latency`) to make the above measurement feasible by making Gazebo capable to delay internal control and status messages.

My preliminary evaluation shows that there is certainly an effect on the behavior of the robotic arm with the introduced network latency in line with our expectations, but a more detailed further study is needed.

Introduction

Designing cyber-physical systems (CPS) is challenging because of a) the vast network and information technology environment connected with physical elements involves multiple domains such as controls, communication, analog and digital physics, and logic and b) the interaction with the physical world varies widely based on time and situation.

To ease the design of CPS, robot simulators have been used by robotics experts. A well-designed simulator makes it possible to rapidly test algorithms, design robots, perform regression testing, and train AI system using realistic scenarios.

There are various alternatives, sets of tools that make it possible to put together a CPS simulation environment, but it is very difficult, needs a lot of interfacing with various tools and impractical.

Gazebo was chosen as the target robot simulation environment that I intend to extend with new functionalities to make it capable of being applied as a CPS.

The main challenge with the design principle of Gazebo is that the control of actuators is deployed and run practically locally to the actuators. In this case, there is no need to consider the effects of a non-ideal link between the actuator and the controller. Considering the CPS context, as controllers are moved away from actuators, it becomes natural and even necessary to analyze the effects of the network link between them.

Gazebo has a plugin system that can be used to provide an interface to my modular network simulation environment.

The goal of this paper is to show the design principles of the network plugin and provide a tool for further research in CPS.

Structure of the paper

The paper consist of 5 chapters.

Chapter 1 Background In this chapter I describe the basics of cyber-physical systems, the Gazebo robot simulation environment, Robot Operating System, its various concepts, and the UR5 robot.

Chapter 2 Motivation and related work I describe the motivations behind this paper, and present related works.

Chapter 3 Proposed method I present the CPS that I address to measure, and the Gazebo plugin that I wrote extending the capabilities of the current Gazebo robotic simulator and turn it into a CPS system.

Chapter 3 Measurements I evaluate the effects of the simulated network latency — added to the CPS by my plugin — on various KPIs.

Chapter 4 Conclusion and further work Finally, I conclude this paper and describe avenues for further research.

Chapter 1

Background

In this chapter I describe the basics of cyber-physical systems, the Gazebo robot simulation environment, Robot Operating System, its various concepts, and the UR5 robot.

1.1 Cyber physical systems

A cyber-physical system (CPS) is a mechanism controlled or monitored by computer-based algorithms, tightly integrated with the internet and its users. Unlike more traditional embedded systems, a full-fledged CPS is typically designed as a network of interacting elements with physical input and output instead of as standalone devices. For tasks that require more resources than are locally available, one common mechanism is that nodes utilize the network connectivity to link the sensor or actuator part of the CPS with either a server or a cloud environment, enabling complex processing tasks that are impossible under local resource constraints. Currently, one of the main focus of cloud based robotics is to speed up the processing of input data collected from many sensors with big data computation. Another approach is to collect various knowledge bases in centralized locations e.g., possible grasping poses of various 3D objects.

Another aspect of cloud robotics is the way in which the robot control related functionality is moved into the cloud. The simplest way is to run the original robot specific task in a cloud without significant changes in it. For example, in a Virtual Machine (VM), in a container, or in a virtualized Programmable Logic Controller (PLC). Another way is to update, modify or rewrite the code of robot related tasks to utilize existing services or APIs of the cloud. The third way is to extend the cloud platform itself with new features that make robot control more efficient. These new robot-aware cloud features can be explicitly used by robot related tasks (i.e. new robot-aware services or APIs offered by cloud) or can be transparent solutions (e.g., improving the service provided by the cloud to meet the requirement of the robot control).

The requirements of a widely applicable CPS are the following:

- Should be modular in terms of interfacing with the CPS
- Should be modular in terms of interfacing with network simulator, realization environment
- Should be able to cooperate with widely applied environments

1.2 Gazebo

Gazebo[1] was chosen as the target robot simulation environment that I intend to extend with new functionalities to make it capable of being applied as a CPS. Gazebo is a 3D dynamic simulator with the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments. While similar to game engines, Gazebo offers physics simulation at a much higher degree of fidelity, a rich library of robot models and environments, a suite of sensors, and interfaces for both users and programs. Gazebo is free and widely used among robotic experts.

Typical uses of Gazebo include: testing robotics algorithms, designing robots, performing regression testing with realistic scenarios.

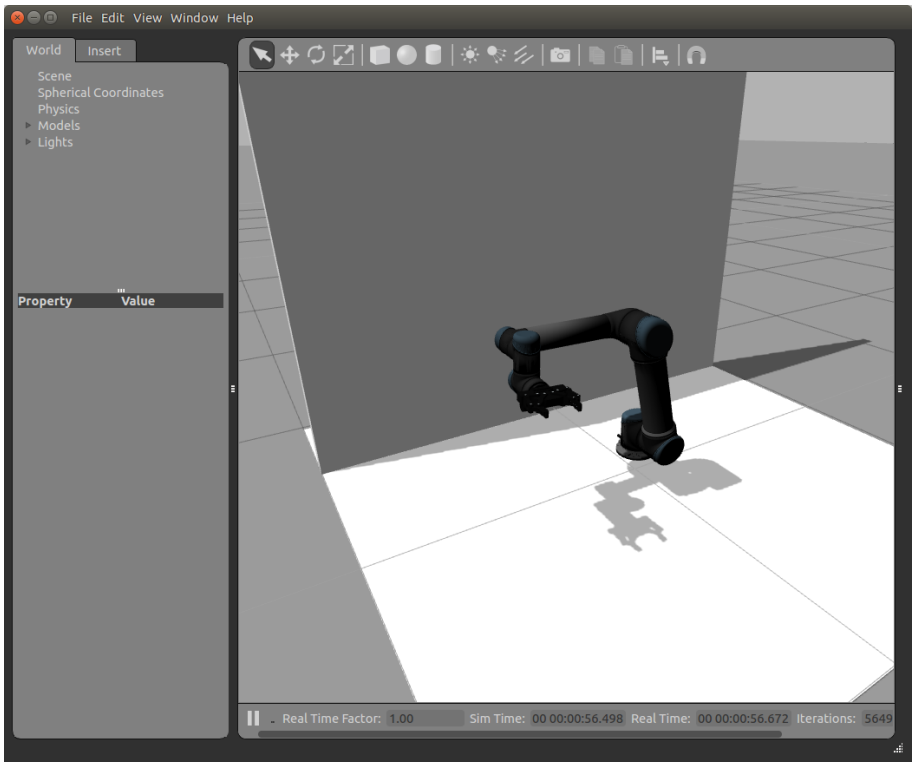

Figure 1.1: *Gazebo simulation of the UR5 robot arm*

1.3 UR5 Robot arm

The UR5[2] robot arm designed by Universal Robots has 6 degrees of freedom with its 6 rotating joints. Its payload can be up to 5 kg. It has a reach of 850 mm. It is controlled by sending text commands to it using a TCP/IP connection. The commands are in a special script language called URScript[3]. By sending commands you can control the robot's Cartesian position, velocity, joint angle and velocity.

1.4 Robot Operating System

Robot Operating System (ROS)[4] is used to control the movement of the robot arm. ROS is an open-source, meta-operating system for robot software development. It provides standard services that would be expected from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers.

ROS is not a realtime framework, though it is possible to integrate ROS with realtime code.

ROS was designed to be as distributed and modular as possible, so that users can use as much or as little of ROS as they desire. The distributed nature of ROS also fosters a large community of user-contributed packages that add a lot of value on top of the core ROS system. At last count there were over 3,000 packages in the ROS ecosystem, and that is only the ROS packages that people have taken the time to announce to the public. These packages range in fidelity, covering everything from proof-of-concept implementations of new algorithms to industrial-quality drivers and capabilities. The ROS user community builds on top of a common infrastructure to provide an integration point that offers access to hardware drivers, generic robot capabilities, development tools, useful external libraries, and more.

The ROS framework is easy to implement in any modern programming language. It is already implemented it in Python, C++, and Lisp, and there are experimental libraries in Java and Lua.

ROS currently only runs on Unix-based platforms. Software for ROS is primarily tested on Ubuntu and Mac OS X systems, though the ROS community has been contributing support for Fedora, Gentoo, Arch Linux and other Linux platforms. [5, 6]

1.5 ROS Concepts

ROS has three levels of concepts: the Filesystem level, the Computation Graph level, and the Community level. [7]

1.5.1 ROS Filesystem Structure

1.5.1.1 Packages

Packages[8] are the main unit for organizing software in ROS. In the file system they are represented by folders which contain a package manifest.

A package may contain ROS runtime processes (nodes), a ROS-dependent library, datasets, configuration files, or anything else that is usefully organized together. Packages are the most atomic build item and release item in ROS. Meaning that the most granular thing you can build and release is a package.

ROS packages tend to follow a common structure.

- include/package_name: C++ include headers
- msg/: Folder containing Message (msg) types
- src/package_name/: Source files, especially Python source that are exported to other packages.
- srv/: Folder containing Service (srv) types
- scripts/: executable scripts
- CMakeLists.txt: CMake build file
- package.xml: Package manifest
- CHANGELOG.rst: Many packages will define a changelog which can be automatically injected into binary packaging and into the wiki page for the package

1.5.1.2 Metapackages

Multiple packages can be organised in a special package, called metapackage to make updating and handling multiple packages easier.[9]

A metapackage is used in a similar fashion as virtual packages are used in the debian packaging world. A metapackage simply references one or more related packages which are loosely grouped together.

The metapackage just contains a package manifest, which lists all its packages as dependencies.

1.5.1.3 Package Manifests

The package manifest[10] is an XML file that must be included with any package's root folder.

Manifests provide metadata about a package, including its name, version, description, license information, build and runtime dependencies, and other meta information like exported packages.

Each package.xml file has the <package> tag as the root tag in the document.

There are a minimal set of tags that need to be nested within the <package> tag to make the package manifest complete.

- <name> - The name of the package
- <version> - The version number of the package (required to be 3 dot-separated integers)
- <description> - A description of the package contents
- <maintainer> - The name of the person(s) that is/are maintaining the package
- <license> - The software license(s) (e.g. GPL, BSD, ASL) under which the code is released.

1.5.1.4 Message types

Message descriptions, stored in .msg files, define the data structures for messages sent in ROS.[11]

This description makes it easy for ROS tools to automatically generate source code for the message type in several target languages. Message descriptions are stored in .msg files in the msg/ subdirectory of a ROS package.

There are two parts to a .msg file: fields and constants. Fields are the data that is sent inside of the message. Constants define useful values that can be used to interpret those fields (e.g. enum-like constants for an integer value). Each field consists of a type and a name, separated by a space:

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

For example:

```
int32 x
int32 y
```

The field name determines how a data value is referenced in the target language. For example, a field called 'pan' would be referenced as 'obj.pan' in Python, assuming that 'obj' is the variable storing the message.

Field types can be:

- a built-in type, such as "float32 pan" or "string name"
- names of Message descriptions defined on their own, such as "geometry_msgs/PoseStamped"
- fixed- or variable-length arrays (lists) of the above, such as "float32[] ranges" or "Point32[10] points"
- the special Header type, which maps to std_msgs/Header

Table 1.1: *Built-in types*

Primitive Type	**Serialization**	**C++**	**Python**
bool	unsigned 8-bit int	uint8_t	bool
int8	signed 8-bit int	int8_t	int
uint8	unsigned 8-bit int	uint8_t	int
int16	signed 16-bit int	int16_t	int
uint16	unsigned 16-bit int	uint16_t	int
int32	signed 32-bit int	int32_t	int
uint32	unsigned 32-bit int	uint32_t	int
int64	signed 64-bit int	int64_t	long
uint64	unsigned 64-bit int	uint64_t	long
float32	32-bit IEEE float	float	float
float64	64-bit IEEE float	double	float
string	ascii string (4)	std::string	str
time	secs/nsecs unsigned 32-bit ints	ros::Time	rospy.Time
duration	secs/nsecs signed 32-bit ints	ros::Duration	rospy.Duration

ROS provides the special Header type to provide a general mechanism for setting frame IDs for libraries like tf. While Header is not a built-in type (it's defined in std_msgs/msg/Header.msg), it is commonly used and has special semantics. If the first field of your .msg is "Header header", it will be resolved as std_msgs/Header.

Table 1.2: *Array handling*

Array Type	**Serialization**	**C++**	**Python**
fixed-length	no extra serialization	boost::array	tuple (1)
variable-length	uint32 length prefix	std::vector	tuple (1)
uint8[]	see above	as above	bytes (2)
bool[]	see above	std::vector<uint8_t >	list of bool

```
Header.msg:
\#Standard metadata for higher-level flow data types
\#sequence ID: consecutively increasing ID
uint32 seq
\#Two-integer timestamp that is expressed as:
\# * stamp.secs: seconds (stamp\_secs) since epoch
\# * stamp.nsecs: nanoseconds since stamp\_secs
\# time-handling sugar is provided by the client library
time stamp
\#Frame this data is associated with
string frame\_id
```

1.5.1.5 Service types

Service descriptions, stored in a.srv files[12] , define the request and response data structures for services in ROS. A service description file consists of a request and a response msg type, separated by '—'. Any two .msg files concatenated together with a '—' are a legal service description.

```
string str
---
string str
```

1.5.2 ROS Computation Graph

The ROS Computation Graph is a peer-to-peer network of processes (potentially distributed across machines) that are loosely coupled using the ROS communication infrastructure. The basic Computation Graph concepts of ROS are Master, nodes, Parameter Server, messages, services, topics, and bags, all of which provide data to the Graph in different ways.

1.5.2.1 Master

The ROS Master[13] acts as a nameservice in the ROS Computation Graph. It stores topics and services registration information for ROS nodes. Nodes communicate with the Master to report their registration information. As these nodes communicate with the Master, they can receive information about other registered nodes and make connections as appropriate. The Master will also make callbacks to these nodes when this registration information changes, which allows nodes to dynamically create connections as new nodes are run.

The Master is implemented via XMLRPC[14], which is a stateless, HTTP-based protocol. XMLRPC was chosen primarily because it is relatively lightweight, does not require a stateful connection, and has wide availability in a variety of programming languages.

1.5.2.2 Parameter Server

The parameter server[15] is a shared, multi-variate dictionary that is accessible via network APIs. It runs inside of the ROS Master. Nodes use this server to store and retrieve parameters at runtime. As it is not designed for high-performance, it is best used for static, non-binary data such as configuration parameters. It is meant to be globally viewable so that tools can easily inspect the configuration state of the system and modify if necessary.

The Parameter Server API is also implemented via XMLRPC[14]. The use of XML-RPC enables easy integration with the ROS client libraries and also provides greater type flexibility when storing and retrieving data. The Parameter Server can store basic XML-RPC scalars (32-bit integers, booleans, strings, doubles, iso8601 dates), lists, and base64-encoded binary data. The Parameter Server can also store dictionaries (i.e. structs).

1.5.2.3 Nodes

Nodes[16] are processes that perform computation. ROS is designed to be modular at a fine-grained scale; a robot control system usually comprises many nodes. Nodes are combined together into a graph and communicate with one another using streaming topics, RPC services, and the Parameter Server. The use of nodes in ROS provides several benefits to the overall system. There is additional fault tolerance as crashes are isolated to individual nodes. Code complexity is reduced in comparison to monolithic systems. Implementation details are also well hidden as the nodes expose a minimal API to the rest of the graph and alternate implementations, even in other programming languages, can easily be substituted.

Every node has a URI, which corresponds to the host:port of the XMLRPC server it is running[14]. The XMLRPC server is not used to transport topic or service data: instead, it is used to negotiate connections with other nodes and also communicate with the Master. This server is created and managed within the ROS client library, but is generally not visible to the client library user. The XMLRPC server may be bound to any port on the host where the node is running.

A ROS node is written with the use of a ROS client library, such as roscpp or rospy.

1.5.2.4 Messages

Nodes communicate with each other by publishing messages to topics[17]. A message is a simple data structure, comprising typed fields. Standard primitive types (integer, floating point, boolean, etc.) are supported, as are arrays of primitive types. Messages can include arbitrarily nested structures and arrays.

They are defined by .msg files that are simple text files specifying the data structure of

a message. The ROS Client Libraries implement message generators that translate .msg files into source code, so the messages are programming language independent.

1.5.2.5 Standard and common messages

The std_msgs package[18] contains wrappers for ROS primitive types, which are documented in the msg specification. It also contains the Empty type, which is useful for sending an empty signal. However, these types do not convey semantic meaning about their contents: every message simply has a field called "data". Therefore, while the messages in this package can be useful for quick prototyping, they are not intended for "long-term" usage.

There is a special message type in std_msgs, the Header type which contains a sequence number, a timestamp and a frame_id string that describes in which coordinate frame this message is relative to. Message types ending in Stamped contain this type.

The common_msgs[19] package contains messages that are widely used by other ROS packages. These includes messages for actions (actionlib_msgs), diagnostics (diagnostic_msgs), geometric primitives (geometry_msgs), robot navigation (nav_msgs), and common sensors (sensor_msgs), such as laser range finders, cameras, point clouds.

1.5.2.6 Topics

Messages are routed via a transport system with publish / subscribe semantics[19]. A node sends out a message by publishing it to a given topic. The topic is a name that is used to identify the content of the message. A node that is interested in a certain kind of data will subscribe to the appropriate topic. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics. In general, publishers and subscribers are not aware of each others' existence. The idea is to decouple the production of information from its consumption. Logically, one can think of a topic as a strongly typed message bus. Each bus has a name, and anyone can connect to the bus to send or receive messages as long as they are the right type.

ROS currently supports TCP/IP-based and UDP-based message transport. The TCP/IP-based transport is known as TCPROS and streams message data over persistent TCP/IP connections. TCPROS is the default transport used in ROS and is the only transport that client libraries are required to support. The UDP-based transport, which is known as UDPROS and is currently only supported in roscpp, separates messages into UDP packets. UDPROS is a low-latency, lossy transport, so is best suited for tasks like teleoperation.

For example, the sequence by which two nodes begin exchanging messages is:[14]

1. Publisher node registers with the Master by sending its name, XMLRPC host:port, topic to publish to and topic type. [XMLRPC]

2. Subscriber node registers with the Master by sending its name, XMLRPC host:port, topic to subscribe to and topic type. [XMLRPC]

3. Master notices that there is a node that is interested in a topic that has a publisher, so it sends the XMLRPC address of the publisher to the subscriber. [XMLRPC]

4. The Subscriber sends a connection request to the XMLRPC address of the Publisher, sending its name, the topic name and a list of supported protocols. [XMLRPC]

5. The Publisher responds with a selected protocol and the address which uses the negotiated protocol. [XMLRPC]

6. The Subscriber connects to the address using the negotiated protocol.

7. The connection is established, data is sent from the publisher to the subscriber.

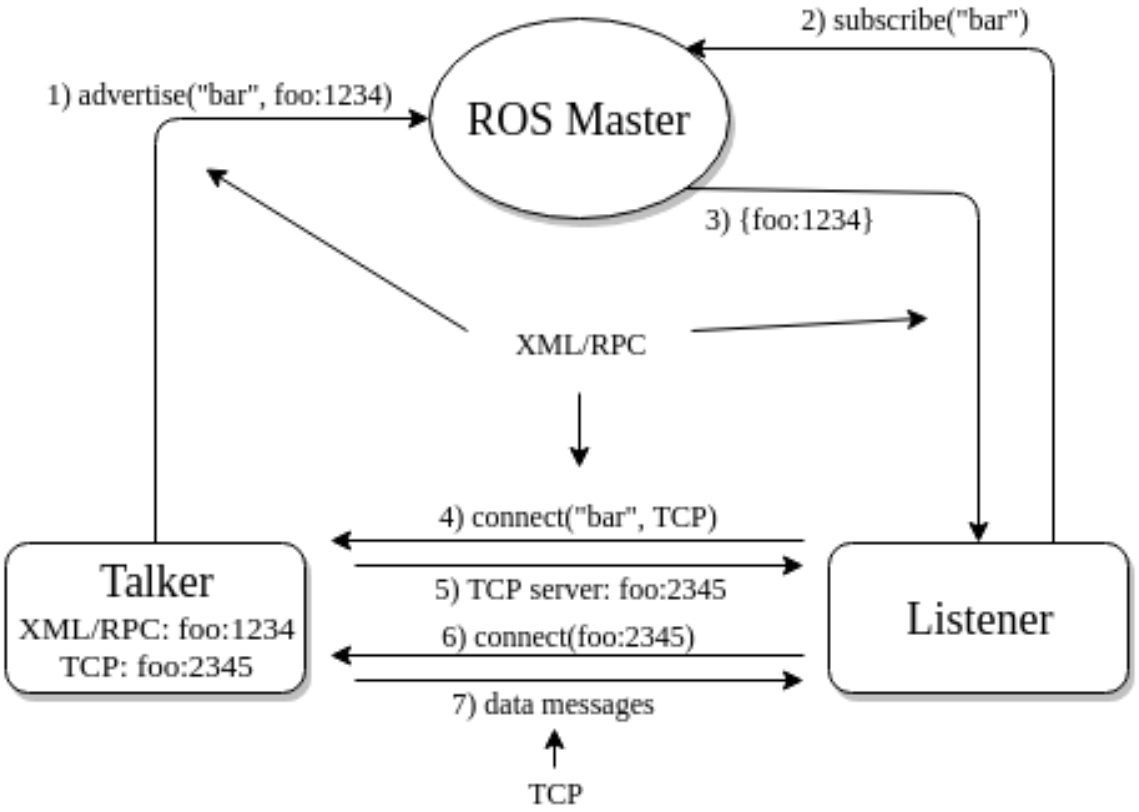

Figure 1.2: *The sequence of connection*

The Master keeps track of the publishers and subscribers of all topics, so when there is a new publisher to a topic, it can notify the subscribers of that topic to connect to that publisher. Also, when there is a new subscriber, it will send all publishers address to it so it can connect to them all.

Consequently, the order in which the nodes are registered does not matter, simplifying the startup processes of complicated computation graphs.

1.5.2.7 Services

The publish / subscribe model of topics is a very flexible communication paradigm, but its many-to-many, one-way transport is not appropriate for request / reply interactions, which are often required in a distributed system. Request / reply is done via services[20] , which are defined by a pair of message structures: one for the request and one for the reply. A providing node offers a service under a name and a client uses the service by sending the request message and awaiting the reply. ROS client libraries generally present this interaction to the programmer as if it were a remote procedure call. Services are defined using .srv files, which like .msg files are compiled into source code by a ROS client library.

1.5.2.8 Actions

In any large ROS based system, there are cases when someone would like to send a request to a node to perform some task, and also receive a reply to the request. This can currently be achieved via ROS services. In some cases, however, if the service takes a long time to execute, the user might want the ability to cancel the request during execution or get periodic feedback about how the request is progressing. The actionlib package[21] provides tools to create servers that execute long-running goals that can be preempted. It also provides a client interface in order to send requests to the server.

1.5.2.9 Bags

Bags[22] are a format for saving and playing back ROS message data. Bags are an important mechanism for storing data, such as sensor data, that can be difficult to collect but is necessary for developing and testing algorithms. Bags are usually created using the rosbag command-line tool.

1.5.3 Other important ROS concepts

1.5.3.1 Client Library

A ROS client library[23] is a collection of code that eases the job of the ROS programmer. It takes many of the ROS concepts and makes them accessible via code. In general, these libraries let you write ROS nodes, publish and subscribe to topics, write and call services, and use the Parameter Server. Such a library can be implemented in any programming language, though the current focus is on providing robust C++ and Python support. Main client libraries are roscpp, rospy and roslisp.

1.5.3.2 Coordinate frames

In a robotic system there are multiple coordinate frames that change in time. Converting vectors between them correctly is not simple.

tf[24] (and its successor tf2[24]) is a package that lets the user keep track of multiple coordinate frames over time. tf maintains the relationship between coordinate frames in a tree structure buffered in time, and lets the user transform points, vectors, etc between any two coordinate frames at any desired point in time.

1.5.3.3 Unified Robot Description Format

The Unified Robot Description Format (URDF)[25] is an XML specification to describe a robot. It is designed to be as general as possible, but obviously the specification cannot describe all robot. Only tree structures can be represented, ruling out all parallel robots.

The specification assumes the robot consists of rigid links connected by joints; flexible elements are not supported. The format can be used to specify the kinematic and dynamic description of the robot, the visual representation of the robot and the collision model of the robot.

1.5.3.4 Plugins

The pluginlib[26] package provides tools for writing and dynamically loading plugins using the ROS build infrastructure. To work, these tools require plugin providers to register their plugins in the package.xml of their package.

It is a C++ library for loading and unloading plugins from within a ROS package. Plugins are dynamically loadable classes that are loaded from a runtime library (i.e. shared object, dynamically linked library).

With pluginlib, one does not have to explicitly link their application against the library containing the classes – instead pluginlib can open a library containing exported classes at any point without the application having any prior awareness of the library or the header file containing the class definition. Plugins are useful for extending/modifying application behavior without needing the application source code.

1.5.3.5 catkin

ROS utilizes a custom build system, catkin[27], that extends CMake to manage dependencies between packages.

The build system is needed, because ROS is a very large collection of loosely federated packages. That means lots of independent packages which depend on each other, utilize various programming languages, tools, and code organization conventions.

Because of this, the build process for a target in some package may be completely

Table 1.3: *Recent distributions*

Distribution	Release Date	End of Life Date
ROS Kinetic Kame	May 23rd, 2016	May, 2021
ROS Jade Turtle	May 23rd, 2015	May, 2017
ROS Indigo Igloo	July 22nd, 2014	April, 2019

different from the way another target is built. catkin specifically tries to improve development on large sets of related packages in a consistent and conventional way. In other words, both rosbuild and now catkin aim to make building and running ROS code easier by using tools and conventions to simplify the process. Efficiently sharing ROS-based code would be more difficult without it.

1.5.4 ROS Community

There are ROS resources that enable separate communities to exchange software and knowledge.

1.5.4.1 Distributions

ROS Distributions[28] are collections of versioned stacks that you can install. Distributions play a similar role to Linux distributions: they make it easier to install a collection of software, and they also maintain consistent versions across a set of software. There are 3 distributions that are maintained at the time of writing.

Indigo Igloo and Kinetic Kame are LTS (Long Term Support) distributions, meaning they receive updates for 5 years. Jade Turtle, not being a LTS release, is only updated for 2 years.

1.5.4.2 ROS Wiki

The ROS community Wiki[29] is the main forum for documenting information about ROS. Anyone can sign up for an account and contribute their own documentation, provide corrections or updates, write tutorials, and more.

1.5.5 ROS Names

1.5.5.1 Graph Resource Names

Graph Resource Names[30] provide a hierarchical naming structure that is used for all resources in a ROS Computation Graph, such as Nodes, Parameters, Topics, and Services.

They are an important mechanism in ROS for providing encapsulation. Each resource is defined within a namespace, which it may share with many other resources. In gen-

eral, resources can create resources within their namespace and they can access resources within or above their own namespace. Connections can be made between resources in distinct namespaces, but this is generally done by integration code above both namespaces. This encapsulation isolates different portions of the system from accidentally grabbing the wrong named resource or globally hijacking names.

Names are resolved relatively, so resources do not need to be aware of which namespace they are in. This simplifies programming as nodes that work together can be written as if they are all in the top-level namespace.

Any name within a ROS Node can be remapped when the node is launched at the command-line.

1.5.6 Package Resource Names

Package Resource Names[30] are used in ROS with Filesystem-Level concepts to simplify the process of referring to files and data types on disk. Package Resource Names are very simple: they are just the name of the Package that the resource is in plus the name of the resource. For example, the name "std_msgs/String" refers to the "String" message type in the "std_msgs" Package.

1.6 Used ROS packages

To avoid reinventing the wheel, multiple ready made packages were used to create the measurement setup.

1.6.1 universal_robot package

The universal_robot metapackage[31] contains packages that provide nodes written in Python for communication with Universal's industrial robot controllers and URDF models for various robot arms (UR3, UR5, UR10).

1.6.1.1 ur_description

This package contains the model of the robot, the urdf and the mesh files describing the robot links.

1.6.1.2 ur_gazebo

This package contains files that aid in starting a robot simulation

1.6.2 ros_control package

Ros_control[32] is a set of packages defining a set of interfaces, which are designed to abstract away differences between robot hardware.

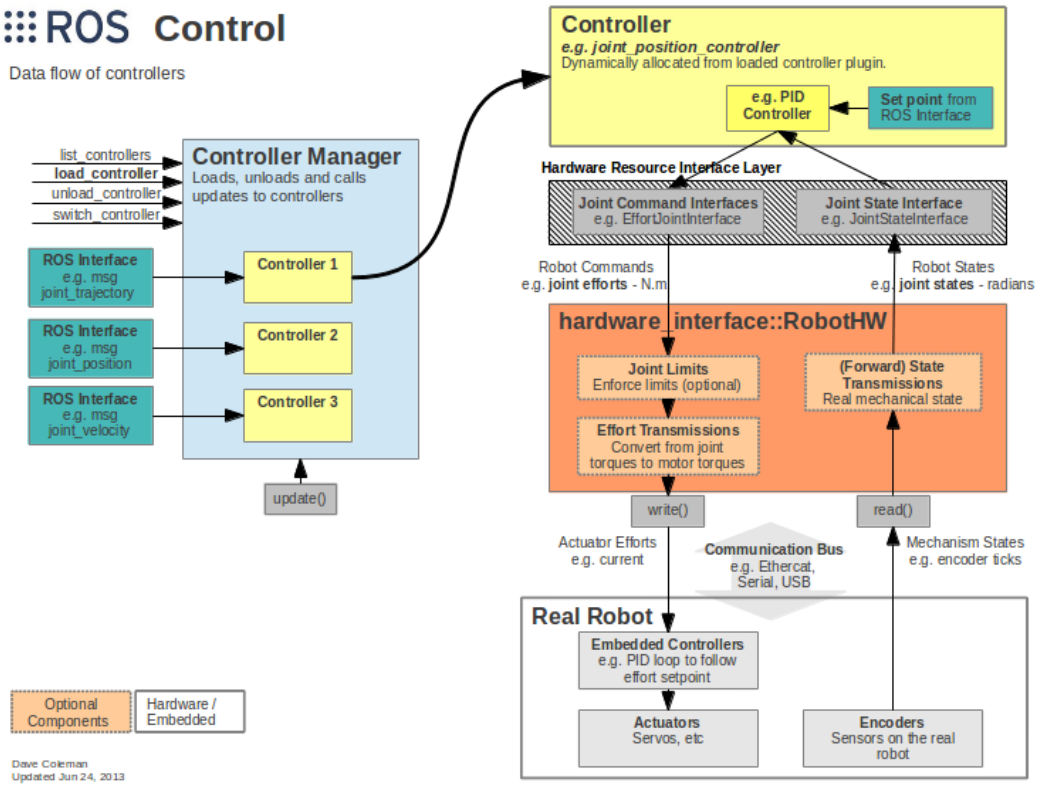

Figure 1.3: *Overview of ros_control*

There are controllers, which provide standard ROS interfaces (topic or service) in order to allow communication between the robot and other ROS nodes using not robot-specific topics, and messages. The controllers are not robot specific, but are using interfaces that are C++ classes to read from and write to. These interfaces represent hardware elements (e.g: VelocityJointInterface can represent a joint that can be controlled using velocity commands). The interfaces basically shared memory where command can be written and state can be read.

The controllers for example can use PID controllers to control the interfaces, that way they can for example receive position commands from a topic, and through a PID controller it can control a VelocityJointInterface.

The controller can also read from the interface, so it can publish information about the hardware represented by the interface (e.g.: joint state, torque information).

The interfaces are implemented in a hardware specific driver extending hardware_interface::RobotHW, that takes care of communicating with the robot using its hardware specific communication method (serial, Modbus, Ethernet, USB).

The controllers and drivers are implemented using the pluginlib package to make them dynamically loaded.

1.6.2.1 joint_state_controller/JointStateController

This is a controller that reads state data (joint angles, velocities, efforts) from JointStateInterfaces, and publishes them in sensor_msgs/JointState messages to the /joint_state topic.

1.6.2.2 velocity_controllers/JointTrajectoryController

It is a controller for executing joint-space trajectories on a group of joints. Trajectories are specified as a set of waypoints to be reached at specific time instants, which the controller attempts to execute as well as the mechanism allows. Waypoints consist of positions, and optionally velocities and accelerations.

1.6.3 moveit package

MoveIt![33] is state of the art software that runs on top of ROS for mobile manipulation, incorporating the latest advances in motion planning, manipulation, 3D perception, kinematics, control and navigation. It provides an easy-to-use platform for developing advanced robotics applications, evaluating new robot designs and building integrated robotics products for industrial, commercial, R&D and other domains.

MoveIt! is designed to work with many different types of planners, which is ideal for benchmarking improved planners against previous methods.

The figure 1.4. shows the high-level system architecture for the primary ROS node provided by MoveIt! called move_group. This node serves as an integrator: pulling all the individual components together to provide a set of ROS actions and services for users to use.

1.6.4 lemniscatepublisher package

This package originally written by me during my summer internship to publish an elaborate trajectory to the UR5 robot. First, it creates list of waypoints, then using MoveIt it plans a precise trajectory — that can be used by the robot — with time parametrized position and velocity data. Then, also using MoveIt, it sends this trajectory to the JointTrajectoryController for execution. I modified this package, to publish — instead of an

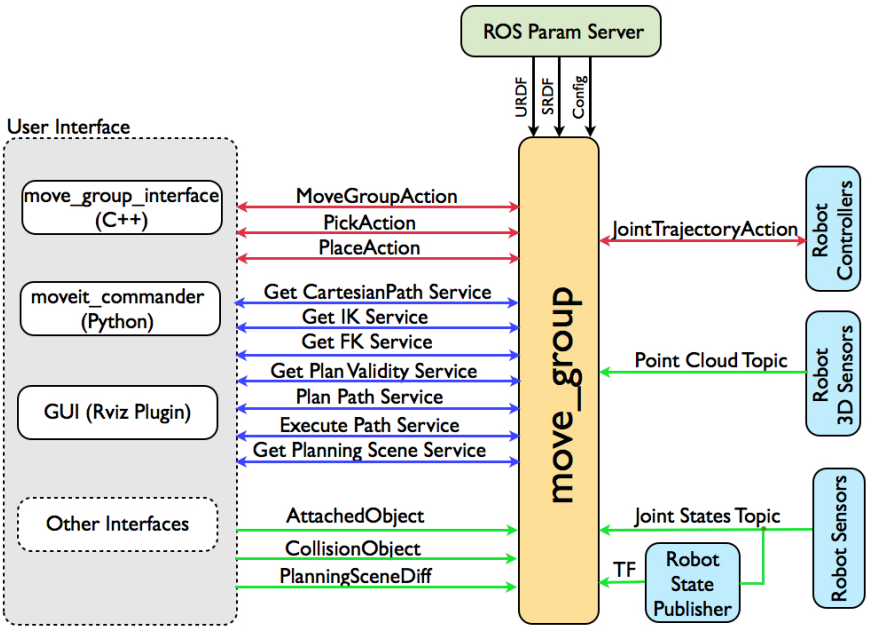

Figure 1.4: *Moveit architecture*

elaborate trajectory — a simple trajectory between 3 points.

1.6.5 gazebo_ros_pkgs package

gazebo_ros_pkgs[34] is a set of ROS packages that provide the necessary interfaces to simulate a robot in the Gazebo 3D rigid body simulator for robots. It integrates with ROS using ROS messages, services and dynamic reconfigure.

It contains a converter that converts URDF into SDF which is the world description language that Gazebo uses. This way there is no need to maintain two sets of models.

1.6.5.1 gazebo_ros_pkgs package

gazebo_ros_pkgs also contains the gazebo_ros_control package which is a ROS package for integrating the ros_control controller architecture with the Gazebo simulator.

It provides a Gazebo plugin which instantiates a ros_control controller manager and connects it to a Gazebo model. The Gazebo plugin also loads in the DefaultRobotHWSim plugin through pluginlib which creates the hardware_interfaces (position, velocity or effort) for each joint as defined in the loaded URDF.

1.6.6 xacro package

The xacro package[35] is most useful when working with large XML documents such as URDFs. 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.

1.6.7 roslaunch package

roslaunch[36] is a tool for easily launching multiple ROS nodes locally and remotely via SSH, as well as setting parameters on the Parameter Server. It includes options to automatically respawn processes that have already died. roslaunch takes in one or more XML configuration files (with the .launch extension) that specify the parameters to set and nodes to launch, it is also possible to upload configurations to the Parameter Server from YAML files.

1.6.8 hector_trajectory_server package

This package provides a node that saves tf based trajectory data given a target and and source frame. The trajectory is saved internally as a nav_msgs/Path and can be obtained using a service or topic.

Chapter 2

Motivation and related work

2.1 Competitions

A frontier method to push research groups to their limits is to organize competitions. DARPA, a research group in the U.S. Department of Defense, announced the DARPA Robotics Challenge with a US $2 million dollar prize for the team that could produce a first responder robot performing a set of tasks required in an emergency situation.

During the DARPA Trials of December 2013, a restrictive device was inserted into the control computers of each competing team and the computer that formed the 'brain' of the robot.

The intent of the network degradation was to roughly simulate the kind of less than perfect communications that might exist during those kinds of emergency or disaster situations in which these robots would be deployed.

The restrictive device –, a Mini Maxwell network emulator from InterWorking Labs – alternated between a 'good' mode and a 'bad' mode of network communication, every sixty seconds. 'Good' minutes permitted communications at a rate of 1 Mbps (in either direction) and a base delay of 50 ms (in each direction.) 'Bad' minutes permitted communications at a rate of 100 Kbps (in either direction) and a base delay of 500 ms (in each direction.)

At the end of each minute, a transition occurred from bad-to-good or good-to-bad. A side effect of these transitions was packet-reordering.

The impact of network degradation on the teams was larger than expected. Informal feedback suggested that several teams did not realize that rate limitation induces network congestion or the ramifications of that congestion. Network congestion means the growth of queues of packets awaiting their turn to pass through the congestion. And that queue growth, in turn, means increases, often very substantial increases, in the time for a packet to move from the sender to the receiver.

Nor did all teams appreciate the degree to which rate limitation induced congestion

would persist and gradually diminish over a period of time after the constraint has been removed.

Several teams made use of the TCP transport protocol without understanding how TCP tries to be a good network citizen by detecting congestion and reacting to that congestion by reducing its transmission rate to avoid adding to the congestion and making things worse. Other teams used the UDP transport protocol: these teams seemed to sometimes be surprised by the reordering of packets.

However, several of those teams quickly made changes to their software to handle out of sequence packets. At least one team switched from a TCP based transport to a UDP based transport mid-way through the trials.

Some teams appeared to have been surprised by the behavior of the network protocol stacks, particularly TCP stacks, in the operating systems underneath their code. [37] The above experiences would have been probably less striking to the teams if they were able to test the network characteristics changes in a simulation environment.

A recent competition Agile Robotics for Industrial Automation Competition (ARIAC)[38] targets industrial related applications. ARIAC is a simulation-based competition is designed to promote agility in industrial robot systems by utilizing the latest advances in artificial intelligence and robot planning. There is no tricky network environment in the ARIAC competition. The industry relies on robust low-delay protocols. That is why it is an interesting aspect to see what happens when those links and protocols are exchanged. For instance, what are the possible performance improvements or degradation when the control or sensors data processing in an industrial scenario are moved further away from the actuators and how different protocols would fare under various network characteristics?

2.2 Choosing simulators

In both of the above competitions, Gazebo provided the simulation infrastructure. In a more structured study about the level of how wide-spread the various simulator tools were done in [39]. It showed that Gazebo emerges as the best choice among the open-source projects.

Authors of [40] describes some early experiments in linking the OMNET++ simulation framework with the ROS middleware for interacting with robot simulators in order to get within the OMNET++ simulation a robot's position which is accurately simulated by an external simulator based on ROS. The motivation is to use well-tested and realistic robot simulators for handling all the robot navigation tasks (obstacle avoidance, navigation towards goals, velocity, etc.) and to only get the robot's position in OMNET++ for interacting with the deployed sensors. My goal is the other way around, thus to introduce

the effects of the network simulator into the robot simulator.

The roadmap of Gazebo development shows that version 9.0 arriving at 2018-01-25 will have support to integrate network simulation (ns-3 or EMANE). Further information regarding if this feature will be like [40] or the one I propose in this paper is not available yet.

Chapter 3

Proposed method

In this chapter I present the CPS that I address to measure, and the Gazebo plugin that I wrote extending the capabilities of the current Gazebo robotic simulator and turn it into a CPS system.

3.1 Overview

The CPS that I address to measure is a robotic arm (UR5 [2]) controlled remotely with velocity commands. The main goal is to measure Quality of Control (QoC) e.g., cumulated PID error during trajectory execution, cumulated difference in joint space between the executed and calculated trajectories, etc. related KPIs during various network conditions in this setup.

Figure 3.1. shows the use case with real hardware that I target to simulate in Gazebo. The left side of the figure (Hardware) shows the same data elements described in ros_control (1.6.2), whereas the right side of the picture (Realization) uses the same colors for the boxes to describe a specific realization. In the specific case, the UR5 can be accessed via TCP/IP ports 50001 to send command messages and port 50003 to read the robot status messages. The lemniscatepublisher (described in 1.6.4) generates a sparse trajectory consisting of a few waypoints in Cartesian space, then it uses the C++ interface of MoveIt to plan the joint space trajectories. Then using MoveIt it sends trajectories to a type of ros_control controller: joint_trajectory_controller (1.6.2.2)(shown in yellow) which at the start of simulation was started by the controller manager.

The ur_modern_driver [41] implements the hardware resource interface layer by simply copying the velocity control packets to the proper TCP sockets. A middle node can be deployed between the robot driver and the robot (green) that can alter the network characteristics.

A trivia approach to setup the above architecture in a simulation environment is pro-

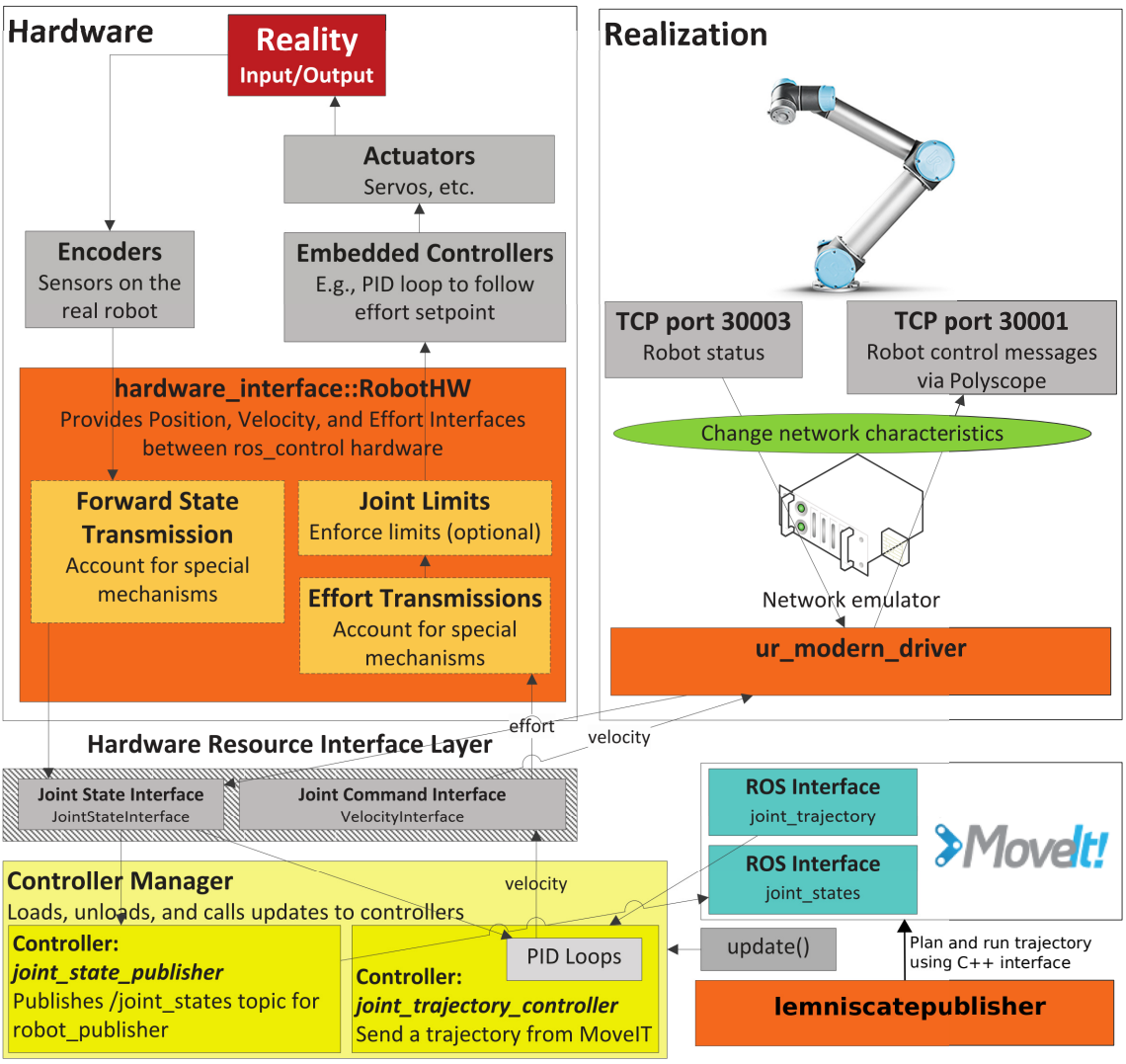

Figure 3.1: *Target architecture to be realized with simulator*

vided by Universal Robots. Universal Robots simulator software [42] is a java software package that makes it possible to create and run programs on a simulated robot, with some limitations. The limitation of this solution is that it is capable to simulate only one robot. There is no chance to integrate the robot in complex environments as you can configure with Gazebo e.g., interacting with other mechanical elements in the workspace, check collisions with the environment, etc.

Another approach is that the system can be simulated using Gazebo. The gazebo_ros_control package (1.6.5.1) can be used instead of the ur_modern_driver to implement the interfaces of the hardware resource layer, alternative to sending robot control messages using TCP, the DefaultRobotHWSim (part of gazebo_ros_control) simply sets the simulated velocities directly, using the Gazebo plugin API. This approach unfortunately can not simulate the network, because DefaultRobotHWSim lacks the

ability to do so.

In the next chapter I propose a method using a custom RobotHWSim plugin that replaces DefaultRobotHWSim which can simulate changing network characteristics.

3.2 Introducing methods to simulate the effects of network characteristics

In ROS, topics are named buses over which nodes exchange messages. ROS currently supports TCP/IP-based and UDP-based message transport. ROS nodes are standalone executables running with individual process IDs in the operating system. One practical way to introduce latency in current ROS deployment is via defining network namespaces among nodes. For a certain namespace, custom delay, jitter, drop characteristics can be defined with tc like in [43]. The main issue is that there is a MoveIt node as an individual process, but the whole joint controller-actuator control loop is realized within Gazebo as one other process, because gazebo_ros_control and the whole ros_control infrastructure uses pluginlib (described in 1.5.3.4 to load each other. The only topic based communication happens between the MoveIt and the monolith Gazebo process. So this kind of solution cannot be applied to the problem.

We have to dig deeper in the architecture of Gazebo and realize the CPS system within. To keep the architecture modular, I decided to implement the proposed method as a Gazebo plugin. While the setup most of these plugins fits well in the current Gazebo architecture and can be done via configuration files, there are still patches needed to be applied on core functional elements of the Gazebo code.

Figure 3.2. shows the architecture of the proposed method. The coloring of the figure follows the way in 1.6.2. Green represents new added plugins, modules, functionalities. The system works the following way.

As a first step, a launch file (1.6.7) that triggers the whole simulation to run setups a parameter on the ROS parameter server (1.5.2.2). This parameter defines the specific latency plugin that will be loaded.

The launch file initiates the Gazebo simulation. Gazebo loads the gazebo_ros_control plugin (left most blue box) that main purpose is to interface with the ROS controller manager. The RobotHWSim interface defined by this module needed a small tweak. In its `readsim()` function, instead of passing the time as value, I modified it to pass by reference allowing modification by plugins.

Gazebo loads configuration files from the `common.gazebo.xacro` file in which it is specified that mycustom `RobotHWSimLatency` plugin should be loaded instead of the `DefaultRobotHWSim` plugin. My `RobotHWSimLatency` plugin is the extension of the `DefaultRobotHWSim` plugin with modified read and write functions and with the task

to load a custom latency plugin. The latency plugin to be loaded is the one that was setup by the parameter server. My `RobotHWSimLatency` plugin also had to modify the way it handled communicated with hardware_interfaces. The original code of `DefaultRobotHWSim` passed the address of the variables that stored the state of joints to the hardware_interface layer during startup, there was no modification of these variables during the working of the plugin that changed the addres, so the hardware_interface layer could always access them. In my system, the variables are written in a way that the pointers that used to point to them are now invalid, so the hardware_interface layer can not access them anymore. I modified the code to pass addresses of separate variables to hardware_interface — that are separate from the variables I modify — and copy these modifications to them in a way that does not invalidate their addresses.

The current latency plugin options include a) the default latency plugin that practically returns the messages with no introduced latency and b) the simple queue latency plugin. This later has a configurable size of the queue to store the messages in them. In each simulation tick (100Hz), the messages are shifted one position forward in the queue and when they reach the end of the queue they are provided to Gazebo as the currently valid message. In the same way, an interface plugin to cooperate with external network simulators like ns3 [44] can be also implemented here.

The detailed working mechanism and call sequence of the plugin system is the following:

1. The gazebo_ros_control update function fires
2. It calls the readSim function; the call is executed in the RobotHWSimLatency plugin which implements the readSim function
3. The states are read from the gazebo internals
4. The delayStates function is called in the Simple queue latency plugin that saves the state messages in a buffer
5. The previously stored and now delayed states are returned from the Simple queue latency plugin to the RobotHWSimLatency plugin
6. readSim writes the joint_states to the JointStateInterface of the Harware Resource Interface Layer
7. gazebo_ros_control calls the update function of the controler_manager
8. The joint_trajectory_controller in the controller manager executes the calculation of the PID-controllers
9. The joint_trajectory_controller writes the calculated velocity commands to the VelocityInterface of the Hardware Resource Interface Layer

10. The gazebo_ros_control calls the writeSim function which is implemented in the RobotHWSimLatency plugin
11. The writeSim function reads the joint commands from the VelocityInterface of the Hardware Resource Interface Layer
12. The writeSim function calls the delayCommands function of the Simple queue latency plugin
13. The previously stored and now delayed commands are returned from the Simple queue latency plugin to the RobotHWSimLatency plugin
14. The writeSim function writes the joint commands to the gazebo internals
15. Gazebo calculates the internal states of the simulation loop
16. A new simulation loop is started by calling the update function of the gazebo_ros_control plugin

The source code of my plugin is available on Github [45].

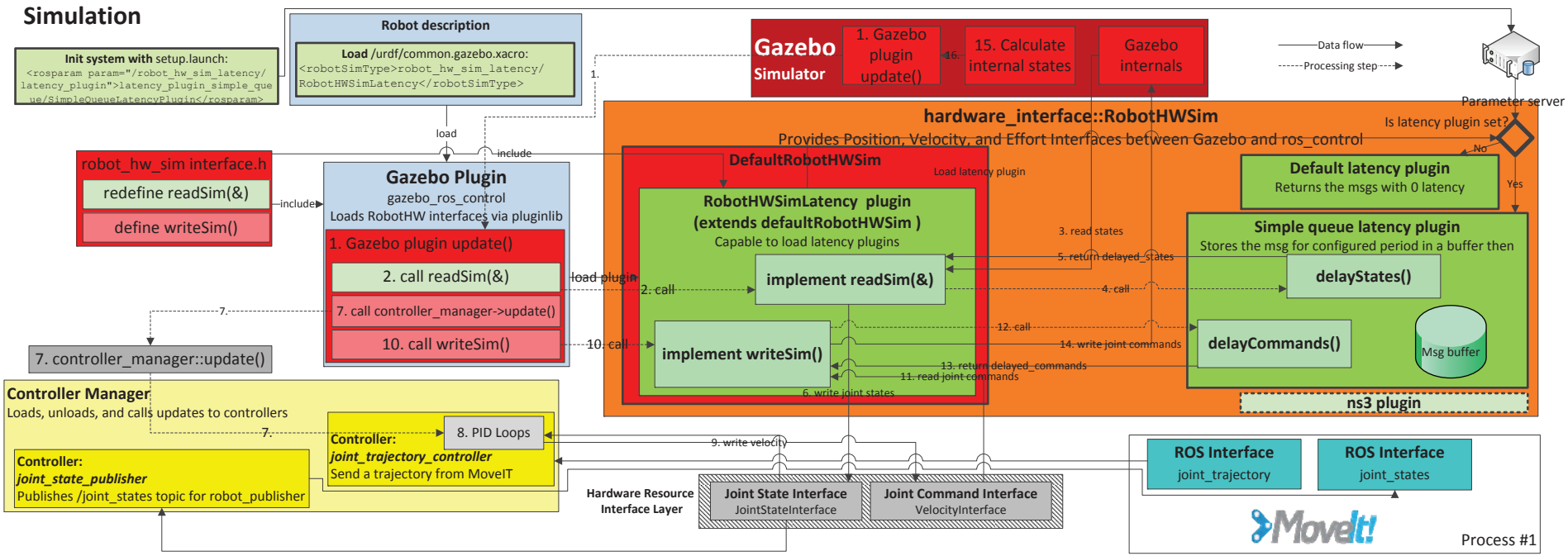

Figure 3.2: *Gazebo architecture*

Chapter 4

Measurements

In this chapter I evaluate the effects of the simulated network latency — added to the CPS by my plugin — on various KPIs.

4.1 Evaluation

I evaluated my proposed method on various Key Performance Indicators (KPIs). The most straightforward evaluation is the visual inspection of the robotic arm movement. For this purpose, I loaded the robot model into rviz and used a ros package (hector_trajectory_server 1.6.8) to visualize the path the end of the arm took.

Figure 4.1. is a screenshot from rviz which shows the visualized trajectories. The bottom left corner of the picture is the starting point of the robotic arm. It passes through the waypoints one-by-one from number 1 to 5. The black lines are the trajectories, while the lines with various colors show the effect of introducing latency into the system. The cyan color shows the reference scenario with 0 latency. In all other cases, I introduced latency in the system in both the command writing and status reading direction and rerun the trajectory planning and execution scenario. The upper right corner of the picture shows a magnified part around the trajectories.

The trajectories were planned with the `RRTConnectkConfigDefault` planner MoveIt plugin [46] which utilizes Rapidly-exploring Random Trees therefore — being non-deterministic because of its random nature —, it sometimes created wildly differing trajectories making it difficult to compare them.

The visualized trajectories show the expected behavior of the system. Increasing the latency increases the deviance from the original trajectories. It should be noted that the planned trajectories are straight in Cartesian-space. To move along these trajectories the robotic arm needs complex movements in the joint-space, thus even the movement in a straight line causes deviation from the reference trajectory. In the other way around, if the

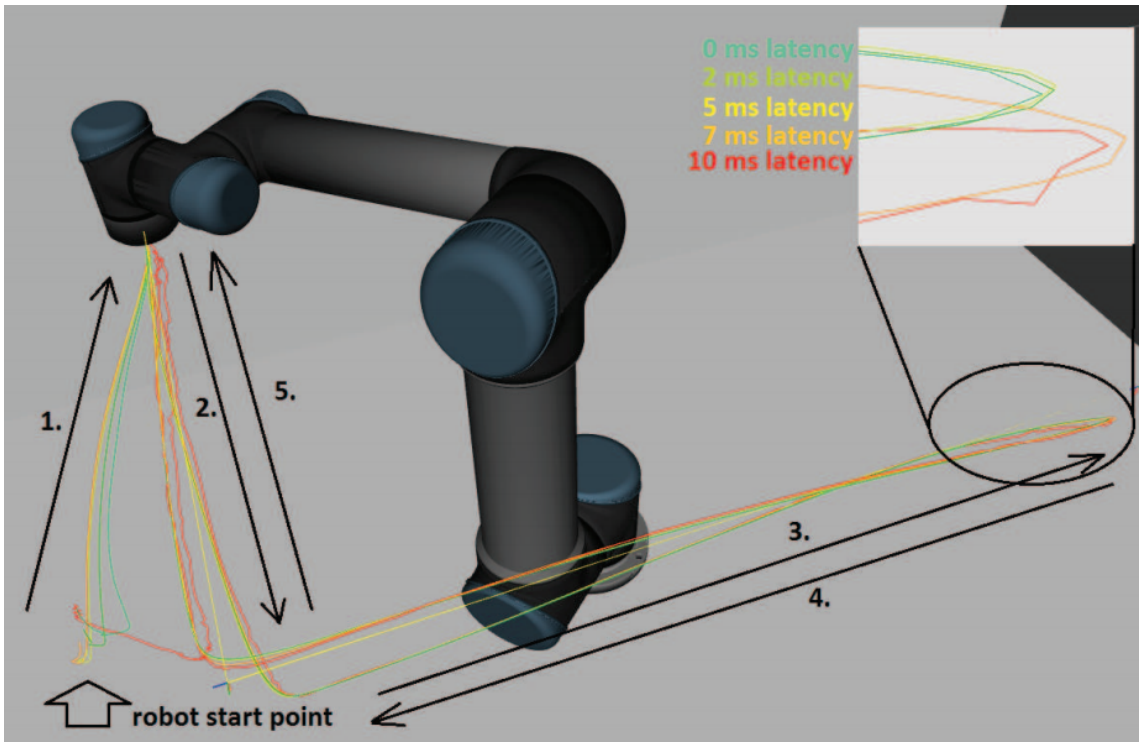

Figure 4.1: *The visualized trajectories*

planned trajectories were straight in the joint-space, I would see a movement in circles by the robotic arm, but the effect of the latency was more negligible.

Figure 4.2. shows the velocity commands sent to the robot in the function of time. Analyzing the velocity commands in such details reveals that comparing the different scenarios are not straightforward for several reasons. One is that the planning is non-deterministic, and a slight difference during the initialization of the gazebo environment ends up with some different planned trajectory. The execution of the trajectories depends on the environment status as well, and it is never the same. Joint 4 shows the expected effect on the velocity commands levels as well, thus the induced latency causes increased velocity command deviation compared to the reference scenario. It is also a clear observation that around 10 ms latency, the system starts to get unstable. This is likely due to the various updating frequency parameters that Gazebo employs to run the simulation. It needs definitely further work to make it clear how the introduced latency affects other characteristics or behaviors, such as the robot commanding frequency, whole physical simulation steps, internal message timings.

Figure 4.3. shows the cumulated difference of the velocity commands comparing to the reference scenario. The 2 ms latency scenario is the closest to the reference as it is expected. In the first 3 sec of the trajectory execution the 5 ms scenario is closer to the reference than the 7 ms scenario, but around 6 sec, the 5 ms scenario collects so much

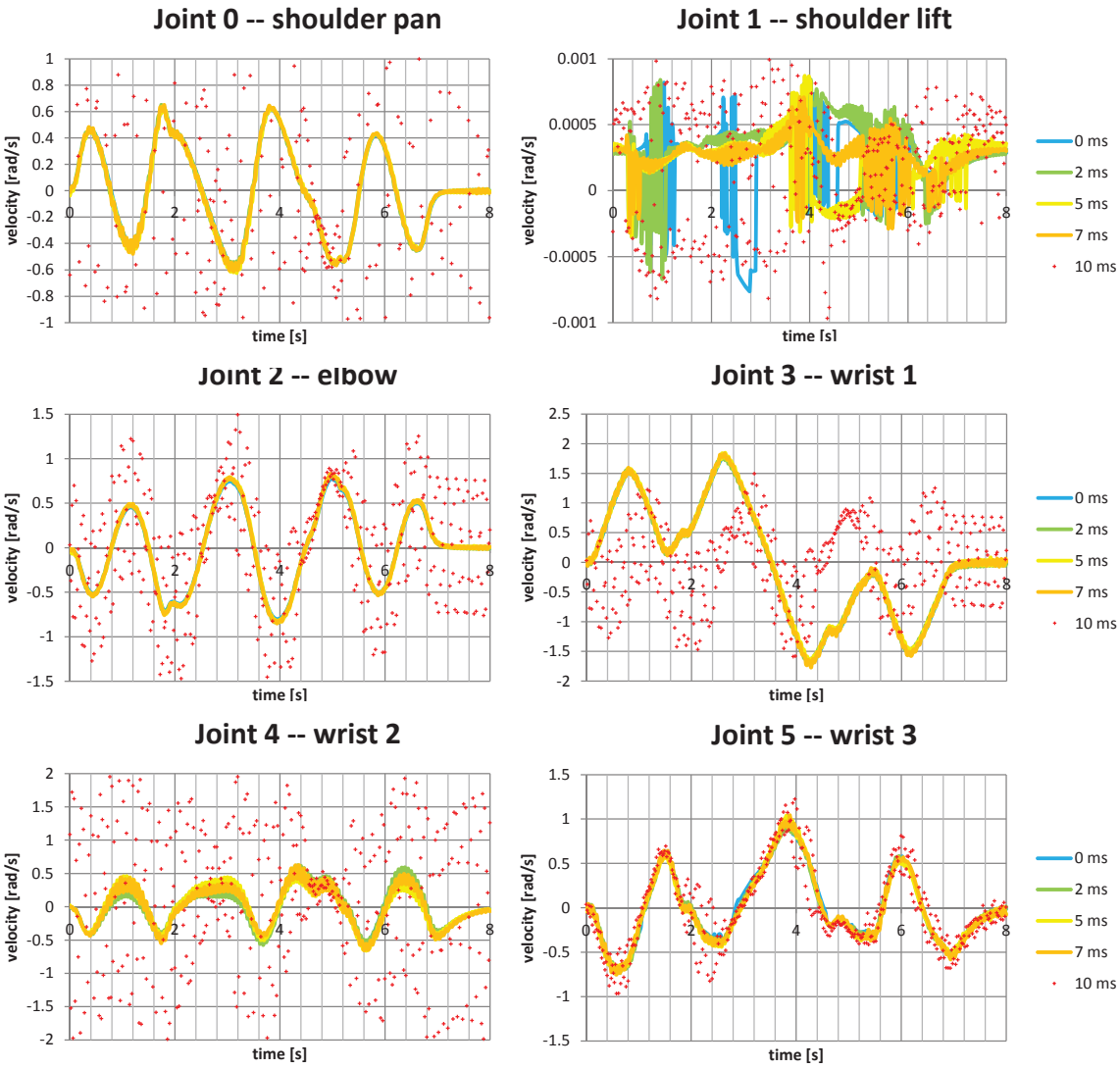

Figure 4.2: *The velocity commands sent to the robot*

error that shows bigger deviation than the 7 sec scenario. The 10 sec scenario has another magnitude of error, and thus cut off the diagram after the first second.

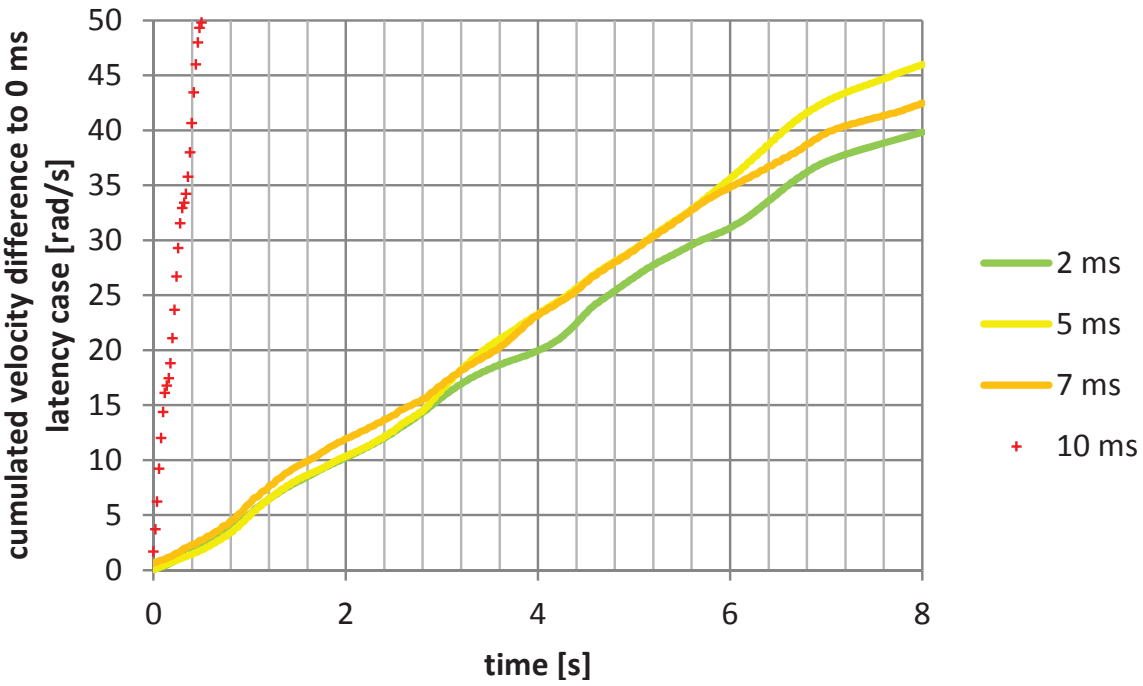

Figure 4.3: *The cumulated difference of the velocity commands comparing to the reference scenario*

Chapter 5

Conclusion and further work

In this paper, I proposed a plugin [45] to extend the capabilities of the current Gazebo robotic simulator and turn it into a CPS system. The realization of the proposed method is a plugin to Gazebo. The plugin fits into the modular design of Gazebo. As of the interface is available, it eases to test various network effects on the robot control. Based on my preliminary evaluations it does affect the QoC KPIs of the robot control.

Further work

The evaluation showed behavior which is expected and reasonable, but also cases which show that the whole system needs fine-tuning.

There are multiple avenues for further research:

- Evaluate the working mechanism of the system with the help of the ROS, gazebo and research communities.
- More extensive measurements with the tool.
- Interface the tool with various radio network simulators and see the effects of the radio on the QoC KPIs.
- Investigate how the system behaves, when taking into account not only the network link characteristics but also the protocols for message exchanging.
- Comparing the level of similarity of the simulation to real robot HW controlled in a real radio network.
- Taking part in the ARIAC competition to evaluate if the tool can provide any advantage in any of the use cases of the competition.

Bibliography

[1] Gazebo Robot Simulator. http://gazebosim.org.

[2] UR5 Robot Arm. https://www.universal-robots.com/products/ur5-robot/.

[3] URScript. https://www.universal-robots.com/how-tos-and-faqs/how-to/ur-how-tos/ethernet-socket-communication-via-urscript-15678/â??, 2017.

[4] Main website of Robot Operating System. http://www.ros.org/.

[5] Introduction to ROS, ROS Wiki. http://wiki.ros.org/ROS/Introduction.

[6] Is ROS for me? http://www.ros.org/is-ros-for-me/.

[7] ROS Concepts, ROS Wiki. http://wiki.ros.org/ROS/Concepts.

[8] ROS Packages, ROS Wiki. http://wiki.ros.org/Packages.

[9] ROS Metapackages, ROS Wiki. http://wiki.ros.org/Metapackages.

[10] ROS Package Manifest, ROS Wiki. http://wiki.ros.org/Manifest.

[11] ROS Msg Files, ROS Wiki. http://wiki.ros.org/msg.

[12] ROS Msg Files, ROS Wiki. http://wiki.ros.org/srv.

[13] ROS Master, ROS Wiki. http://wiki.ros.org/Master.

[14] ROS Technical Overview, ROS Wiki. http://wiki.ros.org/ROS/TechnicalOverview.

[15] ROS Parameter Server, ROS Wiki. http://wiki.ros.org/Master.

[16] ROS Nodes, ROS Wiki. http://wiki.ros.org/Nodes.

[17] ROS Messages, ROS Wiki. http://wiki.ros.org/Messages.

[18] ROS Standard messages, ROS Wiki. http://wiki.ros.org/std_msgs.

[19] ROS Common Messages, ROS Wiki. http://wiki.ros.org/common_msgs.

[20] ROS Services, ROS Wiki. http://wiki.ros.org/Services.

[21] ROS actionlib package, ROS Wiki. http://wiki.ros.org/actionlib.

[22] ROS Bags, ROS Wiki. http://wiki.ros.org/Bags.

[23] ROS Client Libraries, ROS Wiki. http://wiki.ros.org/ClientLibraries.

[24] ROS Coordinate Frames, ROS Wiki. http://wiki.ros.org/tf.

[25] ROS URDF, ROS Wiki. http://wiki.ros.org/urdf.

[26] ROS Plugins, ROS Wiki. http://wiki.ros.org/pluginlib.

[27] ROS catkin Build system, ROS Wiki. http://wiki.ros.org/catkin.

[28] ROS Distributions, ROS Wiki. http://wiki.ros.org/Distributions.

[29] ROS Wiki, ROS Wiki. http://wiki.ros.org/.

[30] ROS Names, ROS Wiki. http://wiki.ros.org/Names.

[31] ROS universal_robot package, ROS Wiki. http://wiki.ros.org/universal_robot.

[32] ROS ros_control packages, ROS Wiki. http://wiki.ros.org/ros_control.

[33] MoveIt. http://moveit.ros.org/.

[34] ROS gazebo_ros_pkgs package, ROS Wiki. http://wiki.ros.org/gazebo_ros_pkgs.

[35] ROS xacro package, ROS Wiki. http://wiki.ros.org/xacro.

[36] ROS roslaunch package, ROS Wiki. http://wiki.ros.org/roslaunch.

[37] Network Emulation at the DARPA Robotics Challenge. https://iwl.com/white-papers/network-emulation-at-the-darpa-robotics-challenge/, 2017.

[38] Agile Robotics for Industrial Automation Competition (ARIAC). http://gazebosim.org/ariac, 2017.

[39] S. Ivaldi, J. Peters, V. Padois, and F. Nori. Tools for simulating humanoid robot dynamics: A survey based on user feedback. In *2014 IEEE-RAS International Conference on Humanoid Robots*, pages 842–849, Nov 2014.

[40] How to link OMNET++/Castalia with ROS. http://cpham.perso.univ-pau.fr/WSN-MODEL/castalia-ros.html, 2017.

[41] Thomas Timm Andersen. Optimizing the universal robots ros driver. Technical report, Technical University of Denmark, Department of Electrical Engineering, 2015.

[42] URSim. https://www.universal-robots.com/download/?option=28545#section16632, 2017.

[43] Network Namespaces and Traffic Control. http://gigawhitlocks.com/2014/08/18/network-namespaces.html, 2014.

[44] ns-3. https://www.nsnam.org/, 2017.

[45] Gazebo latency plugin. https://github.com/Ericsson/robot_hw_sim_latency, 2017.

[46] J. J. Kuffner Jr. and S. M. Lavalle. Rrt-connect: An efficient approach to single-query path planning. In *Proc. IEEE Intâ??l Conf. on Robotics and Automation*, pages 995–1001, 2000.