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

Analysing The Effects of Network Characteristics in Cyber Physical Production Systems

Students' Scientific Conference paper

Author
József Pető

Department Supervisors
Dr. Sándor Molnár
Associate Professor

Dr. Attila Vidács
Associate Professor

Industrial Supervisor
Dr. Géza Szabó
Ericsson Ltd.

October 27, 2018

Contents

Összefoglaló

A robotszimuláció egy nélkülözhetetlen eszköz minden robotikával foglalkozó eszköztárában. Egy jól tervezett szimulátor segítségével gyorsan tudunk algoritmusokat tesztelni, robotokat tervezni, regressziós teszteket végezni és Mesterséges Intelligencia rendszereket tanítani valósszerű környezetben.

Egy népszerű robot szimulátorral, a Gazeboval[1] képesek vagyunk pontosan és hatékonyan szimulálni robotokat komplex beltéri és kültéri környezetben. Erős fizikai motort, kiváló minőségű grafikát és kényelmesen használható programozói és felhasználói interfészeket nyújt. Azonban a Gazebo-ból hiányzik a vezérlési késleltetés modellezése, ami egy teljes értékű kiberfizikai rendszerszimulátorrá tenné.

Ebben a dolgozatban bemutatok egy Gazebo plugint ami képessé teszi a Gazebo-t, arra hogy késleltetést szimuláljon. Ezáltal lehetővé téve különféle, a késleltetések hatását vizsgáló mérések elvégzését szimulált környezetben.

Ezzel a pluginnal több mérést is végrehajtok. Először vizsgálom egy hat szabadságfokú robotkar viselkedését különböző hálózati késleltetések mellett.

A plugint használva hálózati késleltetést állítok be az Agile Robotics for Industrial Automation Competition (ARIAC)[2] környezetében, ami egy szimuláción alapuló verseny, automatizált teljesítmény pontozással. Egy a versenyben részt vett csapat, a Figment Team megoldását értékelem ki hálózati késleltetés esetén.

A plugint egy másik környezettel is kipróbáltam, egy szimulált hatlábú robottal. Ez a robot hasonló egy Ipar 4.0 robotcellához amiben van hat 3 szabadságfokú kar. Ebben a környezetben számos kihívás megjelenhet (például: szervók vezérlése, robotkarok kollaborációja stb.). 5 ms késleltetés a hatlábú robot vezérlésében az egyenes mozgástól eltérést okozott.

A plugin használata különböző környezetekben demonstrálja, hogy ez az általam fejlesztett plugin alkalmas hálózati hatások vizsgálatára. És jövőben segíthet kiberfizikai gyártó rendszerek hálózatainak tervezésében a hálózatüzemeltetőknek.

A dolgozat további eredményekkel bővítve a következő helyeken jelent meg: [3] [4] [5]

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

I propose a Gazebo plugin to make Gazebo capable to simulate delay. Using this plugin one can make measurements examining the effects of latency in simulated systems.

Using this plugin I make several measurements. First, I examine the behaviour of a 6-DOF arm in various latency scenarios.

I use this plugin to introduce network latency effects to the environment of Agile Robotics for Industrial Automation Competition (ARIAC)[2], which is a simulation-based competition, with automated performance scoring. I evaluate performance of the Figment Team's solution in the competition with added network latency.

I also use this plugin in a simulated environment with a hexapod robot, which can be used as an analogue to a Industry 4.0 robot cell with six 3-DOF robotic arm. In this environment a wide spectrum of the challenges can arise in e.g., servo control, collaboration, etc. I found creating 5 ms of delay caused the hexapod robot to drift from the desired direction.

The application of the plugin in various use cases demonstrates that my proposed plugin is useful for the evaluation of network effects. In the long run network operators can use my extended simulation environment to provision networks in Cyber Physical Production Systems (CPPS).

Extensions of this work were published in the following works: [3] [4] [5]

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 [1].

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 is 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 is 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 analyse 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. I also show the usefulness of my plugin in several different environments. I show that it can be used to evaluate the performance of CPS systems in different network scenarios.

This paper is a continuation of my previous Students' Conference Paper[6], in which I only present my plugin, and one test scenario.

Structure of the paper

The paper consists of 6 chapters.

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

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

In Chapter 3 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.

In Chapter 4 I evaluate the effects of the simulated network latency — added to the CPS by my plugin — on various standard KPIs and on a solution to ARIAC.

In Chapter 5 I use a hexapod robot platform to demonstrate the usage of my plugin in measuring the effects of latency on the robot.

Finally, I conclude this paper in Chapter 6 and describe avenues for further research.

Chapter 1

Background

In this chapter I describe the Gazebo robot simulation environment, the UR5 robot, the Robot Operating System, its various concepts, and used ROS Packages.

1.1 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[7].

1.2 UR5 Robot arm

The UR5[8] 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[9]. By sending commands you can control the robot's Cartesian position, velocity, joint angle and velocity.

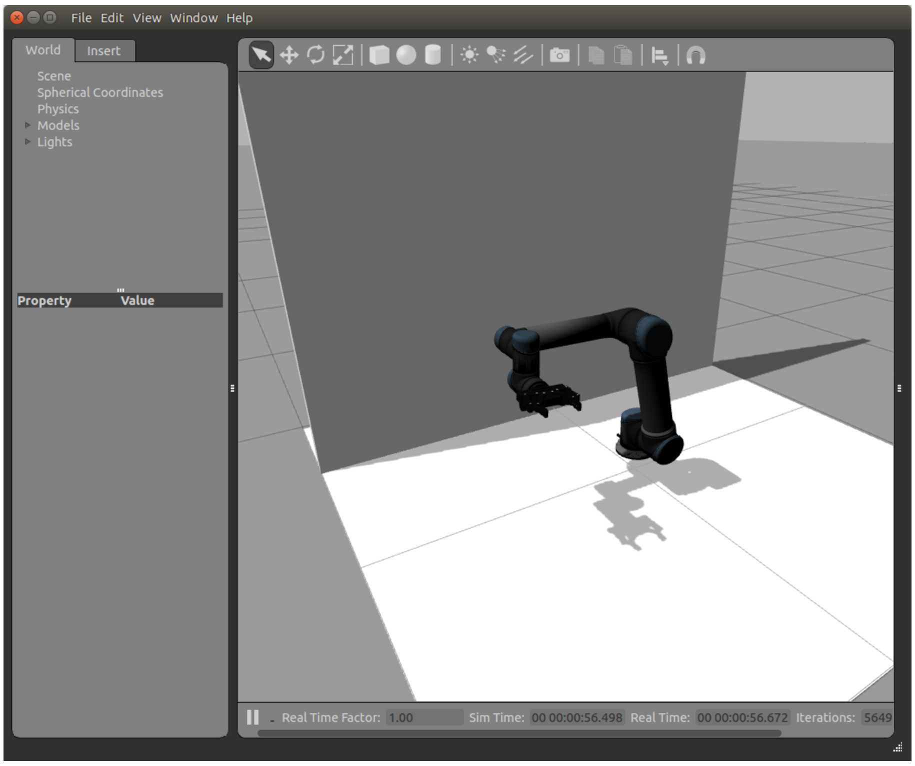


Figure 1.1: *Gazebo simulation of the UR5 robot arm*

1.3 Robot Operating System

Robot Operating System (ROS)[10] 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[11, 12].

1.3.1 Packages

Packages[13] 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.

1.3.2 Master

The ROS Master[14] acts as a nameservice ROS. 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[15], 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.

1.3.3 Parameter Server

The parameter server[16] 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[15]. 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).

1.3.4 Nodes

Nodes[17] 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[15]. 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.

1.3.5 Messages

Nodes communicate with each other by publishing messages to topics[18]. 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.

1.3.6 Standard and common messages

The std_msgs package[19] 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[20] 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.

1.3.7 Topics

Messages are routed via a transport system with publish / subscribe semantics[20]. 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:[15]

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.

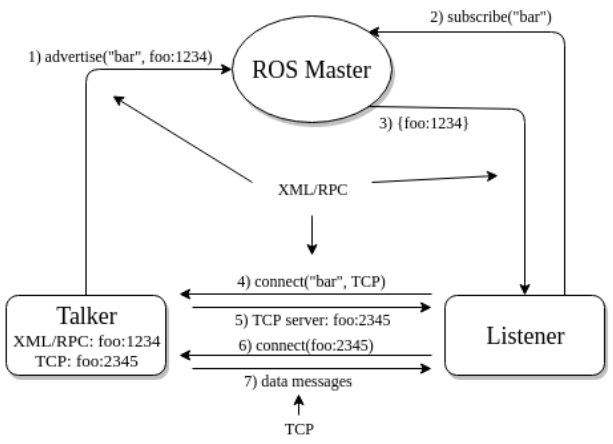


Figure 1.2: *The sequence of connection [15]*

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.

1.3.8 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[21] , 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.

1.3.9 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[22] 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.

1.3.10 Coordinate frames

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

tf[23] (and its successor tf2[23]) 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.

1.3.11 Unified Robot Description Format

The Unified Robot Description Format (URDF)[24] 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.

1.3.12 Plugins

The pluginlib[25] 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.

1.4 Used ROS packages

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

1.4.1 universal_robot package

The universal_robot metapackage[26] 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).

1.4.1.1 ur_description

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

1.4.1.2 ur_gazebo

This package contains files that aid in starting a robot simulation

1.4.2 ros_control package

Ros_control[27] is a set of packages defining a set of interfaces, which are designed to abstract away differences between robot hardware.

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.

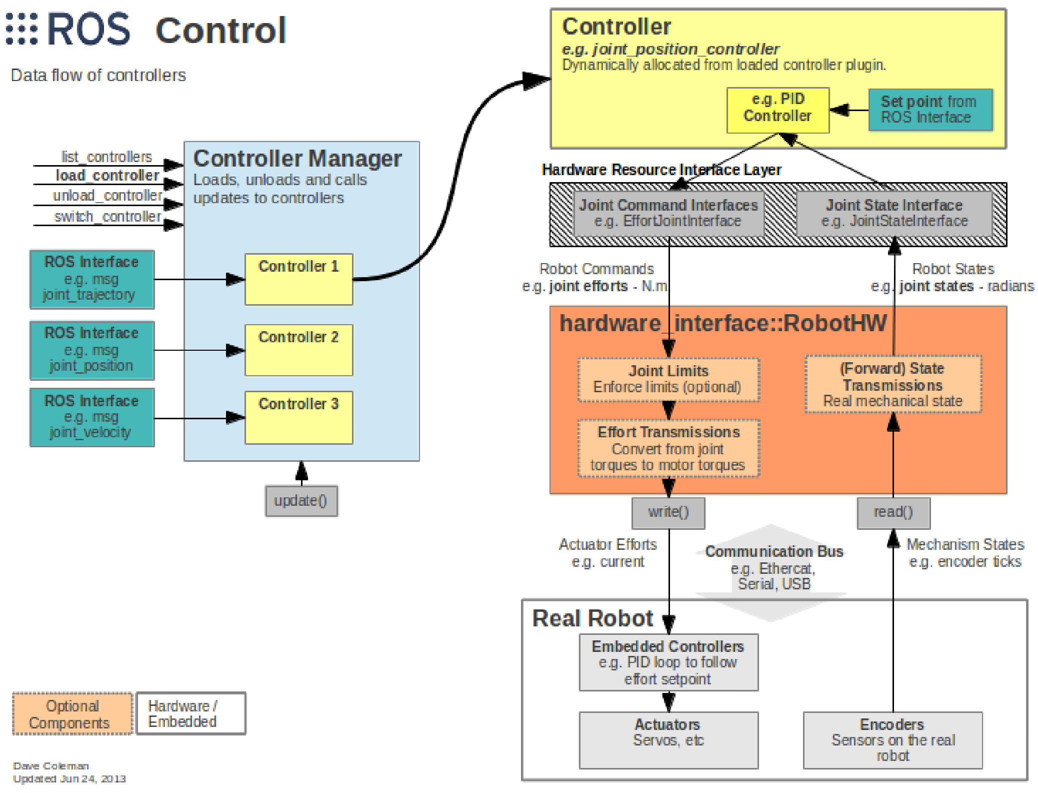

ROS Control
Data flow of controllers
list_controllers
load_controller
unload_controller
switch_controller
Controller Manager
Loads, unloads and calls updates to controllers
ROS Interface e.g. msg joint_trajectory
ROS Interface e.g. msg joint_position
ROS Interface e.g. msg joint_velocity
Controller 1
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

Figure 1.3: *Overview of ros_control[27]*

1.4.2.1 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.

1.4.2.2 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.

1.4.3 moveit package

MoveIt![28] 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.

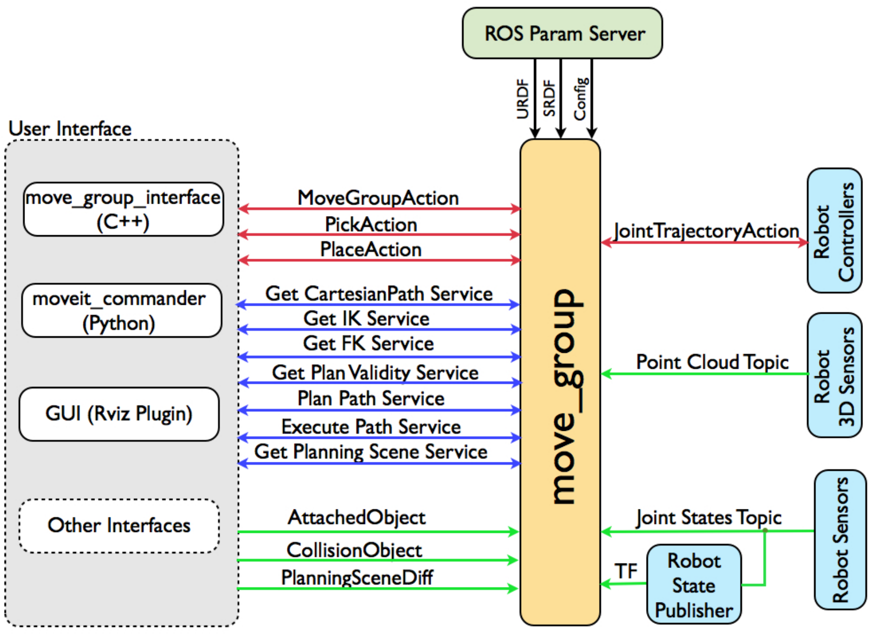


Figure 1.4: *Moveit architecture*

1.4.4 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 elaborate trajectory — a simple trajectory between 3 points.

1.4.5 gazebo_ros_pkgs package

gazebo_ros_pkgs[29] 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.

1.4.5.1 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.

1.4.6 xacro package

The xacro package[30] 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.

1.4.7 roslaunch package

roslaunch[31] 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.

1.4.8 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

In this chapter I describe the motivations behind this paper, and present related works. Parts of this chapter is from my previous Students' Conference Paper[6].

2.1 Cyber Physical System (CPS)

One of the most significant directions in the development of computer science and information and communication technologies is represented by Cyber-Physical Systems (CPSs) which are systems of collaborating computational entities which are in intensive connection with the surrounding physical world and its on-going processes, providing and using, at the same time, data-accessing and data-processing services available on the internet. [32]

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,

2.2 Cyber-Physical Production Systems CPPS

The scoping of general CPS into the industry domain introduces the concept of Cyber-Physical Production Systems (CPPSs). Author of [33] defines CPPS as system where mechatronic components are coupled to a smart logical entity that enables these factory units to interact in an adaptive way. Both [33] and [32] collect various research challenges. According to [32], one of the research challenges of CPPS is the fusion of real and virtual systems. The development of new structures and methods are required which support the fusion of the virtual and real sub-systems in order to reach an intelligent production system which is robust in a changing, uncertain environment. Novel reference architectures and models of integrated virtual and real production subsystems; the synchronization of the virtual and real modules of production systems and their role-specific interaction; and context-adaptive, resource efficient shop floor control algorithms are needed.

2.3 Traditional characteristics of robots

The work [34] describes that an industrial robot has many metrics and measurable characteristics, which will have a direct impact on the effectiveness of the robot during the execution of its tasks. The main measurable characteristics are repeatability and accuracy. In a nutshell, the repeatability of a robot might be defined as its ability to achieve repetition of the same task. While, accuracy is the difference (i.e. the error) between the requested task and the realized task (i.e. the task actually achieved by the robot). Practically, repeatability is doing the same task over and over again, while accuracy is hitting your target each time. For more details about the calculation of accuracy and repeatability, see [35]. The ultimate objective is to have both; a robot that can repeat its actions while hitting the target every time. When the current mass production assembly lines are designed, robots are deployed to repeat a limited set of tasks as accurately and the fastest possible way to maximize the productivity and minimize the number of faulty parts. The reprogramming of the robots rarely occurs (per week, per month basis) and it takes a long time (even days) and it is a difficult task requiring lot of expertise.

2.4 Network aspects

The article [36] compares the network protocols used nowadays in industry applications for example, Modbus, Profinet, Ethercat. All investigated Industrial Ethernet (IE) systems show similar basic principles, which are solely implemented in different ways. Several solutions apply a shared memory and most systems require a master or a comparable management system, which controls the communication or have to be configured manually. Shared memory is implemented via data distribution mechanisms that are based on a high frequency packet sending patterns. These packets have to be transmitted with strict delivery time with minimum jitter. IE protocols rely so heavily on the transport network that protocol mechanisms common in broadband usage like reliable transmission, error detection, etc., are not among the basic features of industrial protocols.

Authors of [37] summarize the fundamental trade-offs in 5G: finite vs. large blocklength, spectral efficiency vs. latency, device energy consumption vs. latency, energy expenditures vs. reliability, reliability vs. latency and rate, SNR vs. diversity, short/long TTI vs. control overhead, open vs. closed loop, user density vs. dimensions (antennas, bandwidth, block length). There are numerous aspects that have to be solved during an industry automation task even when the robot stands still. A remote robot control application induces the frequent transmission of velocity or effort commands throughout the whole production time e.g., in a 125 Hz frequency [38] in the case of UR5.

2.5 Robot cell optimization

The work [39] is a survey of the numerous papers and approaches in the industry aiming to optimize the operation of a robot cell. The purpose of such optimization is to minimize or maximize at least one of the following objective functions: 1) minimizing the execution time, respectively maximizing the robot productivity, considering that the relative speeds of the actuator's elements are limited constructively; 2) minimizing the energy consumption or mechanical work necessary for execution, leading to a reduction of the mechanical stresses in actuators and on the robot structure and obtaining smooth trajectories, easy to follow; 3) minimizing the maximum power required for operating the robot; 4) minimizing the maximum actuation forces and moments. The most common optimization criteria used in the literature are: minimum time trajectory planning; minimum energy trajectory planning or minimum actuation effort and minimum jerk trajectory planning.

The remote control of a robotic cell via wireless is a new type of challenge that the above optimization strategies miss yet.

2.6 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. [40] 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)[2] 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?

2.7 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 [41]. It showed that Gazebo emerges as the best choice among the open-source projects.

Authors of [42] describes some early experiments in linking the OMNET++ simulation framework with the ROS middleware for interacting with robot simulators. 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, to introduce the effects of the network simulator into the robot simulator.

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. This chapter is from my previous Students' Conference Paper[6]

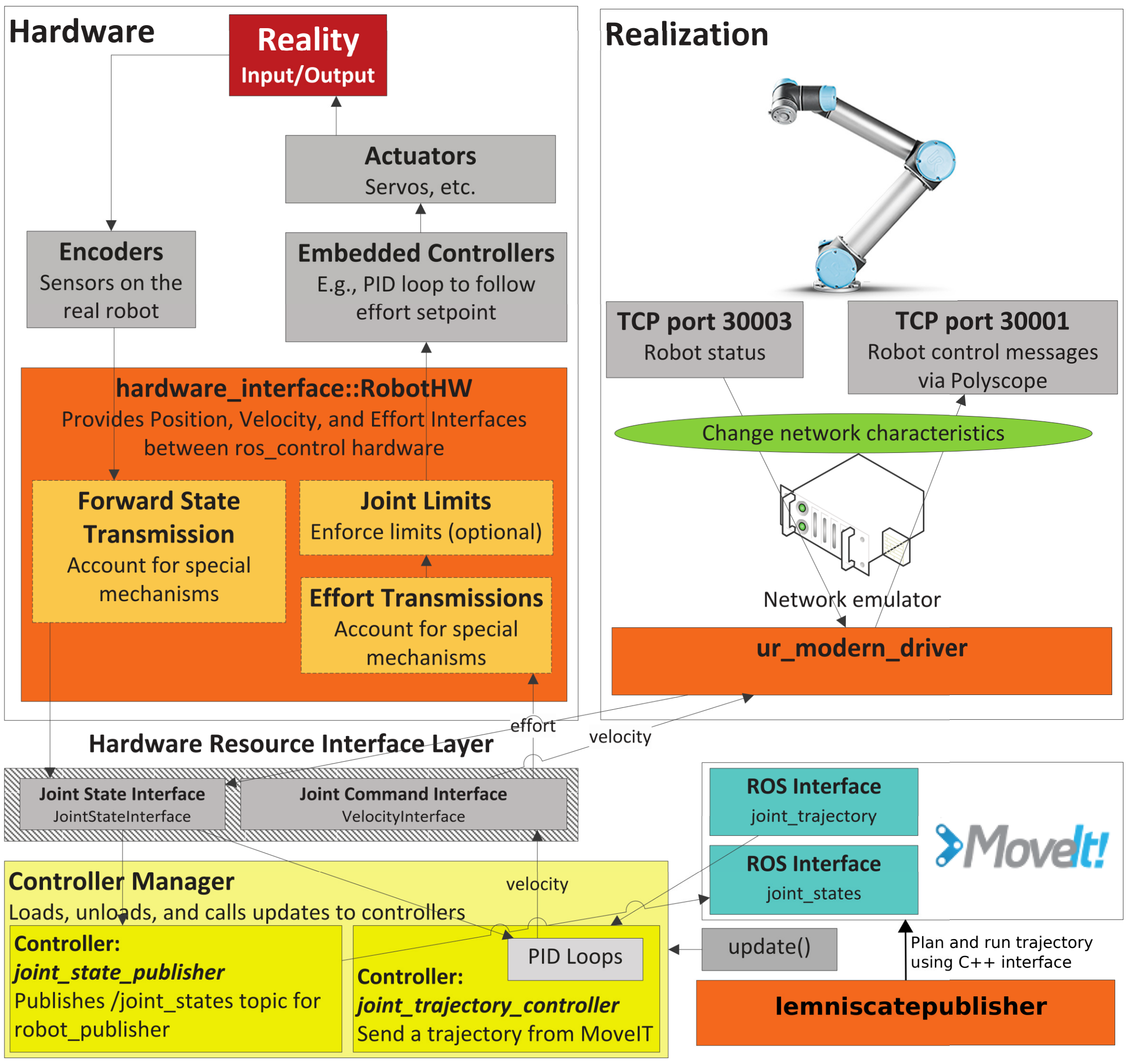
3.1 Overview

The CPS that I address to measure is a robotic arm (UR5 [8]) 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 (1.4.2), 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 1.4.4) 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 (1.4.2.2)(shown in yellow) which at the start of simulation was started by the controller manager.

The ur_modern_driver [38] 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-



Hardware
Reality
Input/Output
Actuators
Servos, etc.
Encoders
Sensors on the real robot
Embedded Controllers
E.g., PID loop to follow effort setpoint
hardware_interface::RobotHW
Provides Position, Velocity, and Effort Interfaces between ros_control hardware
Forward State Transmission
Account for special mechanisms
Joint Limits
Enforce limits (optional)
Effort Transmissions
Account for special mechanisms
Realization
TCP port 30003
Robot status
TCP port 30001
Robot control messages via Polyscope
Change network characteristics
Network emulator
ur_modern_driver
effort
velocity
Hardware Resource Interface Layer
Joint State Interface
JointStateInterface
Joint Command Interface
VelocityInterface
ROS Interface
joint_trajectory
ROS Interface
joint_states
MoveIt!
Controller Manager
Loads, unloads, and calls updates to controllers
velocity
Controller:
joint_state_publisher
Publishes /joint_states topic for robot_publisher
PID Loops
Controller:
joint_trajectory_controller
Send a trajectory from MoveIT
update()
Plan and run trajectory using C++ interface
lemniscatepublisher

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

vided by Universal Robots. Universal Robots simulator software [43] 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 (1.4.5.1) 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

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 [44]. 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 1.3.12) 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 1.4.2. Green represents new added plugins, modules, functionalities. The system works the following way.

As a first step, a launch file (1.4.7) that triggers the whole simulation to run uploads a parameter on the ROS parameter server (1.3.3). 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) whose 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 uploaded 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.

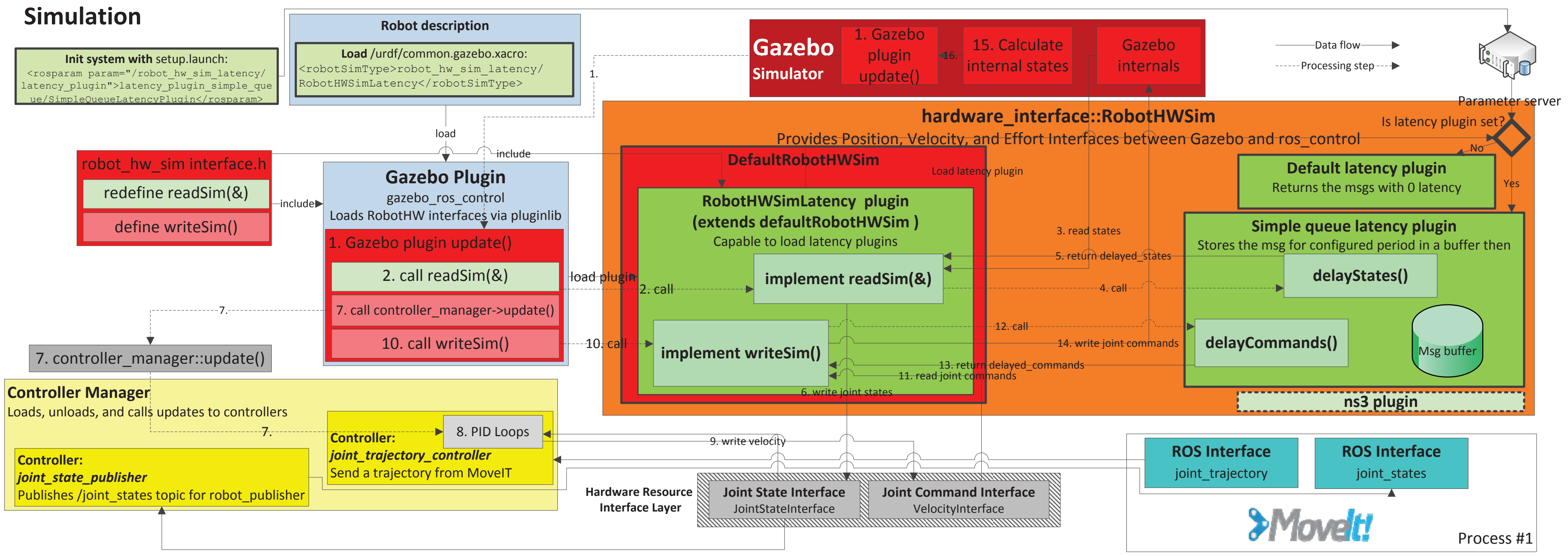
The current latency plugin options include a) the default latency plugin that practically returns the messages with no introduced latency and b) the simple queue latency plugin. This latter has a configurable size of the queue to store the messages in them. In each simulation tick (100Hz), the messages 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 [45] 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 [46].



```
<rosparam param="/robot_hw_sim_latency/
latency_plugin">latency_plugin_simple_que
ue/SimpleQueueLatencyPlugin</rosparam>
```

```
<robotSimType>robot_hw_sim_latency/
RobotHWSimLatency</robotSimType>
```

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 and on a solution to ARIAC. The first part is from my previous Students' Conference Paper[6].

4.1 Evaluation with standard robot KPIs

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 1.4.8) 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 [47] 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

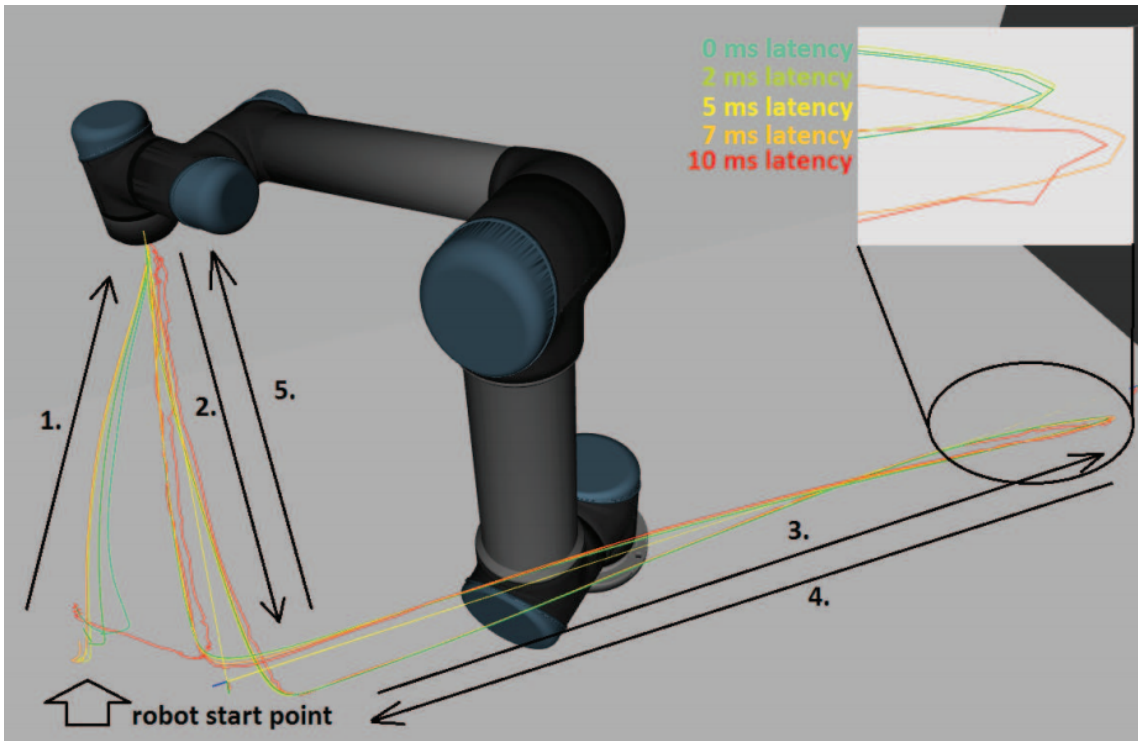


Figure 4.1: *The visualized trajectories*

straight line causes deviation from the reference trajectory. In the other way around, if the 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 would be 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 trajectories. 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

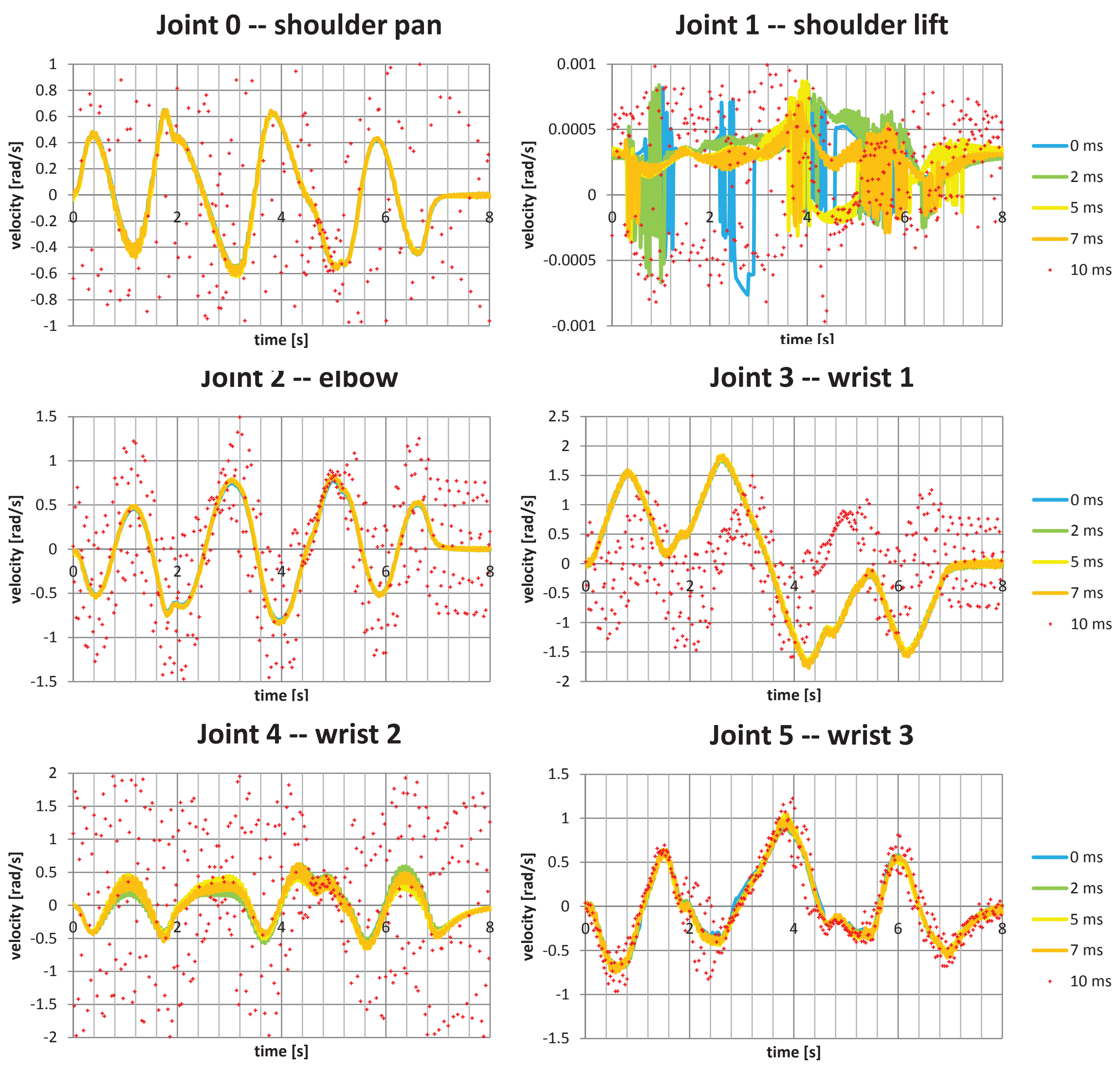


Figure 4.2: *The velocity commands sent to the robot*

reference than the 7 ms scenario, but around 6 sec, the 5 ms scenario collects so much error that shows bigger deviation than the 7 ms scenario. The 10 ms scenario has another magnitude of error, and thus cut off the diagram after the first second.

This shows that my proposed plugin can be used to measure the effect of network latency on several different KPIs. It also shows that between around 10 ms of delay, the control scheme I used (or its parameters) can not be used to accurately control the arm.

4.2 Evaluation using ARIAC

The ARIAC competition involved a simulation of the infrastructure where teams would have to complete a set of *tasks*. The simulation infrastructure was built on top of Gazebo [1]

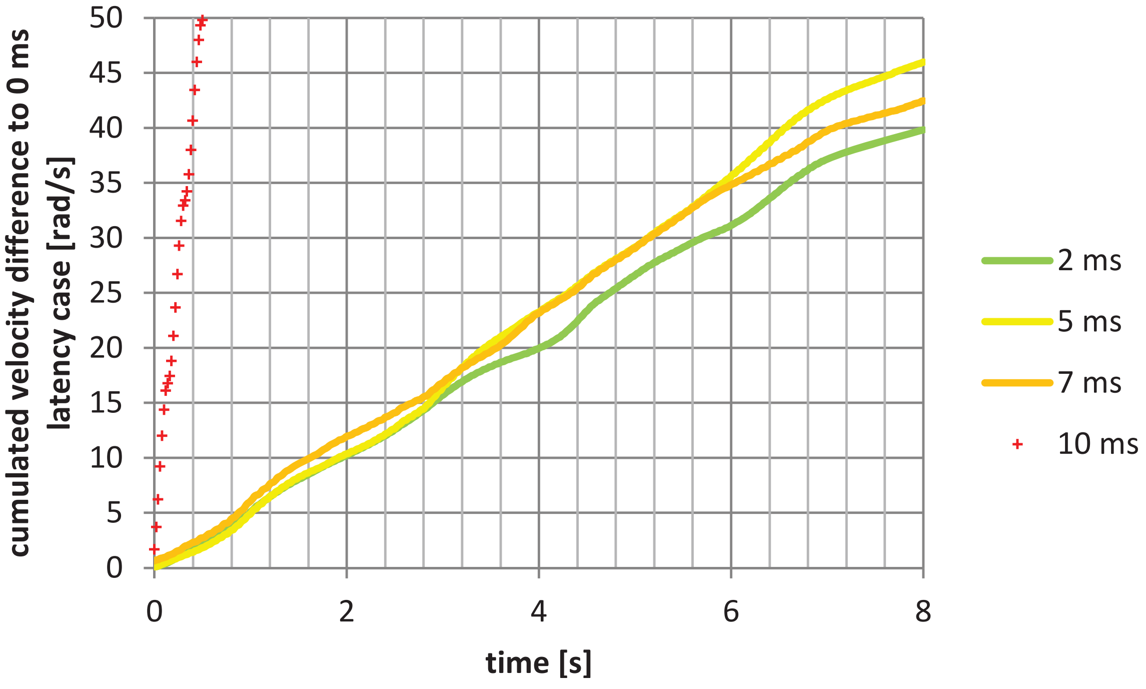


Figure 4.3: *The cumulated difference of the velocity commands comparing to the reference scenario*

and ROS [10]. The tasks were made to comprehend four specific areas: failure identification and recovery, automated planning, fixtureless environment, and plug and play robots. The tasks or challenges were explored with different simulation trials, which represent the configuration of the simulated environment as well as its goals. ARIAC tasks revolve around collecting a set of part pieces and placing them on a tray to be sent for assembling. A task during execution can be seen in Figure 4.4.. It shows the robot arm picking up a part from a conveyor belt.

The *assessment* during the competition and final round was performed with respect to cost and performance metrics. The cost metrics take into account the cost of the system, which was based on the types of sensors used, and the average system costs of all the teams. The performance metrics were based on completeness and efficiency. The completeness metric checked if all the requirements of the competition were completed perfectly i.e., all part pieces from all kits and orders were sent to assembling in the correct position and orientation. The efficiency metric was calculated by using time factors.

The solution [48] I used for solving the ARIAC 2017 competition is available together with the ARIAC evaluation scripts.

To fulfill the ARIAC tasks, this control system is designed to have three major actuator components:

- Order scheduler: orchestrate the execution of orders and its kits. Essentially, it decides on a set of actions based on the status of the working environment

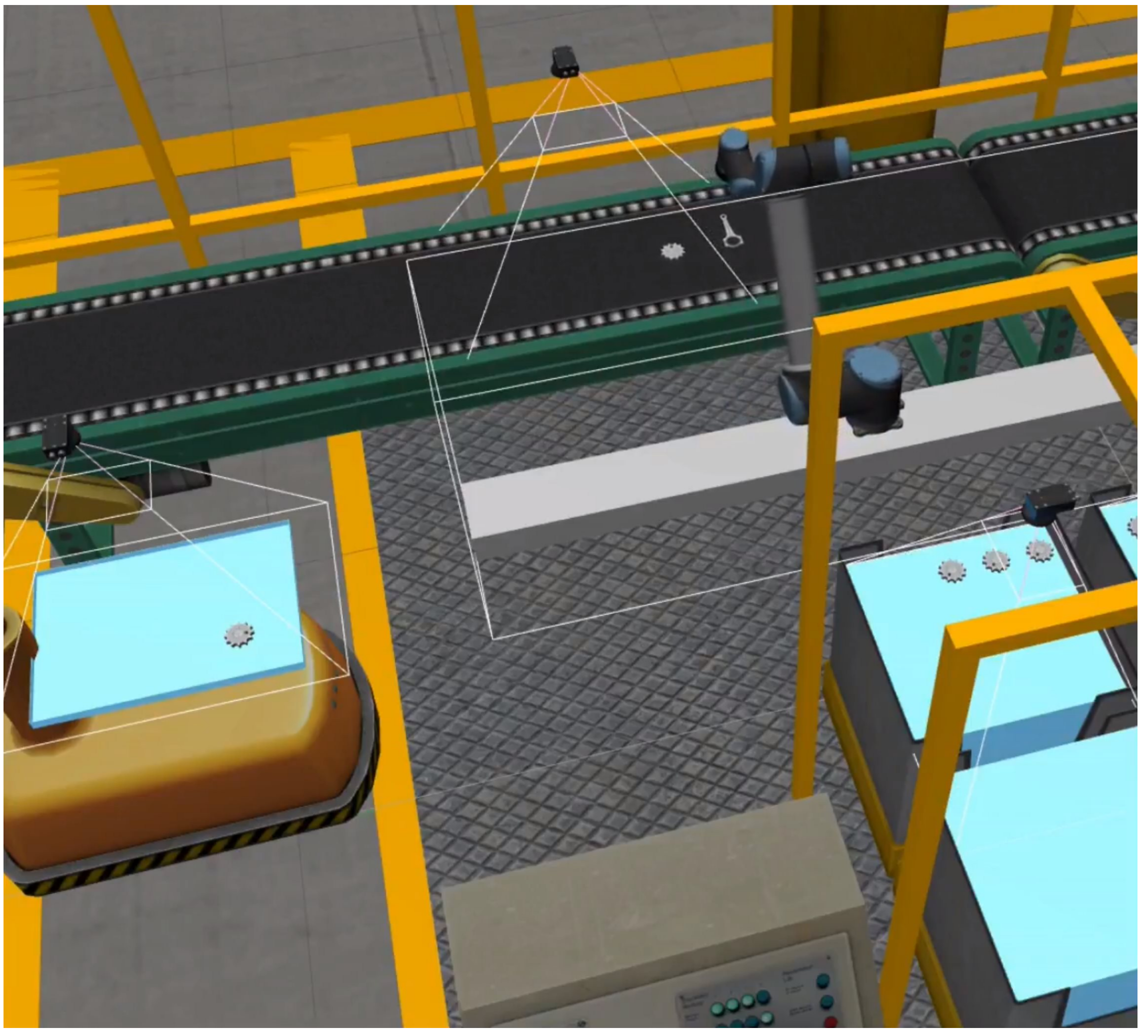


Figure 4.4: *ARIAC environment during the execution of a task*

- Arm and gripper actions: provides a set of abstract robot actions (e.g., move tooltip up, go next to AGV) and access to gripper control (activate and deactivate)
- Inverse kinematics solutions: receives the desired position and compute the set of joint values to achieve the required configuration.

I evaluated my proposed method on various Key Performance Indicators (KPIs). The KPIs are partially the ones defined in the ARIAC scoring and some basic KPIs usually applied for evaluating CPS systems.

The KPI metrics of the ARIAC are shown at the end of each trial run and can be summarized as follows:

- **Total Score (TS)** - number of parts correctly delivered;
- **Total Process Time (TPT)** - period between the time the first order is issued

and the end of the trial;

- **Part Travel Time (PTT)** - amount of time that part pieces travel attached to the robot's gripper;

The Total Score is a compound of presence of required parts, correct position and orientation, and bonus points when all parts are present. In other words, the competition gives: one point for each required part piece placed on the tray; another point for each part piece with correct position and orientation; and finally additional n points when all required parts are present, where n is the number of expected part pieces. As an example, if a kit requires two piston rods and three gaskets, considering that all parts are placed on the tray but one was slightly wrongly positioned, it would receive 14 out of the 15 maximum possible.

Interestingly, there are some scenarios designed to be harder or even impossible to complete. For instance, scenarios that heavily rely on a low number of spare belt parts require faster execution, otherwise one would miss them. Moreover, there are scenarios designed with required part pieces that are just not available in the correct number. In the latter case, while the theoretical maximum score would be three times the number of parts, in practice the best score will be a bit lower.

The CPS-related KPI applied here is the cumulated difference of the velocity commands.

4.2.1 ARIAC, no latency scenario

Table 4.1. presents the results of the ARIAC performance metrics when running the solver with the 15 final trial configuration files.

It can be seen that not all the scores reached the maximum values. The highest priority of the used solution was to complete all tasks successfully, still scenarios F6, F7 purposely were designed to be impossible to complete. These configurations did not provide all pieces required to complete the trials. The idea was to test the planning robustness in such unexpected situation.

I consider the values here as a baseline for the evaluation of other scenarios. The solution finished second in the ARIAC competition, with the highest Total Score and Processing Time (but using a few more sensors than the winner to accomplish that), which means that it is a best-in-class state-of-the-art solution.

4.2.2 ARIAC with fixed bidirectional network delay (15 ms)

I introduced latency in the system in both the command writing and status reading direction and rerun the competition solver on the final configurations. Increasing the latency

Trial	Max score	Total Score		Part Travel Time [sec]		Total Proc. Time [sec]		Cumulated velocity error [rad/s]	
		No latency	15 ms	No latency	15 ms	No latency	15 ms	No latency	15 ms
F 1	6	6	6	9.1	9.9	20.9	22.8	1751	1618
F 2	9	9	8	20.3	25.3	54.8	61.8	3034	5681
F 3	18	18	14	26.3	26.9	79.7	62.8	4807	3672
F 4	24	24	22	36.2	37.4	113.7	115.8	5980	7205
F 5	24	24	15	53.8	34.0	166.8	114.8	7963	10190
F 6	30	21	16	46.9	52.9	131.8	144.5	12324	8707
F 7	30	23	17	49.4	49.1	135.8	123.8	8882	7215
F 8	18	18	7	33.7	120.2	93.8	210.8	5888	12896
F 9	6	6	6	9.4	9.4	27.8	26.8	4528	2078
F 10	15	15	12	22.4	24.6	60.8	65.7	5044	4095
F 11	24	24	20	43.0	42.9	116.8	103.8	6572	6533
F 12	15	15	15	20.9	21.6	111.8	113.8	5593	7428
F 13	18	18	17	27.4	30.3	62.8	66.8	4619	3969
F 14	24	24	8	34.8	110.4	124.8	171.9	6893	6196
F 15	24	24	23	36.3	39.9	152.8	113.8	7798	7280
Sum	285	269	206	469.9	634.7	1454.4	1519.6	91678	94764
Ratio [%]			77		135		104		103

Table 4.1: *Evaluation results for the ARIAC final trials in various network environments*

increases the deviance from the original trajectories as seen in section 4.1.

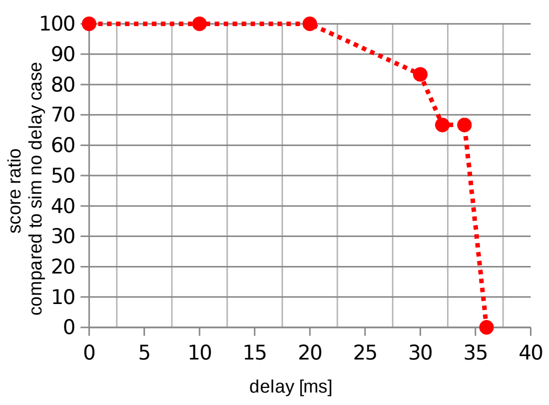
The introduced latency (15 ms in both directions) is high enough to start to deteriorate the total scores, but still makes the solver to execute the whole task without goal reach error. A lower value in the processing time compared to the baseline is usually due to the early return with unsuccessful execution of the whole task. We can see that the cumulated velocity error increases with 3% (to 103%) on average if the delay is increased. An increase in the velocity error translates into additional time required to reach the goal trajectory, which can be observed in the 4% (to 104%) increase in total proc. time and 35% (to 135%) increase in the part travel time. Note that the goal tolerances are increased and the PID values are also tuned to make the solver achieve as high total score as possible. In this way the velocity errors cannot be directly compared with the baseline scenario. The average of the total scores decreases to 77% of the baseline score, which is a sign that the final positioning of the parts becomes more and more inaccurate.

4.2.3 Cumulated ARIAC KPIs in the function of network latency

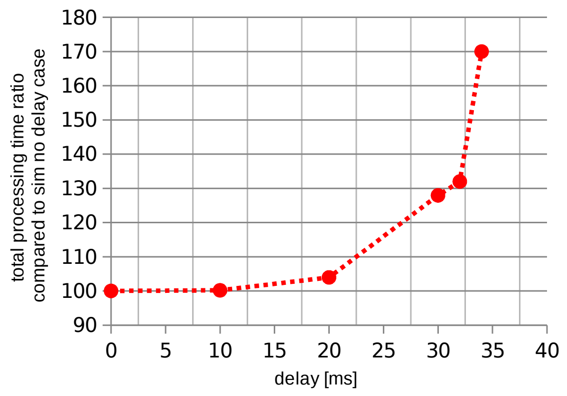
Figure 4.5 shows the ARIAC KPIs of simulated scenarios (i.e., control of the simulated robot over multiple simulated network latency). I cumulated the TS, TPT and PTT for the 15 final trial runs for the default local controller setup(i.e., no extra delay in the control loop). These cumulated values are considered as baseline (0 delay, 100% TS/TPT/PTT simulation) for the evaluation of other scenarios. The first observation that I can make is that the PTT values (see Figure 4.5c) are in the same magnitude in both the DT and the fully simulated case.

Analyzing the TS values (see Figure 4.5a) I can deduce that the simulated robot is robust to the simulated latency, because the score does not get reduced until after 20 ms of delay. The slight decrease of scores in function of increasing delay is the cumulated error of the gripper trying to pick up objects and place them to the AGVs. The accuracy of positioning begins to be sensitive to delay after 20 ms of latency. When TS drops to 0, it means that the robot arm could not position itself accurately enough within the goal tolerance to proceed further and the execution time outs after a while. This drop happens at 36 ms delay with the simulated scenario.

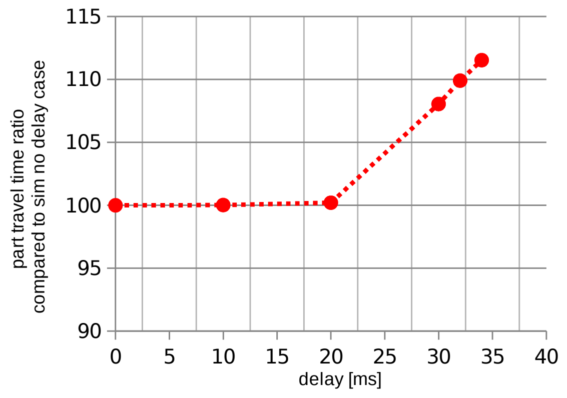
Considering the TPT and PTT values (see Figure 4.5b,4.5c) I can see similar deterioration of the KPIs as the TS after 20 ms of delay. The TPT grows to 170% of the baseline at 35 ms of latency. The PTT does not show that much increase in the values, because it only counts the time the gripper is engaged and when the delay is high, the worse accuracy of the arm causes it to fail grabbing parts.



(a) *score ratio*



(b) *total processing time ratio*



(c) *part travel time ratio*

Figure 4.5: *ARIAC KPIs in the function of network latency (no delay is the baseline)*

Chapter 5

Remote control of a complex platform

5.1 Overview

In the past few years, there has been an increasing demand from customers towards the manufacturing industry to provide more and more customized products [49]. Personalized production is one of the key motivations for manufacturers to start leveraging new technologies that enable to increase, for instance, the flexibility of production lines. High flexibility in general is needed to realize cost effective and customized production by supporting fast reconfiguration of production lines, as well as, easy application development. In typical industry applications, data packets are time-sensitive and require high reliability end-to-end. In the paradigm of Industry 4.0, the introduction of wireless technologies that ensure high reliability and low latency can help to address the flexibility needs. Ultra-reliable and low-latency communications (URLLC) is a new service category that will be supported in the 5G New Radio (NR). Application of such a wireless technology in manufacturing enables, for instance, to reduce cabling in a factory. In case the industrial applications are connected over wireless, there is a need to analyze the effect of network delay which is not an issue with cable-based connectivity. One of the most challenging applications in which the importance and capabilities of URLLC can be demonstrated is the low-level remote control of servos.

When introducing higher collaboration and adaptation capabilities into industrial applications such as robot arms and robot cell control, collaboration of a massive amount of servos may be required, making the use case even more challenging. In this demonstration a wide spectrum of the challenges that arise in a Industry 4.0 robot cell (e.g., servo control, collaboration, etc.) are demonstrated in a visually engaging way.

A hexapod can be considered as six 3 degree of freedom (DOF) robotic arms connected

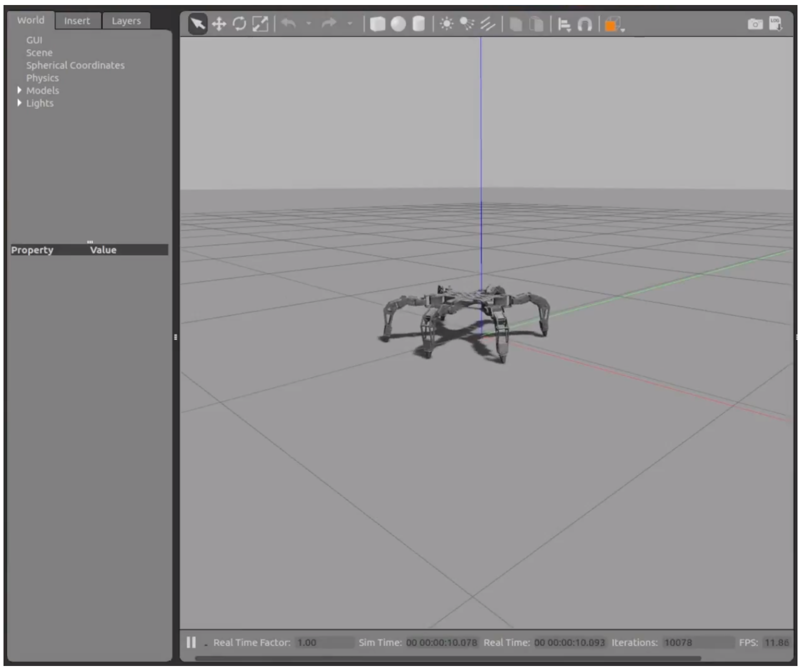


Figure 5.1: *Hexapod in the Gazebo simulator*

via a base link (see Figure 5.1.). In this demo, all servos at the 18 joints are controlled separately from a computer residing a wireless network hop away from the hexapod. This way the hexapod proves to be a good choice for visualizing the effect of synchronized collaboration that results in stable center position, while any glitch in the system results in jiggling of the platform.

5.2 Demonstrated features

5.2.1 Introducing network effects into hexapod control

The Robot Operating System (ROS) package of a hexapod robot [50] provided a regular position controller for the 18 joints (6 legs, 3 joints per leg). The control of the actuators are deployed locally on the robot i.e., there is no sensor or actuator delay at all. To introduce the possibility of analyzing the effect of networking into the system, the first task is to lift the deployment into a cyber-physical system (CPS). I used my plugin to introduce the network effects into this system.

To properly control the robot it needs to exchange information, such as velocity commands and encoder state information, between the controller and the arm in high frequency. The baseline system is completely steady after initialization of the simulation. Unless very small, the introduced network delay in both sensing and actuating processes results in jiggling of the whole robotic platform even at stationary status. The jiggling

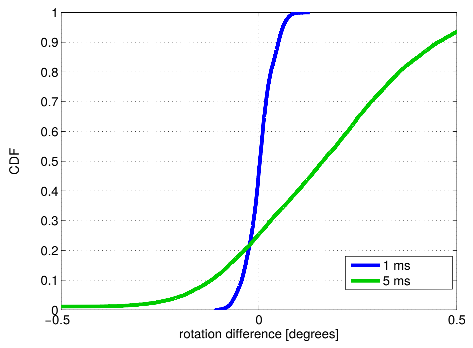


Figure 5.2: *CDF of the orientation differences for various scenarios*

results in small jumps of the robot resulting in deterring from the original orientation. When the hexapod starts to walk, the jiggling occurs during the movement as well and the robot ends up at different position compared to the baseline.

5.3 Effects of latency

The benefit of a low latency control loop provided by URLLC is clearly demonstrated even with visual inspection at the steady state of the robot. The robot jiggles in case of any higher than 1 ms control loop latency. In case of moving the clear indication of the benefit of low latency control loop can be seen on the trajectory of the robot (e.g., during a simple forward movement command). In case of low latency control loops, the trajectory is straight, while introducing latency in the control loop causes drifting away from the straight line. I quantified this drift by extracting the robot base link position and orientation around z axis from the simulation in every 0.1 sec. I calculated the differences of the orientation time series and applied a 1 sec moving average smoothing on the time series as the hexapod has a natural periodic waving in the base link. Bottom of Figure 5.2. shows the CDFs of the above time series for the 2 scenarios. I can see that in case of low latency control loop, the robot has a normal distribution with a mean around 0. In the 5 ms case, the distribution is shifted into the positive direction causing an drift of the robot.

Chapter 6

Conclusion and further work

In this paper, I proposed a plugin [46] 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.

I also utilized my proposed plugin in the ARIAC environment to examine the effect of latency on a more complex behaviour.

And in order to further test my plugin I also used it in a different scenario in which I added latency to a hexapod robot platform and evaluated its performance.

Further work

There are multiple avenues for further research:

- Evaluate the working mechanism of the system with the help of the ROS, gazebo and research communities.
- 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.

Bibliography

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

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

[3] G. Szabó, S. Rácz, J. Petö, and R. R. Aschoff. On the effects of the variations in network characteristics in cyber physical systems. *In Proc., 31st European Simulation and Modelling Conference*, Oct. 2017.

[4] Geza Szabo, Sándor Rácz, Norbert Reider, Jozsef Peto, and Rafael Roque Aschoff. Quality of control-aware resource allocation in 5g wireless access networks. In *19th IEEE International Symposium on "A World of Wireless, Mobile and Multimedia Networks", WoWMoM 2018, Chania, Greece, June 12-15, 2018*, pages 1–6, 2018.

[5] Geza Szabo, Sándor Rácz, Norbert Reider, and József Pető. Qoc-aware remote control of a hexapod platform. In *ACM SIGCOMM Conference Industry Demo Track*, SIGCOMM '18, 2018.

[6] József Pető. Analysing The Effects of Network Characteristics in Cyber Physical Systems, Students' Conference BME VIK. `https://tdk.bme.hu/VIK/ViewPaper/Halozati-karakterisztikak-valtozasainak`.

[7] Gazebo Tutorial. `http://gazebosim.org/tutorials?tut=guided_b1`.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[32] L. Monostori. Cyber-physical production systems: Roots, expectations and r&d challenges. *PROCEDIA CIRP*, 17:9–13, 2014.

[33] L. Ribeiro. Cyber-physical production systems' design challenges. In *2017 IEEE 26th International Symposium on Industrial Electronics (ISIE)*, pages 1189–1194, June 2017.

[34] Ahmed Joubair. What are Accuracy and Repeatability in Industrial Robots? https://blog.robotiq.com/bid/72766/What-are-Accuracy-and-Repeatability-in-Industrial-Robots.

[35] ISO: International Organization for Standardization. 1998. Manipulating industrial robots – Performance criteria and related test methods, NF EN ISO9283 , 1998.

[36] P. Danielis, J. Skodzik, V. Altmann, E. B. Schweissguth, F. Golatowski, D. Timmermann, and J. Schacht. Survey on real-time communication via ethernet in industrial automation environments. In *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, pages 1–8, Sept 2014.

[37] Mehdi Bennis, Mérouane Debbah, and H. Vincent Poor. Ultra-reliable and low-latency wireless communication: Tail, risk and scale. *CoRR*, abs/1801.01270, 2018.

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

[39] M. Ratiu and M. A. Prichici. Industrial robot trajectory optimization- a review. *MATEC Web Conf.*, 126:02005, 2017.

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

[41] 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.

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

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

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

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

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

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

[48] FIGMENT's team solution for the ARIAC competition. `https://github.com/Figment-Gprt/ariac-competition`, 2017.

[49] Yoram Koren, The Global Manufacturing Revolution: Product-Process-Business Integration and Reconfigurable Systems, John Wiley & Sons, Inc., 2010.

[50] ROS package providing Gazebo simulation of the Phantom X Hexapod robot. `https://github.com/HumaRobotics/phantomx_gazebo`, 2018.